THESIS FOR THE DEGREE OF DOCTOR OF PHILOSOPHY

Lock-free Concurrent Search

BAPI CHATTERJEE

Division of Networks and Systems
Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
Göteborg, Sweden 2017

Lock-free Concurrent Search
Bapi Chatterjee

ISBN: 978-91-7597-483-5
Series number: 4164
ISSN 0346-718X

Technical report 136D
Department of Computer Science and Engineering
Distributed Computing and Systems Research Group

Division of Networks and Systems
Chalmers University of Technology
SE-412 96 GÖTEBORG, Sweden
Phone: +46 (0)31-772 10 00

Author e-mail: bapic@chalmers.se, bhaskerchatterjee@gmail.com

Printed by Chalmers Reproservice
GÖTEBORG, Sweden 2017

Lock-free Concurrent Search

Bapi Chatterjee
Division of Networks and Systems, Chalmers University of Technology

ABSTRACT

The contemporary computers typically consist of multiple computing cores with high compute power. Such computers make excellent concurrent asynchronous shared memory system. On the other hand, though many celebrated books on data structure and algorithm provide a comprehensive study of sequential search data structures, unfortunately, we do not have such a luxury if concurrency comes in the setting. The present dissertation aims to address this paucity. We describe novel lock-free algorithms for concurrent data structures that target a variety of search problems.

(i) *Point search (membership query, predecessor query, nearest neighbour query) for 1-dimensional data: Lock-free linked-list; lock-free internal and external binary search trees (BST).*

(ii) *Range search for 1-dimensional data: A range search method for lock-free ordered set data structures - linked-list, skip-list and BST.*

(iii) *Point search for multi-dimensional data: Lock-free kD-tree, specially, a generic method for nearest neighbour search.*

We prove that the presented algorithms are linearizable i.e. the concurrent data structure operations intuitively display their sequential behaviour to an observer of the concurrent system. The lock-freedom in the introduced algorithms guarantee overall progress in an asynchronous shared memory system.

We present the amortized analysis of lock-free data structures to show their efficiency. Moreover, we provide sample implementations of the algorithms and test them over extensive micro-benchmarks. Our experiments demonstrate that the implementations are scalable and perform well when compared to related existing alternative implementations on common multi-core computers.

Our focus is on propounding the generic methodologies for efficient lock-free concurrent search. In this direction, we present the notion of help-optimality, which captures the optimization of amortized step complexity of the operations. In addition to that, we explore the language-portable design of lock-free data structures that aims to simplify an implementation from programmer's point of view. Finally, our techniques to implement lock-free linearizable range search and nearest neighbour search are independent of the underlying data structures and thus are adaptive to similar data structures.

Keywords: *Data Structure, Search, Range Search, Nearest Neighbour Search, Concurrent, Concurrency, Lock-free, Lock-based, Blocking, Non-blocking, Wait-free, Linearizable, Linearizability, Binary Search Tree, Linked-list, kD-tree, Lock-free-kD-tree, Amortized Complexity, Help-aware, Help-optimal, Language-portable, Synchronization*

To my Mother & my late Father

Acknowledgements

First of all, I would like to express my gratitude to my supervisor Prof. Philippas Tsigas for his constant support and guidance. Without his generous help, this thesis would not have been possible.

I also thank the graduate committee members - Prof. Thierry Coquand, Associate Prof. Marina Papatriantafilou, Prof. Jan Jonsson and former committee members Prof. Koen Claessen and Prof. Gerardo Schneider for their kind support and helpful suggestions during the discussions in my Ph.D. study follow-up meetings.

I am honored to have Prof. Pascal Felber as the opponent of my Ph.D. thesis defense. My sincere thanks to the members of the grading committee: Associate Prof. Paolo Romano, Associate Prof. Pedro Petersen Moura Trancoso, Prof. Stefanos Kaxiras, and Prof. Ulf Assarsson.

I also take this opportunity to thank my previous supervisors Prof. Subodh Kumar (Dept of CS&E, IIT Delhi, India) and Prof. Aparna Mehra (Dept of Mathematics, IIT Delhi, India), whose guidance during my master's studies helped me shape my thinking and motivated me enough to join the doctoral studies.

I am extremely grateful to the Swedish Foundation for Strategic Research (SSF) who funded me as a full-time Ph.D. student for five years in the research project "Software Abstractions for Heterogeneous Computers (SCHEME)". I thank the project co-leaders Prof. Per Stenström and Prof. Ulf Assarsson and other members in the project SCHEME with whom I truly enjoyed my collaboration.

Next, I would like to thank everybody in the Distributed Computing and Systems group, of which I am proud to be a member: Amir, Aras, Bashr, Charalampos, Elad, Hannah, Ioannis, Iosif, Ivan, Olaf, Magnus, Thomas, and Vincenzo.

I thank former members of the group, with whom I spent some of the most enjoyable moments during my Ph.D.: Andreas, Daniel, Farnaz, Giorgos, Oscar, Paul, and Valentin. My special thanks to my former office-mates, Nhan and Zhang, for the fun time and interesting discussions on topics confined to an

uncountable number of domains. I also wish all the best to the youngsters in the group.

It will remain incomplete without mentioning former members of the department: Bhavishya, Chien-Chung, and Madhavan, with whom I shared not just many vegetarian meals but also some unforgettable moments in Göteborg.

I also take this opportunity to thank the staff and the Ph.D. students at the Department of Computer Science and Engineering where I am privileged to be a member. It is thanking them for their efforts to make the department such a great place to work. I would like to especially thank Eva, Rebecca, Peter, Tiina, and Tomas for always being helpful and responsive.

Bapi Chatterjee
Göteborg

Contents

Contents and Publications

The focus of this thesis is on the design and implementation of concurrent lock-free search data structures drawing from the contents of the first four papers listed below. Contents in the Chapter 1 draw the description of multi-core and heterogeneous systems from the paper V and VI. The Chapter 2 presents the basic notions required for the descriptions in the later chapters. The Chapter 3 is based on the paper II. The Chapter 4 is based on a revised and a modified version of the lock-free internal BST algorithm published in paper I. The Chapter 5 is based on an extended version of the paper III. The paper IV is extended in the preparation of the Chapter 6. We discuss the overall thesis and present the general conclusions in the Chapter 7. Following is the list of papers published during Ph.D.:

I **Bapi Chatterjee**, Nhan Nguyen and Philippas Tsigas, "*Efficient Lock-Free Binary Search Trees*", In the Proceedings of the 2014 ACM Symposium on Principles of Distributed Computing (PODC 2014), 2014.

II **Bapi Chatterjee**, Ivan Walulya and Philippas Tsigas, "*Help-optimal and Language-portable Lock-free Concurrent Data Structures*", In the Proceedings of the 45th International Conference on Parallel Processing (ICPP 2016), 2016.

III **Bapi Chatterjee**, "*Lock-free 1-Dimensional Range Queries*", In the Proceedings of the 18th International Conference on Distributed Computing and Networking (ICDCN 2017), 2017.

IV **Bapi Chatterjee**, Ivan Walulya and Philippas Tsigas, "*Concurrent Linearizable Nearest Neighbour Search in LockFree-kD-tree*", In the Proceedings of the 19th International Conference on Distributed Computing and Networking (ICDCN 2018), 2018.

V Daniel Cederman, **Bapi Chatterjee**, Nhan Nguyen, Yiannis Nikolakopoulos, Marina Papatriantafilou and Philippas Tsigas, "*A Study of the Behavior of Synchronization Methods in Commonly Used Languages and*

Systems", In the Proceedings of the 27th International Parallel and Distributed Symposium (IPDPS 2013), 2013.

VI Daniel Cederman, **Bapi Chatterjee** and Philippas Tsigas, "*Understanding the Performance of Concurrent Data Structures on Graphics Processors*", In the Proceedings of the 18th International Conference on Parallel Processing (Euro-Par 2012), 2012.

List of Figures

Part I

INTRODUCTION

1

INTRODUCTION

Chapter Abstract

In this chapter, we introduce the concurrent search problem. We describe the varieties of search and how concurrency affects the consistency of the results of such problems. We introduce the basics of concurrent data structures. We give brief descriptions of the succeeding chapters in the thesis and a roadmap of the remaining of the thesis.

1.1 Search Algorithms

1.1.1 Introduction to Search

Given a set of input objects, *searching* one or more that match certain criteria, is an everyday problem. Computationally, we perform search by fixing the criteria based on an *enumerable feature* or *attribute* of the given objects, which is commonly called a *key* (corresponding to an object). Typically, a key is a real number. Often a search can be based on more than one feature of an object. In that case, the key can be a real number tuple, and the search problem is also called a *multi-dimensional search*. The set of keys corresponding to the given input objects is commonly called the *dataset*, and accordingly a key is also called *data* or a *data-point*. Typically, the search problem is formulated in order to extract certain content information from the given dataset, and the criteria of the search are based on a set of keys. The formulated search is also called a *query* and the set of keys that the search is based on is called the *query set*. When the query set contains a single key, it is called a *query key*. The superset formed by all possible datasets in a given context of an application is called the *universe of keys*.

Given a query key or set, we can ask for various kinds of information from the dataset. Depending on the set of query keys and the type of information that we ask, the queries can be of various types. According to the requirements of query type and the maintenance of the dataset, the universe of keys can have a *partial order* and/or a *metric* defined over it. Let U be the universe of keys and

K be the dataset. In general, we can categorize a search problem among the following varieties.

(i) Membership Query: Let $k \in U$ be a query key. Does $k \in K$? Equivalently, in words, is there an object in the given set of objects with a particular key?

(ii) Predecessor Query: Let $k \in U$ be a query key. Let $\preceq$ denote the partial order in U. Find $k_1 \in K$ such that $k_1 \preceq k$ and $\nexists\, k_2 \in K$ s.t. $k_1 \preceq k_2 \preceq k$. Equivalently, find the object whose key is just less than (in the partial order of the universe of the keys) the query key. The type of queries such as find $k \in K$ s.t. $\nexists\, k_1 \in K$ and $k_1 \preceq k$ (or $k \preceq k_1$), or equivalently, find the object with smallest (or largest) key in the given set of objects, are also considered as a predecessor query.

(iii) Nearest Neighbour Query: For the nearest neighbour query, we need a metric or distance in the key universe. Let $d(k_1, k_2)$ denote the distance between two keys $k_1,\ k_2,\ \in U$. Let $k \in U$ be a query key. Find $k_1 \in K$ such that $d(k_1, k) \leq d(k_2, k)\ \forall\ k_2 \in K$. Equivalently, find the object whose key is at the smallest distance from the query key.

(iv) Range Query: Same as nearest neighbour query, we need a metric in the key universe for a range query. Let $d(k_1, k_2)$ denote the distance between two keys $k_1,\ k_2,\ \in U$. A range query can be of the following two types.

(a) Orthogonal range query: In this type, typically we have a *query range* as a closed and bounded rectangle[1] $[l, h]$ described by two keys $l,\ h,\ \in U$. l and h are called *lower limit* and *upper limit*, respectively. Let $[l, h],\ l,\ h,\ \in U$ be a query interval. Find all $k \in K$ s.t. $k \in [l, h]$. Equivalently, find all the objects in the given set of objects whose key falls in the query range.

(b) Circular range query: In geometric problems, range queries are usually circular, that is, the query range is described by a key as centre of a sphere[2] and a radius and the aim is to find the objects whose key falls in the sphere.

Some authors (e.g. see [80, Chap. 4, p. 485]) refer to the query of type (i) as a *point query*. However, in this dissertation, we shall refer to the first three type of queries collectively as a *point search*. There are some generalizations of

[1] In 1-dimensional problems, the rectangle becomes an interval.

[2] For two-dimensional keys the sphere becomes a circle, whereas in 1-dimension cases it is an interval on the real-line.

the above query types, for example, k-nearest neighbour query, which requires finding k objects from the given set, whose keys are closest to a given query key. There are some other types of queries, such as, *all-closest-pair query* [80] and their generalizations, however, we shall limit our discussion to the query types that we mentioned above. Furthermore, for a nearest neighbour query and its generalizations, approximate solutions are also very popular. However, in this dissertation, we will not delve into those.

Sometimes in an application, there are multiple queries of different types. In those cases, we shall use the term *set of queries* to address the collection of all the queries in the application.

Depending on the type of application, a search problem can be *static*, where the entire dataset is given once and for all before the start of the first query, or *dynamic*, where the application takes in *modifications* in the dataset i.e. data can be *added* or *removed* from the dataset during the lifetime of the application. Naturally, in static search problems the focus is on improving the efficiency of the query only, whereas, in dynamic search problems, the efficiencies of addition and removal of data are equally important.

Thus, a search problem has: (a) a dataset, static or dynamic, (b) a set of queries, and (c) set of operations to add or remove data, collectively called *modifications*, in the dataset. A *search data structure* is the main tool to efficiently solve a search problem.

1.1.2 Search Data Structures

The theory of search data structures is the study of arranging the dataset in the main memory or disk of a computer and facilitating efficient queries and modifications in the dataset. We refer to a query and a modification in a data structure as a *data structure operation* or just *an operation* if the context does not have any ambiguity. In section 1.1.1, we described the basics of a search problem. The fundamental aim of an algorithm for a search data structure is to extract the information required by a query by visiting as small a subset as possible of the given (or the available, in the case of a dynamic) dataset. This is made possible by storing extra information with the dataset. Typically, the augmented information includes the addresses of the data-points which makes it possible to navigate the dataset efficiently, and thus *pruning* (avoiding to visit) those subsets that potentially do not contain the information required for the query.

Depending on the static or dynamic nature of the dataset underlying an application, a search data structure can be categorized into *static* and *dynamic*. For static data structures, the focus lies on - (a) efficiently loading the dataset on to

the data structure and (b) efficient querying. Additionally, the storage space is also considered for efficiency. On the other hand, for dynamic data structures, the efficiency of each operation is equally important. Moreover, the efficiency of operations in dynamic data structures is typically valued over the efficiency of storage space. Obviously, the dynamic data structures cannot be efficiently based on arrays, which is the preferred way to store the entire dataset in case of static applications. In dynamic data structures, the cost of shifting data-points across and/or resizing the array is prohibitive. Therefore, for augmenting the extra information, mainly address of other data-points, we use *nodes*, which are packets of a key (or a set of keys), the address of other nodes in the data structure and some metadata related to the structure of the data structure. The address stored in nodes in a specific data structure is based on various considerations that come from the type of query that we intend to perform.

In the present dissertation, we are mainly interested in dynamic data structures. Henceforth unless explicitly mentioned, by a data structure we shall refer to a dynamic data structure. Accordingly, our description of the design of data structures shall be primarily confined to *node-based data structures*.

Now, considering the type of queries, for membership queries, a simple data structure that can be used is an *unordered list*, in which keys are arranged without any navigation order and thus each query may need traversing the entire data structure. However, building an unordered list is straightforward: a linear list of nodes, in which each node consists of a key and address of the *next* node in the list. The next fields form the *links* between nodes. There are two ends of the list, often called *head* and *tail* and the nodes are added at one of the ends only, say at the tail. Clearly, adding a new key in an unordered list is always $O(1)$. The most popular data structure for membership queries is a *Hash table* [27]. Typically, a hash table is constructed as an *array of buckets*. Each bucket is usually an unordered list. A hash table builds upon a *hash function* which maps a key to a bucket. A hash table provides fast membership queries: the average case cost of each query is $O(1)$ irrespective of the number of keys in the data structure. However, unordered lists and hash tables cannot be used efficiently for other types of point search queries i.e. predecessor queries or nearest neighbour searches or for that matter range searches as well. The reason is that the augmented information in the data structure - the address of the next node in each node, which does not necessarily contain the key related in any order to the key in the current node - cannot be used for *order* or *proximity* of the keys.

For predecessor queries, as mentioned before, we need partial order. Here, a simple data structure is a *sorted or ordered list*, commonly called a *linked list*, which has the same node structure as an unordered list, but the next field in a

node always points to the node containing the successor (or predecessor) of the key in the current node. In a linked list the average case complexity of each operation is $O(n)$. Because each node has a single link, a linked list is also called a *singly linked list*, and it facilitates traversing the list in one direction only. Alternately, a *doubly linked list*, which contains a *pre* link in addition to a next, facilitates traversal in the list in both directions (*forward* towards successors and *backward* towards predecessors). However, the asymptotic cost of data structure operations in a doubly linked list is the same as that in a singly linked list.

The most popular data structure for predecessor queries is a *binary search tree (BST)*, in which each node has two links emanating from it, commonly called the left and the right child links, and the nodes connected to these links are called the left and the right child, respectively. The partial order of the key universe is used to *hierarchically* arrange the subsets of the dataset in a BST. The order of hierarchical arrangement is also called the *symmetric order* of the BST. In a BST the average case complexity of each operation is $O(\log n)$. The efficiency of a BST significantly depends on its structure in terms of the distribution of nodes along the left and the right child links of each of the nodes. The distribution is expected to be *balanced* on the two sides. To balance a BST, a number of criteria and techniques have been proposed in the literature. For example, *AVL tree* [1], *Red-black tree* [43], *rank-balanced tree* [45] and many other BST designs have their specific criteria for balancing the BST. BSTs offer generalization in terms of the number of links emanating from a node. Thereby, we have data structures like *k-ary tree* [34], *B-tree* [13] and their variants.

An essential generalization of the above mentioned data structures come from the dimension of the data. So far as the membership queries are concerned, we can use a hash table with a hash function for multidimensional data, such as *Locality Sensitive Hashing* [56]. For predecessor queries, a simple tool to map multidimensional data to one dimension is *Z-order* or *Morton order* [69]. The z-value of a multi-dimensional key is calculated by interleaving the binary representations of its coordinates. However, for multi-dimensional data the predecessor query is not an important query, the reason being that there is no natural *linear order*. For multidimensional data, a typical and important query is the nearest neighbour query. For a nearest neighbour query, a generalization of BST is the *kD-tree* proposed by Friedman et al.[37]. A collection of sophisticated data structures for performing nearest neighbour query in datasets with very high dimensions can be found in Samet's book [80].

In case of the range queries, in one dimensional data, a data structure that facilitates predecessor queries can be directly used. However, for efficiency, a data structure with *fat nodes*, that is one in which a node contains multiple keys,

is preferred because it reduces the number of nodes to be read to collect the keys belonging to the query range. A popular data structure for one-dimensional orthogonal[3] range queries is the *B-tree* [13]. The B-trees have multiple keys in its fat nodes and multiple links emanating from nodes for traversing down the tree. For multidimensional data, the most popular data structure for range search is the R-tree [44].

The literature of search data structures is huge and it is difficult to come up with a survey that can reach each one of them. Additionally, the search data structures are used to design *databases*, where they are commonly known as *index structures* or just *indexes*. Therefore, a comprehensive study of the field of search data structures remains incomplete without covering the database indexes. One-dimensional data structures are commonly part of any celebrated book on algorithms [27, 41, 4]. For multidimensional data structures a comprehensive collection can be found in [80].

Knuth in his seminal book [60, Chapter 3, p. 422] provided an earlier account of search algorithms existing in the literature. Therein, the history of search algorithms can be traced back to the binary search introduced by John Mauchly in 1946 [65], who constructed the first fully functioning electronic digital computer ENIAC (Electronic Numerical Integrator And Computer). Today, the landscape of search problems encompass a countless number of applications. These applications often require dynamic real-time updates of the underlying datasets. We already mentioned the basics of the dynamic search data structures. At the same time, today's multi-core computers come packed with multiple computing cores with high compute power. These machines provide very good platform for the concurrent asynchronous shared memory systems, which naturally fit for the applications that take in dynamic updates. Such applications and compute platforms have escalated the demand for scalable concurrent search data structures. In the next section, we provide the basics of concurrent algorithms for search data structures.

1.2 Concurrent Data Structures

1.2.1 Synchronization Algorithm

It is widely known that in single-core compute processors higher computation performance could be achieved only by way of increased clock frequency and that would come with the drawbacks ranging from large power requirements

[3] In one dimension, an orthogonal range can be converted to a circular range without changing co-ordinates system.

to unmanageably high heat dissipation. A ubiquitous device to solve the optimization problem of maximizing the computation under the constraints of power-consumption and heat-dissipation in contemporary computers is a multi-core Central Processing Unit (CPU). To further enhance the processing of data-parallel components in a program, multi-core CPUs are supported by many-core co-processors such as Graphics Processing Units (GPUs). These coprocessors can run hundreds of lightweight processing threads concurrently. The processing units in such co-processors often have their independent memory hierarchy to store the data to be processed close to compute units. The computer architecture comprising of CPUs and co-processors like GPUs i.e. heterogeneous compute units with added heterogeneity in memory hierarchy is known as *heterogeneous computer architecture*.

However, harnessing the maximum available processing power in the machines equipped with multi-core and many-core processors is not plain sailing. An algorithm may not always lend itself for easy parallelization, specially when it involves *concurrency* of multiple threads to modify shared data. For example, consider the case of two threads that increment a shared counter which broadly happens in three compute steps - (1) read the counter (2) increment the counter (3) store the incremented value at the shared memory word. We can list out all possible $\binom{6}{3} = 20$ valid interleavings[4] of the instructions by the two threads. But it is not hard to see that there can be only 2 valid interleavings which will increase the counter meaningfully. The problem of finding a valid interleaving that can resolve the conflict between concurrent threads to produce a correct solution of a problem involving concurrent access of shared memory is called *synchronization*. An algorithm is called a *synchronization algorithm* that solves the synchronization problem in a concurrent program. A synchronization algorithm becomes more complex with increasing number of shared memory words to be modified in order to accomplish an operation. And, the challenge of optimizing such an algorithm is immense because there is no guarantee provided by the implementation platform about the relative speeds of the threads.

Now, before describing the synchronization algorithm, we need to specify the model of the shared memory system used for the concurrent algorithms in this dissertation. In the chapter 2 we shall describe the same in detail, but for a self-contained discussion here, we briefly outline it. We consider a concurrent shared memory system in which a finite set of *threads* communicate asynchronously by reading and writing on *shared memory words*. Asynchrony

[4] There are altogether 6 instructions and a thread executes 3. We pick-up any 3 out of 6 in $\binom{6}{3}$ ways for one thread and 3 out of remaining 3 in $\binom{3}{3}$ for the second thread. Counter will be increased by 2 counts only if the first thread performs all its instructions in order before the second thread starts and does the same or vice-versa.

implies that there is no assumption on the relative speeds of the threads. Each thread is considered to be executing a sequential program over a finite set of variables as memory words each of which consists of one or more bits. A variable is *local* to a thread if the thread holds an exclusive access to it, and is *shared* when two or more threads can access it. The sequential program consists of steps and a step can contain computation on local variables and at most a single access to a shared variable.

Access to a shared-variable happens by means of an execution of an *atomic synchronization primitive* or *synchronization primitive*. By *atomic* it indicates that it takes place in an *undividable* unit of time. Some of the widely used synchronization primitives in synchronization algorithms are listed here. The shared-variables (`Sv`) passed as arguments are passed by reference whereas the value type (`Vt`) arguments are passed by value.

1. Read

```
Vt read(Sv r)
{
 return r;
}
```

2. Write

```
void write(Sv r, Vt v)
{
 r = v;
}
```

3. Test-and-set (`TAS`)

```
Vt TAS(Sv r, Vt v)
{
 initial = r;
 r = v;
 return initial;
}
```

4. Compare-and-swap (CAS)

```
bool CAS(Sv r, Vt old, Vt new)
{
 if(old = r)
 {
  r = new;
  return true;
 }
 else
 {
  return false;
 }
}
```

The synchronization primitives read, write, TAS and CAS are natively provided by almost all of the multi-core and many-core architectures available in market. Some multi-core architectures, for example MIPS [49], also provide Load-linked/Store-conditional (LL/SC). LL/SC is actually a pair of instructions used together. At every LL(Sv old, Vt r), the value of r is not only written at old but also written at a linked register.

5. Load-linked LL

```
void LL(Sv old, Vt r)
{
 old = r;
}
```

6. Store-conditional `SC`

```
Vt SC(Vt new, Sv r)
{
▷ assuming that r was read as old by the last LL
 if(r = old)
 {
  r = new;
  return old;
 }
 else
 {
  return 0;
 }
}
```

Now coming back to the synchronization algorithms, they are categorized in two classes - (a) *blocking* and (b) *non-blocking*. At the heart of a blocking synchronization algorithm is the *critical section* which is a piece of code that needs to be executed by multiple concurrent threads and in which threads need to access shared memory words. If the critical section is executed by more than one thread, it can result in unexpected and unwanted return by the concurrent program. Naturally, a thread is allowed to take an only finite number of steps to execute a critical section. To implement a critical section in an algorithm, a simple and straightforward way is using the construct of *lock*. Before entering a critical section, a thread *acquires a lock*, and when coming out of the critical section it *releases the lock*. In the critical section, the thread is said to be *holding the lock*. A lock can not be acquired by more than one threads and thus the critical section at any time is executed by no more than one thread.

Many efficient designs of locks are available in the literature. A simple test-and-set (TAS) lock works as described below:

```
while(!TAS(Sv lock, Vt 1)){}
critical_section{}
lock = 0;
```

The shared-variable `lock` is called to be *acquired* by a thread if the thread

succeeds to set 1 at it given that it was initially 0. Setting the variable `lock` back to 0 is called *releasing* the lock. Now, if the number of threads increases and the memory bus gets locked by many threads continuously, the performance drops drastically. Therefore, a better and optimized version can be written as

```
while(lock!=0 || !TAS(Sv lock, Vt 1)){}
critical_section{}
lock = 0;
```

In the revised format a thread reads the variable `lock` first and if it is available then only tries to acquire the lock. Using the CAS synchronization primitive also we can construct a lock as below

```
while(!CAS(Sv lock, Vt 0, Vt 1)){}
critical_section{}
lock = 0;
```

Nevertheless, the above formulations of locks essentially bring busy waiting because of high degree of contention for the variable `lock`. A way to lower the contention on a lock is to use some *backoff function*. A backoff function tells a process to wait for a certain amount of time before checking again. A popular backoff function is the *exponential* backoff in which the backoff time is increased exponentially for every failed attempt to acquire the lock. But still the problem to tune the backoff function persists in the sense that some processes might have to wait for much longer than other processes before acquiring the lock.

In the above lock designs it is easy to notice that the available cores in a processor are used for useless works during *busy waiting* and so it is imperative to minimize that. An alternative method of lock implementation is *queue lock* in which when a process fails to acquire a lock it adds itself to a queue associated with the lock and does a context switch so that another thread can use the processor while it is waiting to acquire the lock. After a thread finishes its critical section and releases the lock, it notifies the next thread waiting in the associated queue about it and context switches itself. Unfortunately, the cost of context switching in multiprocessors could be high and therefore the queue locks get outperformed by the `TAS` or `CAS` locks.

Not just the above problems that arise with locks but also there is problem of *lock conveying* described as the situation in which a thread acquiring the lock gets preempted by the thread scheduler. This causes other threads to wait longer than necessary because the thread that got swapped could not release the lock and this results in overall slowdown of the entire program. A related problem is that of the *priority inversion* when a high priority thread has to wait for a low priority thread holding lock. This problem can be solved by increasing the priority of the lower priority thread that holds the lock to a certain high ceiling priority or to the priority of the higher priority thread. The first method is called the *priority-ceiling-protocol (PCP)* and the second method is called the *priority-inheritance-protocol (PIP)*.

Having described the issues associated with a lock, we can clearly see that the blocking synchronization algorithm are often uninteresting. Alternatively, a non-blocking synchronization algorithm does not have a critical section in its design and thus is not vulnerable to the problems associated with a lock. Essentially no thread needs to be blocked from executing any (atomic) *step* on any shared memory word in a non-blocking synchronization algorithm. Thus, the biggest advantage is that no thread can die or get delayed while holding a lock over a critical section. This brings a progress guarantee attached to a non-blocking algorithm.

The basic idea is that instead of holding any kind of lock, the threads copy the value of shared-variable using an atomic read primitive to its local variable, makes the changes as needed locally and then changes the shared-variable using a stronger synchronization primitive like `CAS` in one atomic step. In case it fails to successfully perform `CAS`, because another thread would have applied its own changes on the shared-variable since this thread read the value, it retries after updating its local value. This way a greater fault-tolerance comes that enables avoiding issues associated with blocking synchronization.

Based on the kind of progress guarantee, non-blocking algorithms are categorized in the following three classes:

(a) **Wait-free:** These algorithms guarantee that if a thread remains alive and taking steps, it can always finish its operation regardless of other threads getting delayed or even failing, and thus all the non-faulty threads are guaranteed to complete their operations in a finite number of steps.

(b) **Lock-free:** These algorithms guarantee that at least one of the non-faulty operation will complete its operation in a finite number of steps. Clearly, the lock-free algorithms provide weaker progress guarantee than the wait-free algorithms.

(c) **Obstruction-free:** These algorithms provide weakest progress guarantee - if a non-faulty thread works in isolation, it is guaranteed to complete its operation in a finite number of steps.

In the chapter 2, we shall detail the methodology to prove the correctness of a synchronization algorithm. However, for the sake of reference, here we briefly mention about the consistency notion of a synchronization algorithm called *linearizability*. linearizability ensures that despite concurrent execution of operations by multiple threads, the operations should appear to take effect sequentially and thereby a correct behavior of the concurrent algorithm becomes equivalent to a correct behavior of a sequential algorithm. With that we now describe synchronization algorithm for search data structures, which we refer to as concurrent search data structures.

1.2.2 Concurrent Search Data Structures

In the section 1.1.2, we described the dynamic search data structures, which are built on nodes and links. Adding a new node or removing one requires modification of links. In fact, for an operation that removes a node, multiple links may need to be modified together. In a concurrent setting, these modifications create shared state access scenarios by multiple threads. Naturally, we need correct synchronization algorithms for correct implementations of concurrent search data structures. The multi-core and many-core architectures, with more than a single level of hierarchy in the memory access, add further complexity as the size of the cache memory comes to play important role in the performance of such data structures. Thus the design, analysis, and implementation of *concurrent search data-structures* on multi-core and many-core architectures is an immensely challenging task.Depending on the choice of synchronization algorithm, the concurrent data structures can be classified into blocking and non-blocking classes as explained below.

(A) Blocking concurrent data structures

The design of a blocking concurrent data structure is primarily based on *locks*. Modification of a link becomes a critical section and guarded by a lock. If multiple links need to be modified for an operation, they lead to the possibility of either being guarded by a single lock, called *coarse-grained locking*, or multiple locks, called *fine-grained locking*. In case of fine-grained locking, both acquiring and releasing locks need to follow a pre-specified order, to avoid malformation of the data structure. Another issue with fine-grained locking is *deadlock* - two different threads hold two different locks and wait for each other to release

their locks before releasing their own. Thus, designing a correct fine-grained locking based concurrent data structure is a deft task.

Design of lock-based concurrent data structures has now come of age, and many libraries of lock-based concurrent data structures are available. Still, as described before, they come with all the drawbacks associated with blocking synchronization algorithms. In addition to that, another well-known problem that occurs in these data structures is that of *lack of composability* of two or more concurrent data structures that use blocking synchronization. Although the individual concurrent data structures could not have the issue of deadlock as described before, in the composed one two threads could keep on waiting for each other to release their respective locks while holding their own locks.

Ordinarily, locks suffer from poor scalability and that also gets transferred to the concurrent data structure using those locks. Thus, in their entirety, the problems with lock-based concurrent data structures make it important to design and implement concurrent data structures which do not use locks i.e. non-blocking concurrent search data structures.

(B) Non-blocking concurrent search data structures

As described before, an alternative design of concurrent search data structure can be based on non-blocking synchronization. Accordingly, depending on the kind of progress guarantee associated with the non-blocking data structures they can be categorized into wait-free, lock-free and obstruction-free classes. The various levels of progress guarantee come from the design aspect of taking care of slow non-faulty threads when they attempt to modify shared links of a concurrent data structure. Often an operation in a concurrent data structure needs more than a single shared link to modify. As there is no assumption on the relative speeds of the threads, if a thread gets delayed, other threads in order to progress without getting blocked in any manner actually *help* the delayed thread. This is called *helping mechanism* in a non-blocking data structure. In the core of a helping mechanism is the idea that whenever in an operation more than one shared link is needed to be modified, the modifications should be done in an orderly manner and some indicator should be used in order to indicate the progress of the operation. However, helping should always be optimized as it makes the threads perform many atomic accesses to shared-variables even when that is not needed.

Helping in a non-blocking data structure actually decides its class. If all the delayed threads are helped by the faster concurrent threads, so that each of the non-faulty threads get guaranteed to complete their operations in finite number of steps, it becomes a *wait-free* data structure. In *lock-free* data structure,

most often only those slow threads are helped by concurrent ones, which actually create obstruction by way of having modified some of the multiple shared links required to be modified in an operation. Those threads whose operations do not obstruct any concurrent operation, mostly the threads performing read only operations, can still remain without completing their operation if they get delayed in a lock-free data structure. Finally, if we completely do away with the helping mechanism, we can have the guarantee of completion of operations by non-faulty threads only when they run in isolation, and in that case, the non-blocking data structures become *obstruction-free*. Given the complexity of wait-free data structures and weaker progress guarantee associated with obstruction-free data structures, most often we are interested in lock-free data structures for an application.

CAS is the most widely used synchronization primitive for modifying the shared-memory variables in a non-blocking synchronization method. One reason is that it is available in almost all the available multi-core and many-core architectures. By CAS, we indicate the synchronization primitive for a single shared-memory word. However, as we mentioned above, usually in a link-based data structure, we need to modify multiple links to do modifications (ADD/REMOVE). Thus, it requires a lot of effort to design a correct lock-free data structure based on CAS. To circumvent this issue, a number of work exist in the literature that use software constructs based on CAS. Double-compare-and-swap (DCAS), Multi-word-compare-and-swap (MCAS), Double-compare-single-swap (DCSS) are some of the well-known ones. A DCAS performs CAS over two shared-memory words. A MCAS is a generalization of DCAS. A DCSS performs CAS over a single word, however, it checks two words before performing the CAS. Henceforth, in the thesis we shall mean a single-word compare-and-swap by CAS.

With the benefits that the lock-free data structures bring with them, there comes a challenging issue - the *memory reclamation*. Next we discuss that.

(C) Memory reclamation in non-blocking concurrent data structures

Because memory is always limited and the running duration of an application based a dynamic node-based data structure may not be fixed before the application starts, the memory management becomes extremely significant. Allocating memory in terms of new nodes by individual threads in a concurrent data structure is a thread local operation and therefore is same as that in a sequential data structure. However, reclaiming memory occupied by a node is a completely different task in a lock-free data structure. For that matter, it is different from that even in a lock-based data structure. In a sequential data structure after a node

is removed, the single thread that removes the node returns it to the operating system or a memory pool if one is used. In a blocking data structure, the thread that holds a lock at the node being removed, does the same before releasing the lock. Thus it is straightforward in these cases.

On the other hand, consider the case of removing a node in a lock-free data structure. It is absolutely possible that more than a thread may hold the address of the node under remove in their local memory. For example, the threads that are either trying to help the remove operation by a concurrent thread or itself trying to execute the remove operation. Now, it is also possible that while holding the address of the node under remove, a thread gets pre-empted by the operating system and thus goes to sleep. Given that, when the sleeping thread wakes up, because there is no lock around, it can attempt to access the node whose address it was holding if it was not informed by the concurrent thread that actually freed the node under remove to the operating system. This can very likely cause the program to crash or create other unexpected errors. Obviously, carelessly freeing memory after removing a node in a lock-free data structure is not an option. Therefore, we need some specific mechanism for memory reclamation in lock-free data structures.

Mostly, recent papers on lock-free data structures focus on the main algorithm. Their reason is justified in the sense that the research literature of automatic memory reclamation, commonly known as *garbage collector*, is large and growing regardless of the research in lock-free data structures. The automatic garbage collectors are included in the virtual machine based high-level popular programming languages like Java and C#, and with every new release of the compiler the efficiency of garbage collector also typically gets improved. Still, many authors have presented tailor-made lock-free memory reclamation schemes and made them part of concurrent data structure libraries. In this dissertation, we shall not go into the memory reclamation of the presented algorithms and shall assume the availability of lock-free memory reclamation method. Our sample implementations in Java use the Java Virtual Machine (JVM) provided garbage collector whereas implementations in C++ use epoch based memory reclamation.

Finally, last but not the least, another challenge in designing non-blocking concurrent data structures is ABA problem. We describe it here.

(D) ABA problem in non-blocking concurrent data structures

We mentioned above that `CAS` is the most widely used primitive to design lock-free data structures. However, use of `CAS` brings one of the most interesting problems in a concurrent setup known as *ABA problem*. `CAS(Sv A, Vt B,`

Vt C) is not able to discover whether A was changed to B and then changed back to A between it was read and CAS is performed. In many situations, it causes problems in which a concurrent data structure can become malformed. On the other hand, using Load-Link/Store-Conditional LL/SC avoids ABA issue. Unlike CAS, it succeeds only if A, when changed in Store-Conditional to C, has not changed since it was read in Load-link as B. This is guaranteed because the value read buy a LL is always stored at a linked auxiliary register.

A second well-known mechanism to tackle ABA problem is via DCAS. A counter is packed with a shared-memory single-word length variable that needs to be modified. Thus, it makes an augmented variable. Now, to modify an augmented variable, we can use DCAS, and whenever the word changes its counter is incremented. That way, an ABA problem can not occur because of an incremented counter. However, as DCAS is not provided natively, this technique imposes a high cost.

1.3 Our Contributions

Further in the thesis, first in the chapter 2, we present the system model that we use to describe the presented algorithms. Thereafter, the detailed descriptions of the lock-free search data structures are given. Specifically, this thesis presents the following :

Lock-free one-dimensional point search: In the part II, we describe point search for one-dimensional datasets. In one-dimensional data, the nearest neighbour search is analogous to predecessor search. We present lock-free algorithms for linked-list and external BST in the chapter 3 and internal BST in the chapter 4. An important contribution of the chapter 3 is the notion of *help-optimality* which captures the optimization in the amortized complexity of operations in the data structures. Additionally, in the same chapter, we present a methodology for language-portable designs of lock-free data structures. To evaluate the efficiency of language-portability we implement the algorithms in both Java and C++ and compare their experimental performance with that of related existing implementations. In the chapter 4, we present an algorithm for lock-free internal BST. We present the amortized analysis of the algorithm. We prove that the presented lock-free data structures are linearizable.

Lock-free one-dimensional range search: In the part III, we describe a generic method for implementing linearizable lock-free range searches for one-dimensional datasets. The data structures linked-list, skip-list, and BST, which facilitate

predecessor query, are used as underlying data structures. We implement the algorithm and compare its experimental performance with that of an existing method of lock-free range search in a k-ary tree.

Lock-free multi-dimensional point search: In the part IV, we describe point search algorithms for multi-dimensional datasets. kD-tree is one of the most utilized data structure for nearest neighbour as well membership queries for multi-dimensional data. We present LFkD-tree- a lock-free design of the kD-tree. We present a linearizable nearest neighbour search method, which is generic and can be adapted to other multidimensional data structures. We also provide experimental results of an implementation of the presented algorithm.

2

SYSTEM MODEL AND PRELIMINARIES

Chapter Abstract

In this chapter, we present the definitions used in the algorithms. We also describe the consistency model of the algorithms and present an overview of proof techniques for linearizability. Finally, we describe the notion of contention measures that help in the amortized analysis of the lock-free algorithms.

2.1 System Model

Designing concurrent search data structures is a tedious task. However, proving the correctness of these algorithms is even harder. Rigorous proofs of these algorithms are as important as bug-free sample implementations. Now before describing the methodology to prove correctness, we define the system model used for the design and correctness proof of the algorithms in this dissertation.

Shared Memory System: We consider an *asynchronous shared memory system* $\mathcal{U}$ which comprises a set of word-sized *objects* $\mathcal{V}$ and a finite set of threads[1] $\mathcal{P}$ and supports *primitives* `read`, `write` and `CAS` (compare-and-swap). $\mathcal{U}$ guarantees that the primitives are *atomic* i.e. they take effect instantaneously at an indivisible time-point [51]. Each object $v{\in}\mathcal{V}$ has a unique *address*, commonly known as a *pointer to* v, denoted by $v{\cdot}$ref. `CAS`(v·ref, exp, new) compares the value of v with exp and on a match updates it to new in a single atomic step and returns true; else it returns false without any update at a. Let $|\mathcal{P}|{=}n$. Processes $p{\in}\mathcal{P}$ communicate by accessing the objects $v{\in}\mathcal{V}$ using a primitive. A *configuration* $\mathcal{U}_t$ of $\mathcal{U}$ specifies the value of each of $v{\in}\mathcal{V}$ and the state (values of local variables, etc.) of each of $p{\in}\mathcal{P}$ at time t. The *initial configuration* $\mathcal{U}_0$ represents the initial value of each of $v{\in}\mathcal{V}$ and the initial state of each of $p{\in}\mathcal{P}$.

[1] In this dissertation, we shall use threads instead of processes, which is also very common in describing a shared memory system. The reason is that our program implementation is multi-threaded instead of multi-processing.

Abstract Data Type: In this dissertation, by *multidimensional data* we mean a point from the real space $\mathbb{R}^d$ and by *distance* we mean Euclidean distance $||a,b||_2 \; \forall \; a,b{\in}\mathbb{R}^d$. Accordingly, if not mentioned explicitly, by *data* we shall mean a one-dimensional real number. Let $F^{\mathbb{R}^d}$ be the set of all countably finite subsets of $\mathbb{R}^d$. Let $k{\in}\mathbb{R}^d$ and $\mathbb{A}{\in}F^{\mathbb{R}^d}$. Let $\mathcal{B}{:=}\{\mathsf{true},\mathsf{false}\}$. With that, we shall define a universal *abstract data type* (ADT) $\mathcal{A}$ for all the data structures in this dissertation, which we specify as a set of mappings $\mathcal{M} = \{\textsc{Add}, \textsc{Remove}, \textsc{Contains}, \textsc{NNSearch}, \textsc{RangeSearch}\}$ as defined below:
ADT Operations:

1. $\textsc{Add} : \mathbb{R}^d{\times}F^{\mathbb{R}^d} \mapsto \mathcal{B}{\times}F^{\mathbb{R}^d}$ s.t. $\textsc{Add}(k,\mathbb{A}) = (\mathsf{true}, \mathbb{A}\cup k)$ if $k{\notin}\mathbb{A}$ **and** $\textsc{Add}(k,\mathbb{A}) = (\mathsf{false},\mathbb{A})$ if $k{\in}\mathbb{A}$.

2. $\textsc{Remove} : \mathbb{R}^d{\times}F^{\mathbb{R}^d}{\mapsto}\mathcal{B}{\times}F^{\mathbb{R}^d}$ s.t. $\textsc{Remove}(k,\mathbb{A}) = (\mathsf{true},\mathbb{A}/k)$ if $k{\in}\mathbb{A}$ **and** $\textsc{Remove}(k,\mathbb{A}) = (\mathsf{false},\mathbb{A})$ if $k{\notin}\mathbb{A}$.

3. $\textsc{Contains} : \mathbb{R}^d{\times}F^{\mathbb{R}^d} \mapsto \mathcal{B}{\times}F^{\mathbb{R}^d}$ s.t. $\textsc{Contains}(k,\mathbb{A}) = (\mathsf{true},\mathbb{A})$ if $k{\in}\mathbb{A}$ **and** $\textsc{Add}(k,\mathbb{A}) = (\mathsf{false},\mathbb{A})$ if $k{\notin}\mathbb{A}$.

4. $\textsc{NNSearch} : \mathbb{R}^d{\times}F^{\mathbb{R}^d}{\mapsto}\mathbb{R}^d{\times}F^{\mathbb{R}^d}$ s.t. $\textsc{NNSearch}(k,\mathbb{A}) = (a^*,\mathbb{A})$ where $a^* \in \mathbb{A}$ **and** $||a^*,k||_2 \leq ||a,k||_2 \forall a{\in}\mathbb{A}$.

5. $\textsc{RangeSearch} : \mathbb{R}^d{\times}\mathbb{R}^d{\times}F^{\mathbb{R}^d}{\mapsto}F^{\mathbb{R}^d}{\times}F^{\mathbb{R}^d}$ s.t. $\textsc{RangeSearch}(x,x',\mathbb{A}) = (S,\mathbb{A})$ where $S{\in}F^{\mathbb{R}^d}$ **and** $\forall k{\in}S,\; k{\in}\mathbb{A} \wedge x_i \leq k_i \leq x'_i$ where $1 \leq i \leq d$.

Please note that in case of one-dimensional data, $d = 1$. Furthermore, in this dissertation we consider only orthogonal ranges that are given by two corner points as $[x,x']$.

Data Structure: A *data structure* Υ stores points from a dataset $\mathbb{A}{\in}F^{\mathbb{R}^d}$. The *state* of Υ in configuration $\mathcal{U}_t$, denoted Υ_t, stores points from $\mathbb{A}_t{\in}F^{\mathbb{R}^d}$. For an unbounded and dynamic design, Υ is constructed using *nodes* and *links* that are assembled of the objects $v{\in}\mathcal{V}$. Υ may contain instances, and pointers thereto, of other classes to support a specific algorithm. A *node* nd instantiates a *class* **Node**, which provides a template for (contiguously) packaging required number of objects $v{\in}\mathcal{V}$ to store the values of the *members* of the class. Thus, the address of nd, more commonly known as *pointer to the node* or *node-pointer* nd, is the address of the first object of the object-packet assigned to it. We use a·m to denote the value of a member m of nd, if nd·ref=a. A *link* represents the address of a node, and thus is a node-pointer, is stored at a single object $v{\in}\mathcal{V}$. **Node** has a member ky which is immutable and represents a unique data-point

$k \in \mathbb{R}^d$. We denote nd by $\mathtt{Nd}(k)$ if nd·ref·ky $= k$. **Node** may have one or more links as its members depending on a specific design of Υ. **Node** may also have members, such as lock or some version-info-object, to facilitate synchronization of the concurrent threads accessing Υ. The access of Υ is availed by root - the address of a fixed *sentinel* node (We may have more than one sentinel nodes in a data structure). At any time $t \geq 0$, a node N is said to be *physically present* in Υ_t, if it can be *reached* following links starting from root. In many cases a node is removed from a lock-free data structure using multiple CAS steps. One of these steps are known to be *logical remove* step. At any time $t \geq 0$, a node N is said to be *present* in Υ_t, denoted by $\mathsf{N} \in \Upsilon_t$, if it is physically present in Υ_t and is not logically removed.

Implementation: An *implementation* $\mathcal{I}_\mathcal{O}$ of the ADT $\mathcal{A}$ is an algorithm, which implements mappings $\mathcal{O} \subseteq \mathcal{M}$ using *operations* on Υ. We call the implementation *full* if $\mathcal{O} = \mathcal{M}$, otherwise it is called *partial*. We assign the operations same name as its corresponding mapping. Thus, a mapping $op(k, \mathbb{A})$, where $op : \mathbb{R}^d \times F^{\mathbb{R}^d} \mapsto \mathcal{B} \times F^{\mathbb{R}^d}$, $k \in \mathbb{R}^d$ and $\mathbb{A} \in F^{\mathbb{R}^d}$, is implemented by an operation $op(k)$ that outputs true or false and makes appropriate changes in Υ storing $\mathbb{A}$. A NNSEARCH$(k, \mathbb{A})$ is implemented by NNSEARCH(k), which outputs a point $a^* \in \mathbb{R}^d$ according to the mapping definition. Similarly, a RANGESEARCH$(x, x', \mathbb{A})$ is implemented by RANGESEARCH$([x, x'])$ that outputs S which is a subset of $\mathbb{A}$.

Operation Steps: A thread $p \in \mathcal{P}$ performs an operation op as a set of *steps*. If op is large, often we group a subset of steps in op as a method, which is *called* from inside of op. A *step* $s = \langle v, g, h, p \rangle$, where g and h are the values of the object v before and after the execution of s, comprises at most one execution of a primitive and can contain some calculations over thread-local variables of p. The *execution-point* of s is the point on a real time-line where its atomic primitive takes effect. We denote the *invocation* and *response* steps of op by $s_i(op)$ and $s_r(op)$, respectively. The execution-points of $s_i(op)$ and $s_r(op)$, denoted by $t^i(op)$ and $t^r(op)$, are called the *invocation point* and *response point* respectively. $\mathcal{I}_\mathcal{O}$ also specifies the initial configuration $\mathcal{U}_0$.

Execution History: An *execution* α of $\mathcal{I}_\mathcal{O}$ is a (finite or infinite) sequence of steps performed by the threads $p \in \mathcal{P}$, starting from $\mathcal{U}_0$. A *history* $\mathcal{H}$ of α is its subsequence consisting of the invocation and response steps. A *subhistory* of $\mathcal{H}$ is its subsequence. A *thread subhistory* of $\mathcal{H}$, denoted by $\mathcal{H}|_p$ is its subsequence containing steps executed by a $p \in \mathcal{P}$. We call histories $\mathcal{H}$ and $\mathcal{H}'$ *equivalent*, denoted $\mathcal{H} \equiv \mathcal{H}'$, if $\forall p \in \mathcal{P}$, $\mathcal{H}|_p = \mathcal{H}'|_p$. In $\mathcal{H}$, a response step of an operation *op matches* an invocation step if the two are performed by the same thread. A history is called *sequential*, if the first step is an invocation and every invocation step, except possibly the last one, follows by a matching response step. We

assume that every history $\mathcal{H}$ is *well-formed*: $\forall p \in \mathcal{P}$, $\mathcal{H}|_p$ is sequential. An operation in a history is effectively the *pair* of its invocation and response steps. Let op_1 and op_2 be two operations in $\mathcal{H}$. We call op_1 *precedes* op_2 in $\mathcal{H}$, denoted $op_1 \underset{\mathcal{H}}{\longrightarrow} op_2$, if $t^r(op_1) < t^i(op_2)$. We call two operations op_1 and op_2 *concurrent* in $\mathcal{H}$, if neither precede the other. $\mathcal{H}$ is called *concurrent* if it contains at least one pair of concurrent operations.

Extension and Completion of History: We call an invocation s *pending* in $\mathcal{H}$, if $\mathcal{H}$ does not contain a matching response to it. An *extension* of $\mathcal{H}$, denoted $ext(\mathcal{H})$, is obtained by appending matching response steps to every pending invocation in $\mathcal{H}$. A *completion* of $\mathcal{H}$, denoted by $complete(\mathcal{H})$, is obtained by dropping the pending invocation steps, or equivalently, dropping the pending operations, from $\mathcal{H}$.

Consistent Sequential History: A *sequential specification* of $\mathcal{I}_{\mathcal{O}}$ is a set of sequential histories with some properties. Let $s_i(op), s_r(op) \in \mathcal{S}$, where $\mathcal{S}$ is a sequential history. Let $\Upsilon_{t^i(op)}$ and $\Upsilon_{t^r(op)}$ be the states of Υ at $t^i(op)$ and $t^r(op)$, which store the datasets $A_{t^i(op)}$ and $A_{t^r(op)}$, respectively. We call the operation *op consistent* with respect to the ADT $\mathcal{A}$ in $\mathcal{S}$ if the output arguments at the response and $A_{t^r(op)}$ satisfy the corresponding mapping definition of the ADT $\mathcal{A}$. The sequential history $\mathcal{S}$ is *consistent* if each operation in it is consistent.

2.2 Correctness and Complexity

2.2.1 Correctness

The concurrent data structures in which operations are executed by multiple threads concurrently are very hard to debug. The reason is asynchrony in the shared memory systems which makes it difficult to replicate a bug. But even before the implementation a concurrent data structure needs rigorous proof of correctness. Given a concurrent data structure that provides an implementation $\mathcal{I}_{\mathcal{O}}$ of $\mathcal{A}$, to prove its correctness, as mentioned by Lamport [61], we need to essentially prove two types of properties:

(a) *Safety:* Intuitively some bad things never happen.

(b) *Liveness:* Intuitively a good thing eventually happens.

Often, we refer to safety property as *consistency condition* and liveness property as *progress condition*. For the blocking data structures, wherein a critical section is protected by a lock around it, for proving the safety and liveness properties, it is essential to prove:

- *Mutual Exclusion* : Two threads executing concurrently can not be in their critical section simultaneously. This is a safety property.
- *Deadlock-freedom* : If a thread attempts to enter its critical section, then some thread, not necessarily the same one eventually enters its critical section. This is a liveness property.

Another stronger and quite desirable liveness property with respect to the critical-section based blocking data structures is Starvation-freedom.

- *Starvation-freedom* : A thread, that attempts to enter its critical section, must eventually succeed.

In case of non-blocking data structures, where there is no critical section, the most popular safety property is linearizability that is defined below. Here we use the notion of history of the implementation as described in the section 2.1.

Linearizability: A history $\mathcal{H}$ is *linearizable* if $\exists \mathcal{H}_e{=}ext(\mathcal{H})$ and a consistent sequential history $\mathcal{S}$ s.t. (a) $complete(\mathcal{H}_e) \equiv \mathcal{S}$ and (b) $op_1 \underset{\mathcal{H}_e}{\longrightarrow} op_2 \implies op_1 \underset{\mathcal{S}}{\rightarrow} op_2$. We call an implementation $\mathcal{I}_{\mathcal{O}}$ *linearizable* if every execution history of $\mathcal{I}_{\mathcal{O}}$ is linearizable.

The most common approach to prove linearizability is: (a) define *linearization point* of each operation *op* as the execution-point of a step, called *linearization*, which should be between the invocation and response point of *op* then (b) in an arbitrary history $\mathcal{H}$ append appropriate response (in any arbitrary order) of all the operations which have performed their linearization to obtain $ext(\mathcal{H})$, then (c) drop the invocation steps without a matching response to obtain $complete(ext(\mathcal{H}))$, and (d) construct a sequential history $\mathcal{S}$ by arranging the invocation-response pair of operations according to their linearization points. It is easy to argue that $complete(ext(\mathcal{H})) \equiv \mathcal{S}$. And, finally, show that the constructed sequential history $\mathcal{S}$ is consistent.

Another weaker consistency condition is *sequential consistency*. A sequentially consistent concurrent data structure guarantees that the threads executing different operations provided by the data structure will see the effect of the data structure in their respective program order. In other words, the individual thread subhistories $\mathcal{H}|_p$ are equivalent to some consistent sequential histories. It is weaker than the linearizability in the sense that in a linearizable implementation a user looking from outside the threads gets the illusion of operations running in their program order. An even weaker consistency condition is *quiescent consistency* which ensures that two operations separated by a period of quiescence take effect in their real time order but concurrent operations i.e. those whose execution interval overlap can take any order. It can also be seen

as proving that the subhistory of individual thread histories obtained by purging those operations that are concurrent with any other operation in the execution history, are equivalent to some sequential histories. The sequential consistency and quiescent consistency are less often used in proving consistency of concurrent data structures. The former is often useful in describing the correctness of low level concurrent systems such as hardware memory interfaces and the latter is used to provide even weaker constraints in object behaviors, mostly in order to obtain a higher computational performance.

The safety properties described in terms of consistency of execution histories are equally applicable to blocking data structures. In fact, it is desirable to prove that a blocking data structure that satisfies mutual execution property also satisfies linearizability.

The liveness property of non-blocking data structures is expressed in terms of the progress guarantee that they provide. We already described the classes of non-blocking data structures in the last chapter. To prove wait-freedom in terms of an execution α of an implementation $\mathcal{I}_{\mathcal{O}}$, we show that for no operation an infinite number of steps exist in any α. To prove lock-freedom, we show that there exists at least one operation such that it takes only finite number of steps in an arbitrary execution α of an implementation $\mathcal{I}_{\mathcal{O}}$. Finally, to prove obstruction-freedom, we purge the operation steps taken due to obstructions by concurrent operations in any α, and show that for no operation there exist infinite number of steps in $\mathcal{I}_{\mathcal{O}}$.

At times, in an implementation of $\mathcal{A}$, some operations can be wait-free while others are still blocking. Usually, such an implementation can be seen where CONTAINS operations are wait-free under the assumption that there are only finite number of elements in the key universe and modification operations ADD and REMOVE are blocking, for example in the lock-based linked list of Heller et al. [48]. In that case, we prove the the properties of different operations differently with respect to their respective liveness properties.

2.2.2 Complexity

As there is no assumption on the relative speeds of the the threads in an asynchronous shared memory system, the time complexity of an algorithm, which involves synchronization, is difficult to be analyzed. For concurrent algorithms we count the total number of steps taken by all the operations in an execution. It is referred as the *step complexity* of the execution. In a step, to a thread, at most one atomic access to any shared-variable is allowed. However, depending on the architecture of the machine and the memory hierarchy, access to a shared-variable that is cached in a memory close to the processor is orders of

magnitude faster than the access to a shared-variable that is not cached. Therefore it becomes imperative to count only those steps in which an access to a remote shared-variable is performed. This is called *Remote Memory Access* measure. A remote memory access may refer to an attempt by a thread to access a shared-variable residing at either a central shared memory location or in the local memory of a core in which the thread is not running. In both the cases the memory access attempt goes across the memory bus. Depending on the architecture there are two possible remote memory access complexity models

1. Coherent Caching (CC) model - An access to a shared-variable not in the cache memory of the core running the thread is called a remote access.
2. Distributed Shared Memory (DSM) model - An access to a shared-variable in the cache memory of a core that the thread is not running is called a remote access.

Often we need to derive the bounds of operations provided by a concurrent data structure. For this purpose amortized analysis is the most popular method, specifically in the contexts where the operations are not run in isolation. In concurrent data structures the amortized analysis can be used to give the bounds of complexity of operations. For a blocking concurrent data structure, it is impossible to give any upper bound of an operation which follows from the following result due to Alur and Taubenfield [5], we mention it here without its proof.

Theorem 1. *There is no two (or more) thread mutual exclusion algorithm, with an upper bound on the number of times a winning thread may need to access the shared memory in order to enter its critical section in presence of contention.*

Nevertheless, for a non-blocking concurrent data structure, an amortized analysis of its upper bound can be presented in terms of the size of the data structure and the *measure of contention*. In principle the measure of contention is the number of concurrent threads in the recent history of a thread during an interval. Starting with the size of the data structure at the invocation of the operation, the change in size of the data structure during the execution interval of an operation can be accounted to a measure of contention. Moreover the number of extra steps that a thread incurs on account of helping other threads during a given operation can be measured using a measure of contention and hence on amortization we could count total number of steps taken by all the threads performing the operations in a finite execution. Some of the often used measures of contention *during an operation* are as following.

Let op be an operation with t_i and t_r as its invocation and response points respectively,

- *Interval Contention* [3] - Total number of operations whose execution interval overlaps the interval $[t_i, t_r]$.
- *Point Contention* [8] - Maximum number of operations being executed concurrently at any point in the interval $[t_i, t_r]$.
- *Overlapping-Interval Contention* [74] - Maximum interval contention of any operation whose execution interval overlaps the interval $[t_i, t_r]$.

Some authors name the interval contention as *cumulative contention* and point contention as *concurrent contention* [52]. Point contention is a tighter measure of contention compared to interval contention. As explained before the number of extra steps that a thread needs to take on account of the helping mechanism in the design of a concurrent data structure should always be optimized. However, in some cases a thread may end up taking extra read steps due to extended traversal path in a linked concurrent data structure which may arise because of conservative helping. Overlapping interval contention is used in these cases in which a thread has to incur extra steps during an operation which can be accounted to the operations whose execution interval do not overlap with that of itself. For example, in a double linked-list if the insert operations do not help a concurrent insert then there can be such a situation (example taken directly from [74]). Consider that out of two links connecting two nodes in a double linked-list, a link gets updated by an insert and the other is pending while the thread performing insert gets delayed. In the meantime many insert operations succeed to insert multiple nodes between the node whose insert is pending and the node that has been connected by one of its link. Now if after the successful insert operations return, a predecessor query travels extra steps from one direction to the other, these extra steps can not be accounted to an operation whose execution interval overlaps the predecessor query. In such cases overlapping interval contention is used.

Recently, Gibson et al. [39] showed that the amortized number of steps per operation are asymptotically equivalent. This result is important in the sense that we can perhaps see no theoretical incentive to optimize a lock-free data structure algorithm to obtain the amortized complexity in terms of point contention instead of interval contention. Practically, in an implementation the CONTAINS operations are made not to help concurrent ADD or REMOVE operations and that way the amortized complexity is obtained in terms of interval contention. However, it has been observed that minimizing the number of steps taken in helping is practically a much better design choice as we shall see in the chapter 3.

Part II

LOCK-FREE 1-DIMENSIONAL POINT SEARCH

3

HELP-OPTIMAL AND LANGUAGE-PORTABLE LOCK-FREE CONCURRENT DATA STRUCTURES

Chapter Abstract

Helping is a widely used technique to guarantee lock-freedom in many concurrent data structures. An optimized helping strategy improves the overall performance of a lock-free algorithm. In this chapter, we propose help-optimality, which essentially implies that no operation step is accounted for exclusive helping in the lock-free synchronization of concurrent operations. To describe the concept, we revisit the designs of a lock-free linked-list and a lock-free binary search tree and present improved algorithms. Our algorithms employ `CAS` *primitives and are linearizable.*

We design the algorithms without using any language/platform specific mechanism. Specifically, we use neither bit-stealing from a pointer nor runtime type introspection of objects. Thus, our algorithms are language-portable. Further, to optimize the amortized number of steps per operation, if a `CAS` *execution to modify a shared pointer fails, we obtain a fresh set of thread-local variables without restarting an operation from scratch.*

We use several micro-benchmarks in both C/C++ and Java to validate the efficiency of our algorithms against existing state-of-the-art. The experiments show that the algorithms are scalable. Our implementations perform on a par with highly optimized ones and in many cases yield 10%-50% higher throughput.

3.1 Introduction

3.1.1 Overview

The literature on lock-free data structures has grown sufficiently over the last decade [46, 67, 35, 33, 53, 71, 32, 24, 78]. Typically, *practical* lock-free designs use single-word atomic compare-and-swap synchronization primitives (henceforth referred to as `CAS`) to modify shared variables. Thus, to implement a lock-free version of a dynamic pointer-based data structure, in which (multiple) mutable links (pointers) are shared among threads in a concurrent set-up, either by design or due to necessity, one or more `CAS` executions are performed to complete a modify (add or remove) operation. For example, in the lock-free linked-list of [46], two successful `CAS` executions are required to complete a remove operation, whereas in [35] three such executions are required for the same operation. Considering the lock-free external binary search trees (BSTs), three successful `CAS` executions are necessary to remove a node in [71], whereas in [33] and [32], four such executions are required for the same purpose. Furthermore, in [33] and [32], two successful `CAS` executions are required to add a node. Naturally, concurrent operations which modify overlapping sets of links, face each other at a stage where they would have partially completed and would still need to perform one or more `CAS` to complete. Herein, we call this situation *concurrent obstruction*.

For operations on a concurrent data structure, linearizability [50] is the most commonly used consistency framework. Intuitively, a concurrent data structure is linearizable if every execution provides time-points, called *linearization points*, between the invocation and the response of each operation, where it seems to take effect instantaneously. Thus, using a sequence of seemingly instantaneous operations, described by the *real-time order* of the linearization points, we perceive the concurrent operations displaying their sequential behaviour.

In a lock-free algorithm, often a `CAS` execution step is taken as the linearization point of an operation performing multiple `CAS`. Such a step may not necessarily be the last one. Most commonly in a remove operation, on the success of the `CAS` representing the linearization point, the target node is considered *logically removed*, [46, 67, 35, 33, 53]. This results in each traversal passing through a logically removed node and hence extra read steps get counted in step complexity of operations.

A well-known mechanism to deal with such situations is *helping*. Helping essentially implies that if multiple operations face concurrent obstruction or need to perform extra read while traversing over a transient deformation in form

of a logically removed node, based on a fixed protocol, the pending steps of one of the operations are completed by the concurrent operations, before furthering their own course of steps. This strategy ensures lock-freedom because a non-faulty thread definitely completes its operation in finite number of steps.

In the prevalent research on lock-free data structure design, the helping mechanism now holds a center stage. In the lock-free linked-lists of [46, 35], every concurrent operation offers helping to a remove operation which successfully performs the CAS to logically remove the target node and is yet to execute one more CAS. Barnes [11] proposed a helping mechanism called *cooperative technique*. The cooperative technique applied to a data structure requires a modify operation to atomically write the description of planned steps in the node whose links it targets to modify and thereby a concurrent obstructed operation ensures completion of those steps in case the original operation gets delayed. This method is applied in the BST of [33, 32], where even add operations require helping.

In the lock-free BST of Natarajan et al. [71], the links are used much in the same way as in the linked-lists of [46, 35] to modulate helping, and unlike [33, 32], the add operations there do not require help. Broadly, their design provides better progress conditions for concurrent operations, which they showed experimentally. However, we notice that they put the linearization point of a remove operation at the very last CAS execution, which necessitates a concurrent remove operation to help a pending remove operation of the same query key, even though it does not change its return that is false. Clearly, the number of helping steps are not necessarily minimized.

A common suggestion found in the papers on lock-free data structures is that one should avoid helping during traversal by an otherwise unobstructed operation, which if the obstructing operation is not delayed, predominantly goes to wastage. The works analysing experimental performance of concurrent data structures [18, 42, 29] further emphasize on the same. Gibson et al. [39] showed that the amortized number of steps per operation are asymptotically equivalent irrespective of avoiding help by read operations. However, a design optimization to minimize the number of steps incurred by modify operations in helping at a concurrent obstruction is largely un-attempted.

Another noticeable characteristic of existing lock-free algorithms is that their descriptions are very close to the programming language of the sample implementations used by the authors to validate their claim of efficiency. For example, in the linked-lists of [46, 35] and the BST of [71], the design is described in terms of using unused bits from a pointer which points to a memory-word aligned at a fixed boundary. This technique is popularly known as *bit-stealing* in programming parlance. The correctness proof thereof is inherently con-

nected to bit-stealing. In Java toolkit [73], `AtomicMarkableReference` and `AtomicStampedReference` classes are used to simulate bit-stealing, but are not too popular from the performance point of view. The lock-free external BST designs of [33, 32] use polymorphism, class inheritance and type introspection of objects at runtime (also known as real-time-type-information or RTTI), to describe their algorithm. The correctness proofs in these papers are presented accordingly.

In the lock-free skip-list implementation in Java [62], Doug Lea uses extra *splice nodes* to simulate the pointers masked with stolen bits. Such a node is identified with a specific assignment of one of its fields, for example, the value field of a *marker* node points to itself in [62]. A marker node stores the original pointer in its next field enabling unmasking of the pointer off any stolen bit. Lea remarks that in spite of some temporary extra nodes, this technique could still be faster for a traversal with quick garbage collection of removed nodes and is worth avoiding the overhead of extra type testing.

Usually, the lock-free implementations in C/C++, for example in [18] or [29], use their own memory allocation and garbage collection strategies to improve performance. Obviously, these implementation environments of C/C++ very much resemble one in Java and yet they entail each traversal step to unmask a pointer off a possible stolen bit. This underlines a motivation to present the lock-free algorithms that utilize temporary splice nodes and thereby achieving *language portability*. However, the efficiency of such an implementation in C/C++ remains still unexplored for the research community.

In literature, the efficiency of a lock-free algorithm is also presented in terms of the amortized step complexity per operation [35, 32, 24]. Often in a lock-free data structure, when a `CAS` execution in a modify operation returns false, the local variables in the thread become unusable for a reattempt. Hence, the thread needs to restart the operation from a *clean location* to get a fresh set of local variables. Usually, the first sentinel node where an operation starts from (head of a linked-list, root of a BST), is always clean. However, there can be as many as c restarts per operation if c concurrent threads access the data structure. To get a pointer to backtrack to a *local* clean location and thus restart the operation from there, improves the amortized number of steps per operation (counting both read and write). It can be interesting to use a splice node to store a pointer to a node in a local clean location and thus *locally restart* a modify operation.

The contributions of this work are the following:

1. We introduce the concept of *help-optimality* which essentially revisits the lock-free algorithms to optimize the number of `CAS` steps in helping at concurrent obstructions.

2. We describe help-optimal lock-free designs of a linked-list and a BST to implement Set abstract data types (ADT) which export linearizable ADD, REMOVE, and CONTAINS operations. CONTAINS are wait-free in the linked-list for a finite key space.

3. The presented algorithms do not use language specific constructs like bit-stealing or type introspection of objects at runtime and hence are *language-portable* for a programmer.

4. We also show that on a CAS failure at a conflict, the modify operations in our algorithms restart locally to optimize the amortized step complexity per operation.

5. We implement the algorithms in both C++ and Java. Our implementations perform on a par with highly optimized implementations and outperform them in many cases.

Further in this chapter, first, we present a simple lock-free BST algorithm as a motivation for a help-optimal design (section 3.2). Thereafter, we present efficient lock-free algorithms of a linked-list (section 3.3) and a BST (section 3.4), to describe the concept of help-optimality as used in practice. Having described it algorithmically, we specify help-optimality more formally (section 3.5). Finally, we discuss the experimental performance of the presented algorithms (section 3.6).

3.1.2 Related Work

The first CAS-based lock-free linked-list was presented by Valois [84]. He suggested to augment every node with an auxiliary node to manage synchronization. Heller et al. [48] were perhaps the first to suggest that the CONTAINS operations in a concurrent linked-list must progress in a wait-free manner for a finite key space. They presented lock-based linked-list, called lazy list, to show that it favours performance. They also recommended that the CONTAINS operations in Michael's lock-free linked-list [67] should not be involved in helping. Subsequently, to the best of our knowledge, no concurrent data structure was designed in which CONTAINS operations are obstructed; interestingly, some researchers called it *conservative helping*, for example in [33]. In the lock-free internal BSTs presented by Howley et al. [53], Chatterjee et al. [24] and Ramachandran et al. [78] CONTAINS operations complete without helping any concurrent operation.

3.2 Help-optimality: Motivation

Let us consider a very simple lock-free design of an external BST to implement a Set ADT that exports ADD, REMOVE, and CONTAINS operations as given in the Algorithm 3.1.

In this data structure, a node has two pointer fields lt and rt in addition to a key field k, see line 1. Without ambiguity, we shall use *k* to denote a node with key k. The pointer fields lt and rt connect a node to its left and right children respectively, which are null in a leaf (also called external) node. In this BST, the external nodes are *data-nodes* and the internal nodes are *routing-nodes*. There is a *symmetric order* of node-arrangement - the nodes in the *left subtree* of a routing-node *k* have keys less than *k*, whereas in its *right subtree* the nodes have keys at least *k*. We denote the parent of a node *k* by $p(k)$ and there is a unique node called *root* s.t. $p(root) = null$. Each parent is connected to its children via links (we indicate the link emanating from *k* and incoming to *l* by $k \rightsquigarrow l$; we use the terms pointer and link interchangeably). The other child of $p(k)$, i.e. sibling of *k*, is denoted by $s(k)$.

Pseudo-code convention: N·ref represents the reference to a variable N. Thus, f(N·ref) indicates passing N by reference to a method f. If x is a member of a class C then pc.x returns field x of an instance of C pointed by pc. dir L and dir R represent the left and right directions. CAS(A·ref, exp, new) compares A with exp and updates to new in one atomic step if A = exp and returns true; else it returns false without any update at A.

We initialize the BST with a subtree consisting of an internal node root with key ∞_1 and two children with key ∞_0 and ∞_1, where $\infty_1 > \infty_0 > |k| \ \forall$ key k, as its left- and right- child respectively, see Figure 3.1 and line 2. The method Search, line 12 to 13, is used for traversal by a data structure operation. Search takes variables *par*, *cur* and *k* as input which are two node-pointers and a query key, respectively. At the invocation of Search, *cur* points to the child of the node pointed by *par* in the direction of the subtree which can contain *k*. At the termination of Search, *cur* points to a leaf-node which is identified by the lt field being null.

To perform REMOVE(k), line 20 to 26, we use Search to arrive at a leaf-node pointed by ℓ. If *k* matches the key at ℓ, we use a CAS to replace ℓ with a special node with the same key, but with its rt field pointing to itself, see line 24. We call such special nodes Dead. See method IsDead at line 10 which is used to identify a Dead node. If the CAS succeeds, REMOVE returns true; if *k* was not found or ℓ was already Dead, REMOVE returns false. For ADD(k), line 27 to 36, arriving at ℓ using Search, we use a CAS to replace ℓ with (i) a new leaf-node with key *k*, if ℓ was Dead and (ii) a new internal node created using

Algorithm 3.1. A Simple Help-optimal Language-portable Lock-free Binary Search Tree

```
1  struct Node {K k; Node* lt, rt;};

2  root := Node(∞1, Node(∞0)·ref, Node(∞1)·ref);

3  Dir(Node* par, K k) {return k < par·k ? L : R};

   ChCAS(Node* par, Node* exp, Node* new, dir cD)
4  | if (cD == L) and par·lt == exp then
5  | | return CAS(par·lt·ref, exp, new);
6  | else if (cD == R) and par·rt == exp then
7  | | return CAS(par·rt·ref, exp, new);
8  | else return false;

9  GetDead(K k) {n := Node(k); n·rt := n; return n;}

10 IsDead(Node* leaf) {return leaf·rt == leaf;}

   Child(Node* par, dir cD)
11 | return cD == L ? par·lt : par·rt;

   Search(Node* par, Node* cur, K k)
12 | while cur·lt ≠ null do
13 | | par := cur; cur := Child(par, Dir(par, k));

   NewNode(Node* a, Node* b, K pKey)
14 | left := (a·k < b·k ? a : b);
15 | right := (a·k < b·k ? a : b);
16 | return Node(pKey, left, right, null);

   CONTAINS(K k)
17 | p := root·ref; ℓ := root·lt;
18 | Search(p·ref, ℓ·ref, k); cD := Dir(p, k);
19 | return ℓ·k == k and !IsDead(ℓ);
```

Algorithm 3.1. A Simple Help-optimal Language-portable Lock-free Binary Search Tree

```
    REMOVE(K k)
20  | p := root·ref; ℓ := root·lt;
21  | while true do
22  |   Search(p·ref, ℓ·ref, k); cD := Dir(p, k);
23  |   if ℓ·k ≠ k or IsDead(ℓ) then return false;
24  |   if ChCAS(p, ℓ, GetDead(k), cD) then
25  |     return true;
26  |   ℓ := Child(p, Dir(p, k));
```

```
    ADD(K k)
27  | nd := Node(k)·ref;
28  | p := root·ref; ℓ := root·lt;
29  | while true do
30  |   Search(p·ref, ℓ·ref, k); cD := Dir(p, k);
31  |   if !IsDead(ℓ) then
32  |     if ℓ·k == k then return false;
33  |     n := NewNode(nd, ℓ, max{k, ℓ·k})·ref;
34  |     if ChCAS(p, ℓ, n, cD) then return true;
35  |   else if ChCAS(p, ℓ, nd, cD) then return true;
36  |   ℓ := Child(p, Dir(p, k));
```

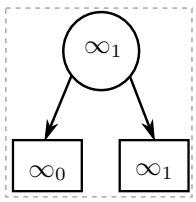


Fig. 3.1: Sentinel Nodes: Simple Lock-free BST

NewNod, line 14 to 16, if ℓ was not Dead and k does not match at ℓ. If the CAS succeeds at line 34 or at line 35, ADD returns true; if ℓ was not Dead and contained k, it returns false. A CONTAINS(k), line 17 to 19, returns true if k is found at a leaf-node which is not Dead, else it returns false.

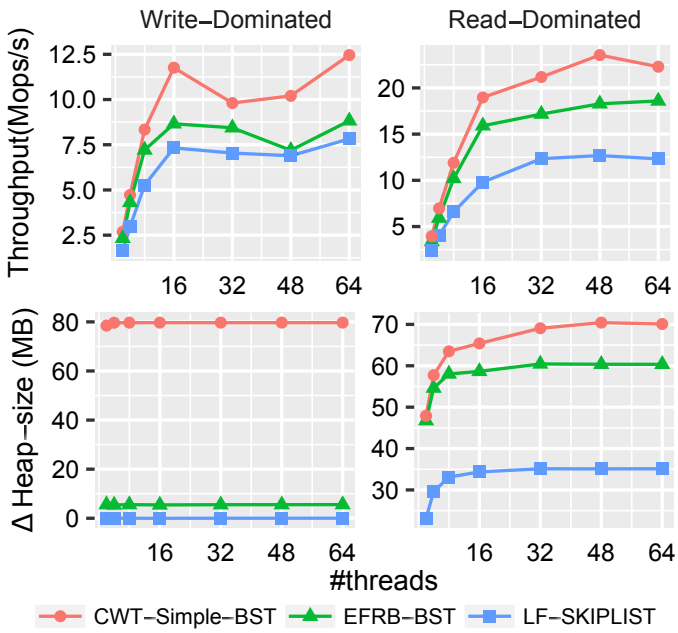


Fig. 3.2: Performance graph: Lock-Free Basic BST

The main idea of this algorithm is to discard the requirement of helping by *not* cleaning out a node in a REMOVE operation, which otherwise uses multiple CAS executions. Thus, a single successful CAS is required by both ADD and REMOVE operations, much like a lock-free stack. We skip the proofs of correctness and lock-freedom of this algorithm, which are straightforward. An interested reader may take them as a simple exercise. Please note that we have not used any language specific construct to describe this algorithm.

We implemented the Algorithm 3.1 in Java and compared it against the (author provided) implementation of lock-free BST of [33] and the lock-free skip-list of Java library [62]. The set-up and methodology of the experiments are described in section 3.6. The throughput and memory usage by the algorithms to implement a Set formed by at most 2^{20} distinct keys are plotted in the Figure 3.2.

We see that this simple lock-free BST handsomely outperforms state-of-the-art implementations of a skip-list and a BST. However, on account of memory footprint, it performs poorly. We get enough motivation to design lock-free data structures which optimally reduces the number of CAS executions by each ADT operation aided with optimal memory footprints.

3.3 Help-optimal Lock-free Linked-list

3.3.1 Design

We implement an ordered linked-list based Set ADT which exports ADD, REMOVE and CONTAINS operations. The pseudo-code is given in the Algorithm 3.2. A node has two pointer fields nxt and bck in addition to the key field k, see line 1. As before, we use k to denote a node with key k. The field nxt points to the successor of k, denoted by $s(k)$. We describe the use of bck later; it is null in a regular node. The predecessor of k is denoted by $p(k)$. Initially, the linked-list consists of four sentinel nodes tailNxt, tail, headNxt and head with keys ∞_1, ∞_0, $-\infty_0$ and $-\infty_1$, respectively, where $\infty_1 > \infty_0 > |k| \; \forall$ key k. See line 2 to 5 and the Figure 3.3.

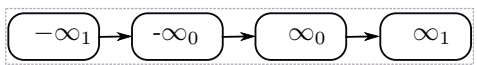


Fig. 3.3: Sentinel Nodes: Lock-free linked-list

We aim to reduce the number of CAS steps incurred in helping not only during traversal, which is simple, but also in the concurrent obstruction. In the Algorithm 3.1 we observed that obstruction can be fully avoided in an external BST if REMOVE operations do not try to clean out the removed nodes. However, that strategy led to undesirably large memory footprint. So, the question we ask - can we overcome the drawbacks? Observing carefully, in a linked-list we can leverage the linear structure to connect the predecessor of the leftmost node to the successor of the rightmost node of a contiguous chunk of removed nodes by a single CAS and thus solve the issue. We describe it below.

At the basic level, ADD(k) in a lock-free linked list comprises - finding $p(k)$ and $s(k)$ s.t. $p(k)$.k$<k<s(k)$.k, allocating the node k s.t. k.nxt$=s(k)$ and using a single CAS execution to swing the $p(k)$.nxt from $s(k)$ to k. We have seen it in [46, 35]. Similarly, REMOVE(k) comprises - finding nodes $p(k)$ and k, logically removing k using a single CAS and then swing the $p(k)$.nxt from k to $s(k)$ using a CAS.

Algorithm 3.2. Help-optimal lock-free linked-list

```
1  struct Node {K k; Node* nxt, bck;};

2  tailNxt := Node(∞1, null, null);
3  tail := Node(∞0, tailNxt·ref, null);
4  headNxt := Node(−∞0, tail·ref, null);
5  head := Node(−∞1, headNext·ref, null);

   Search(Node* pre, Node* nex, Node* cur, Node* suc, K k)
6    while cur·k < k do
7      if IsSp(suc) then cur := suc·nxt;
8      else pre := cur; nex := suc; cur := suc;
9      suc := cur·nxt;

   CONTAINS(K k)
10   c := headNext·nxt;
11   while c·k < k do c := cur·nxt;
12   return c·k == k and !IsSp(c·nxt);

   ADD(K k)
13   p := head·ref; n := headNxt·ref;
14   c := headNxt·ref; s := headNxt·nxt;
15   while true do
16     Search(p·ref, n·ref, c·ref, s·ref, k);
17     if IsSp(s) then
18       while IsSp(s) do {c := s·nxt; s := c·nxt};
19     else if c·k == k then return false;
20     if CAS(p·nxt·ref, n, Node(k, c, null)) then
21       return true;
22     BckTrck(p·ref, n·ref); c := p; s := n;

23 BckTrck(Node* pre, Node* nex)
24   nex := pre·nxt;
25   while IsSp(nex) do
26     pre := nex.bck; nex := pre·nxt;

27 IsSp(Node* c) {return c·k == −∞2;}
```

However, our aimed implementation as described before will require additional tricks over this basic idea. Firstly, in order to make the algorithm language-portable, we find a way of using *splice nodes* as Lea [62], instead of bit-stealing like [46, 35]. For that, when logically removing k, we add a splice node between k and $s(k)$. We fix the key of a splice node as $-\infty_2$, where $\infty_2 > \infty_1$, by which it can be identified, see line 35 and the method `IsSp` at line 27.

Algorithm 3.2. Help-optimal lock-free linked-list

```
    REMOVE(K k)
28  p := head·ref; n := headNxt·ref;
29  c := headNxt·ref; s := headNxt·nxt;
30  r := null; spNd := null; mode := INIT;
31  while true do
32      Search(p·ref, n·ref, c·ref, s·ref, k);
33      if mode == INIT then
34          if c·k ≠ k or IsSp(s) then return false;
35          spNd := Node(−∞2, s, p)·ref;
36          while true do
37              if CAS(c·nxt·ref, s, spNd) then
38                  if CAS(p·nxt·ref, n, s) then return true;
39                  r := s; mode := CLEAN; break;
40              s := c·nxt; if IsSp(s) then return false;
41              spNd·nxt := s;
42      else if s ≠ spNd or CAS(p·nxt·ref, n, r) then
43          return true;
44      BckTrck(p·ref, n·ref); c := p; s := n;
```

Secondly, to avoid eager helping during traversal by a modify operation and yet be able to clean out the logically removed nodes (along with the splice nodes succeeding them), we use two trailing node-pointers during traversal. This trick is similar to [71], there used in BST. We use them to store the address of the last node, which was *not* logically removed, and its successor. Thus, at the termination of a traversal, we have reference to the predecessor, say $p(k)$, of the leftmost node of a possible contiguous chunk of logically removed nodes. Hence, when we swing the pointer $p(k)$.nxt, using a `CAS`, to connect either to a new node k for ADD or to $s(k)$ for REMOVE, zero or more logically removed

nodes are cleaned out.

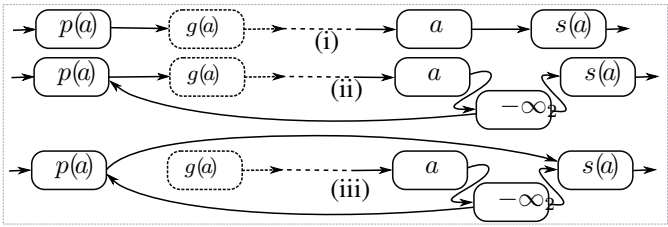


Fig. 3.4: Steps of REMOVE in the linked-list.

And thirdly, to backtrack to a clean zone on CAS failures, we use the idea of back-pointers as applied in [35]. However, our approach differs from them. When allocating a splice node, we save the address of $p(k)$ in its bck field, which is always null for a regular node. Essentially, our approach is novel in the following ways - (a) we do not use an extra CAS to fix (*flagging*) the pointer $p(k)$.nxt. Given the use of trailing pointers, we do not often travel a long chain of back pointers. And (b) we do not set back-pointer of a regular node and thus, save an extra atomic write of a shared pointer. Indeed, a splice node in our algorithm splices two node paths.

The basic steps of REMOVE(k), are shown in the Figure 3.4. The node $n(k)$ denotes the first node of a possible contiguous chunk of logically removed nodes before and adjacent to k. In case there is no such chunk before k, $n(k)$ coincides with it.

We perform traversal for a modify operation using the method Search, line 6 to 9. We advance the variables *pre* and *nex* only if *suc* is not a splice node, that is when *cur* is not a logically removed node. Otherwise, we advance *cur* to the node saved at the *nxt* of the splice node *suc*. In REMOVE and ADD, at the first call of Search, the variable *pre* points to head, *nex*, *cur* point to headNxt and *suc* points to the successor of headNext, see lines 13 and 14. Thus, at the termination of a traversal, when *cur* points to a node with key *not greater than k*, *pre* points to the predecessor of the first node of a possible contiguous chunk of logically removed nodes and *nex* points to the first node of such a chunk.

To perform REMOVE(k), line 30 to 44, at the termination of a traversal, we check the key at the node pointed by c, and if *k* does not match at it or the node pointed by s is found splice (indicating node pointed by c is already logically removed), we return false, line 34. Otherwise, we perform a CAS to add a splice node between c and s to logically remove c, line 37. The steps taken up to this point are identified by a variable mode with value INIT. After this step,

mode changes to CLEAN and we attempt to swing the p.nxt from n to s using a CAS at line 38. If the CAS fails, we save s as r, and perform a BckTrck at line 44 to find a fresh pair of p and n.

In the method BckTrck, line 24 to 26, we keep on traversing back, following the bck of splice nodes, until we find the first node which is not logically removed. If the call of BckTrck was due to a CAS failure caused by an ADD of a new node, added between *pre* and *nex*, it is guaranteed that the chunk of contiguous logically removed nodes must have been cleaned out. We explain it in the next paragraph.

The operation ADD(k), line 13 to 22, performs a similar traversal. At the termination of the traversal, we check if the node pointed by c is logically removed by checking whether s points to a splice node, line 17. If the node at c is not logically removed and contains the query key k, we return false; else, we find the first node succeeding it which is still not logically removed, line 18, and attempt a CAS to add the new node between p and c to return true. On a CAS failure, we perform BckTrck as explained before and reattempt the previous steps. Thus, on a successful ADD it cleans out a complete chunk of contiguous logically removed nodes.

Note that, a modify operation in the Algorithm 3.2 differs from one in [46, 35], in the sense that on a CAS failure at $p(k)$.nxt, we do not perform any help before reattempting. Instead of that, we selfishly attempt the CAS from a clean location. Thus, the operations are essentially *selfish* in our algorithm.

A CONTAINS operation, line 10 to 12, traverses in a wait-free manner and returns true only if the node at which it terminates, the one pointed by c, is not logically removed and contains the query key, else it returns false.

3.3.2 Correctness and Lock-freedom

It is easy to observe that the field k of a node is never modified after initialization. Scanning through the pseudo-code, we can observe that once a splice node is added at the nxt of a node, no CAS is performed at it. Further, unless the nxt of a node k is splice, it is not removed from the list. Thus, we can show that a node $p(k)$ is present in the list, when we connect a new node k or successor $s(k)$ of a removed node k to it. Additionally, we can observe that a traversal terminates with c pointing to a node which has a key greater than or equal to k in all the operations, which in turn shows that we maintain the order of node arrangement in an ADD or a REMOVE operation. At the initialization, the sentinel nodes form a valid ordered list. Hence, using induction we can prove that the ADT operations maintain a valid ordered list.

The linearization point for an unsuccessful ADD operation is at line 9 dur-

ing a call of Search. Similarly for a successful CONTAINS operation it is at line 11, when we read c·nxt for the first time. For a successful ADD or a REMOVE operation, the linearization point lies at the first successful CAS execution to add a new or a splice node. For an unsuccessful CONTAINS, the linearization point is (a) just after that of the concurrent REMOVE operation which (logically) removed k, if k existed in the list at the invocation point of CONTAINS and (b) at the invocation point itself, if k was not present in the list at that point. The linearization point of an unsuccessful REMOVE is determined similar to an unsuccessful CONTAINS operation.

We can observe that the CAS to add a splice node is reattempted only if a new node is added at the nxt of k. Before every reattempt of a CAS to swing the nxt pointer of $p(k)$, in both ADD and REMOVE, we perform a BckTrck and a Search which guarantees that we have a fresh set of variables for references of $p(k)$ and $n(k)$. Hence, it is guaranteed that a modify operation can not take an infinite number of steps without a modification in the data structure. It proves the lock-freedom of the ADD and REMOVE operation. It is easy to observe that a non-faulty CONTAINS always finishes in a finite number of steps if the key space is finite and thus is wait-free.

3.3.3 Amortized Step Complexity

We can observe that the splice nodes are never adjacent. Similar to [42], we do not perform help in a CONTAINS operation. Additionally, in ADD and REMOVE as well, no step is taken for helping during traversal. On a CAS failure to add a splice node, we do not perform any traversal. On a CAS failure to add a new node or to clean out a chunk of logically removed nodes, we perform backtrack and do not start from the head. Following the same method as [35], we can show that the amortized number of steps per operation is $O(n+c_I)$, where c_I is the total number of concurrent operations between invocation and response of o, called *interval contention* [3] and n is the size of the list at the invocation of o. In the light of theorem 1 of [39], it is asymptotically equivalent to $O(n+c_P)$, where c_P is the maximum number of concurrent operations at any point in the lifetime of o, called *point contention* [8].

3.4 Help-optimal Lock-free BST

Having described a simple lock-free BST and an improved lock-free linked-list, where we do not spend any CAS execution for helping, we are ready to describe an efficient lock-free BST, in which we introduce the notion of *help-awareness*.

Algorithm 3.3. Help-optimal lock-free binary search tree

```
1  struct Node {K k; Node* lt, rt, bck;};

2  root := Node(∞1); grRoot := Node(∞0);
3  root·lt := Node(∞2)·ref; root·rt := Node(∞1)·ref;
4  grRoot·lt := root·ref; grRoot·rt := Node(∞0)·ref;

   Search(Node* gPar, Node* nex, Node* par, Node* leaf, K k)
5  | while leaf·lt ≠ null do
6  | | if IsSp(leaf) then par := leaf·rt;
7  | | else gPar := par; nex := leaf; par := leaf;
8  | | leaf := Child(par, Dir(par, k));

   GetNxt(Node* leaf)
9  | return IsSp(leaf) ? leaf·rt : leaf;

10 GetKey(Node* leaf) {return GetNxt(leaf)·k;}

   GetDeadBl(Node* gPar, K k)
11 | n := GetDead(k); n.bck := gPar; return n;
12 IsBl(Node* leaf) {return leaf.bck ≠ null;}

   GetSp(Node* gPar, Node* leaf)
13 | if IsDead(leaf) then
14 | | return GetDeadBl(gPar, leaf);
15 | else return Node(−∞3, leaf·lt, leaf, gPar);

   AddSp(Node* par, Node* gPar, dir sD)
16 | while true do
17 | | sib := Child(par, sD);
18 | | if IsBl(sib) then return sib;
19 | | else if ChCAS(par, sib, GetSp(gPar, sib), sD) then return sib;

20 IsSp(Node* leaf) {return leaf·k == −∞3;}

   BckTrck(Node* gPar, Node* nex, K k)
21 | nex := Child(gPar, k);
22 | while IsSp(nex) do
23 | | gPar := nex.bck; nex := Child(gPar, k);
```

3.4.1 Design

The pseudo-code of the design is given in the Algorithm 3.3. The symmetric order of the BST is same as that in section 3.2. We borrow the notations from the Algorithm 3.1 along with the methods `Dir`, `Child`, `ChCAS`, `GetDead`, `IsDead` and `NewNod` as they are described there. We denote the parent of $p(k)$ by $g(k)$.

The main drawback of the lock-free BST of the Algorithm 3.1 was removing a node *k* by replacing it with a `Dead` node and not cleaning the `Dead` node out that caused memory-wastage. Therefore, in the Algorithm 3.3 we make a REMOVE operation clean out the added `Dead` node. Consequently, the ADD operations will have to synchronize with the REMOVE operations which now make structural changes in the BST.

In a sequential set-up, removing a node *k* from an external BST is a one step process of modifying the link $g(k) \rightsquigarrow p(k)$ to connect $s(k)$ to $g(k)$. This process also cleans out the removed node. It essentially removes the node $p(k)$ from the (unordered) linked-list described by the nodes on the path from the root to $s(k)$. Thus, to perform REMOVE(k) with cleaning out *k* in a lock-free BST can be visualized as a two stage process - (a) single `CAS` to logically remove *k* by replacing it with a `Dead` node as in the Algorithm 3.1 and (b) two `CAS` steps to remove $p(k)$ - adding a splice node between $p(k)$ and $s(k)$ to logically remove $p(k)$ and then swinging the pointer $g(k) \rightsquigarrow p(k)$ to connect $s(k)$ to $g(k)$, as in the Algorithm 3.2. Let us call these stages LREMOVE and PREMOVE, respectively. This understanding gives us the fundamental idea of the Algorithm 3.3.

LREMOVE is quite straightforward. Now, to perform PREMOVE efficiently, along the lines of the Algorithm 3.2, we carry two trailing node-pointers during the traversal for a modify operation. Thus, the method `Search` in the Algorithm 3.3, line 5 to 8, becomes a blend of the same method in the previous two algorithms. At the termination of `Search`, the variable *gPar* points to the parent of the root of the sub-tree in which all the nodes are logically removed. To avoid special cases arising in placing the trailing node-pointers in an empty BST, we use a set of sentinel nodes as shown in the line 2 to 4 and the Figure 3.5.

We assign key $-\infty_3$ for a splice node, such that $\infty_3 > \infty_2 > \infty_1 > \infty_0 > |k|$ $\forall$ key k. It ensures that at a splice node a traversal always *goes right*. Hence, we connect $s(k)$ to rt of a splice node. We copy the lt field of $s(k)$ to the splice node that it connects to, which if null, indicates that $s(k)$ is a leaf node. Thus, a traversal may terminate at a splice node. Considering that, we always use the method `GetNxt` to access the actual leaf node, see line 9; and following that the method `GetKey` gives the key at that leaf node, see line 10. Further,

Algorithm 3.3. Help-optimal lock-free binary search tree: ADD

```
    ADD(K k)
24  | g := grRoot·ref; n := root·ref; p := root·ref;
25  | ℓ := root·lt; nd := Node(k)·ref;
26  | while true do
27  |   Search(g·ref, n·ref, p·ref, ℓ·ref, k);
28  |   cD := Dir(p, k); pD := Dir(g, k);
29  |   if !IsDead(ℓ) then
30  |     if GetKey(ℓ) == k then return false;
31  |     nl := NewNode(nd, GetNxt(ℓ), (k+GetKey(ℓ))/2)·ref;
32  |     if IsSp(ℓ) then
33  |       if ChCAS(g, n, nl, pD) then return true;
34  |     else if ChCAS(p, ℓ, nl, cD) then return true;
35  |   else
36  |     if IsBl(ℓ) then
37  |       sib := AddSp(p, g, !cD);
38  |       if !IsDead(sib) then
39  |         nl := NewNode(nd, GetNxt(sib), (k+p·k)/2)·ref;
40  |         if ChCAS(g, n, nl, pD) then return true;
41  |       else if ChCAS(g, n, nd, pD) then return true;
42  |     else if ChCAS(p, ℓ, nd, cD) then return true;
43  |   BckTrck(g·ref, n·ref, k); p := g; ℓ := n;
```

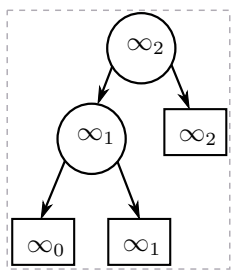


Fig. 3.5: Sentinel Nodes: Lock-free BST

to achieve local restart as in the Algorithm 3.2, we include a bck pointer in the node structure to implement splice nodes that can provide reference to a node in a local clean zone. However, the local restart here is more complex, as discussed below. Consider these cases:

Algorithm 3.3. Help-optimal lock-free binary search tree: REMOVE

```
    REMOVE(K k)
44  g := grRoot·ref; n := root·ref; p := root·ref; ℓ := root·lt;
45  dNdBl := null; sib := null; mode := INIT;
46  while true do
47  | Search(g·ref, n·ref, p·ref, ℓ·ref, k);
48  | cD := Dir(p, k); pD := Dir(g, k);
49  | if mode == INIT then
50  | | if GetKey(ℓ) ≠ k or IsDead(ℓ) then return false ;
51  | | dNd := GetDead(k);
52  | | if !IsSp(ℓ) and p ≠ g then
53  | | | dNdBl := GetDeadBl(g, k);
54  | | | if ChCAS(p, ℓ, dNdBl, cD) then
55  | | | | sib := AddSp(p, g, !cD); mode := CLEAN;
56  | | | | if IsSp(sib) then return true;
57  | | | | else if IsDead(sib) then
58  | | | | | ChCAS(g, n, dNd, pD); return true;
59  | | | | else if ChCAS(g, n, sib, pD) then return true;
60  | | | else if ChCAS(g, n, dNd, pD) then return true;
61  | | else
62  | | | if ℓ == dNdBl and p ≠ g then
63  | | | | if ChCAS(g, n, sib, pD) then return true;
64  | | | else return true;
65  | | BckTrck(g·ref, n·ref, k); p := g; ℓ := n;
```

(A) An ADD operation o just before performing its CAS step gets preempted by the operating system scheduler. Let g be the trailing node-pointer pointing to the last internal node of the traversal path which is not logically removed. Suppose that, by the time o wakes up, the BST changes in a way that both the children of the node pointed by g are replaced by Dead nodes and the node itself cleaned out of the BST. Consequently, o will have *no* link to reach a clean zone except restarting from the root of the BST, which we want to avoid. To tackle this issue, we use the bck pointer of a Dead node, which replaces a

node *k* in a REMOVE operation, to store g. We call a `Dead` node with a non-null bck field a `DeadBl` node.

(B) Two concurrent REMOVE operations o_1 and o_2, at the end of their traversal, target to remove two leaf nodes k_1 and k_2, which are children of the same internal node, say p. Also suppose that o_1 and o_2 have same pair of trailing node-pointers - g and n - in their thread-local memory and thus for both o_1 and o_2 there is access to no link to backtrack above g in the BST. Suppose that LREMOVE stage of both o_1 and o_2 finished without contention, and thus after that both the children of p are `DeadBl`. Therefore, after its PREMOVE, if o_1 successfully connects the `DeadBl` node k_2 to g, o_2 will not get a node-pointer to reach a local clean zone to get a fresh g. It becomes untenable to restart o_2 in such a situation without accessing root, which we want to avoid (it may well be with o_1 symmetrically). To tackle this issue, we let o_2 fall back to the approach of the Algorithm 3.1 and instead of cleaning the `DeadBl` node out it adds a `Dead` node containing key k_2 at g and gets out of the system to ensure progress. Therefore, similar to the Algorithm 3.1, we make an ADD operation replace a `Dead` node with a new leaf node, knowing that no REMOVE operation takes step to clean out such a node.

With basics in place, we are ready to describe the pseudo-code of REMOVE and ADD operations of the Algorithm 3.3; a CONTAINS operation works absolutely same as that in the Algorithm 3.1.

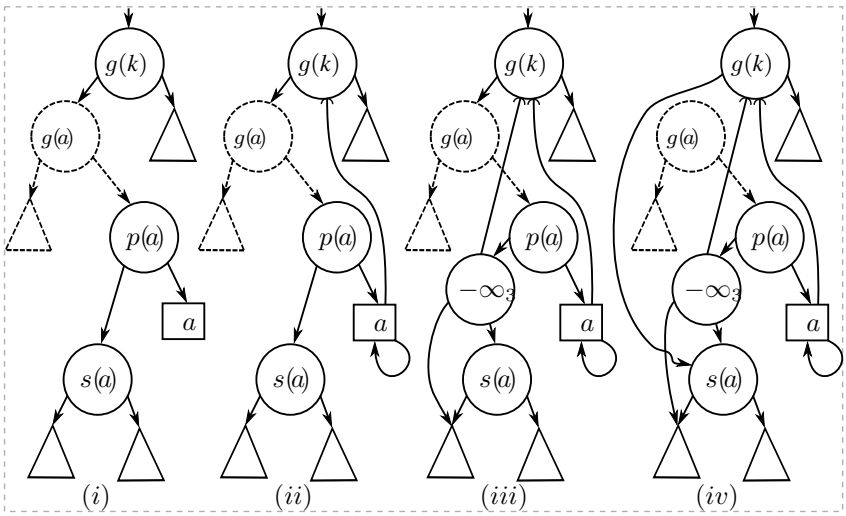


Fig. 3.6: Steps of REMOVE in the BST.

The steps of a REMOVE(k) operation, line 44 to 65, are shown in the Figure 3.6. Let $n(k)$ be the last logically removed internal node in the traversal path obtained by a call of `Search` at line 48, and $g(k)$ be its parent as shown in the Figure 3.6 (i). $n(k)$ coincides with $p(k)$ in case there is no chunk of logically removed nodes above $p(k)$ in the traversal path. Replacing the target node k with a `DeadBl` node containing same key, Figure 3.6 (ii), logically removes k, line 54. After that, we add a splice node between $p(k)$ and $s(k)$ to logically remove $p(k)$ as shown in the Figure 3.6 (iii). Finally, update the link $g(k) \rightsquigarrow n(k)$ to connect $s(k)$ to $g(k)$ as shown in the Figure 3.6 (iv). If all these `CAS` executions are successful, we complete the REMOVE operation with cleaning out the `DeadBl` node.

In the stage LREMOVE itself, if the leaf-node at which traversal terminates, say ℓ, does not contain k or is found `Dead` (note that a `DeadBl` node is also `Dead`), REMOVE(k) returns false, line 50. If ℓ is a splice node, it shows that the actual node to remove is pointed by `GetNxt`(ℓ) which is ℓ.rt. To make a REMOVE operation *selfish*, we do not perform any `CAS` to help the pending REMOVE. However, we do not have a possibility for a local restart to get a fresh $g(k)$ after the completion of LREMOVE, similar to the case (B) above. Therefore, we replace the link $g(k) \rightsquigarrow n(k)$ by a `Dead` node containing k, and return true if the `CAS` succeeds, line 60.

In the stage PREMOVE, the first step is to add a splice node between $p(k)$ and $s(k)$, line 55. We use the method `AddSp`, line 16 to 19, to do that. In `AddSp`, if $s(k)$, denoted by sib, is found splice or `DeadBl`, indicated by non-null bck link, we return it as it is, line 18, because both indicate that the child-link of *par*, referred by sib, is never updated ever after. If sib is `Dead`, we perform a `CAS`, line 19, to replace it with a `DeadBl` node connecting its bck field to *gPar*, created at line 14, so that a concurrent ADD does not replace it directly.

If the method `AddSp` returns a splice node at line 55, it indicates that a concurrent ADD operation is working selfishly to progress (we describe it later) and we can safely allow the remaining steps of REMOVE(k) to be assimilated in the steps of ADD. Considering that, REMOVE(k) returns true, line 56. We call this behaviour of REMOVE(k) its *help-awareness* which is a main component of a *help-optimal implementation*.

On the other hand, if `AddSp` returns a `Dead` or `DeadBl` node, it indicates a scenario of case (B) and we handle it accordingly, see line 58. Finally, if a regular leaf node is returned as sib, we attempt a `CAS` to connect it to $g(k)$ to return true at line 59. If this `CAS` fails, it indicates that $g(k) \rightsquigarrow n(k)$ has changed and therefore we perform a `BckTrck` at line 65 similar to the Algorithm 3.2 and reattempt the `CAS` if required, see line 63. Along the lines of the

Algorithm 3.2, the steps taken to add splice node between $p(k)$ and $s(k)$ are identified by the value of a variable mode set as `INIT` and after that mode is changed to `CLEAN`, line 55.

To add a new node in an external BST, we add a new sub-tree. We use the following *midpoint rule* to determine the key at the root of the new sub-tree.

Theorem 3.1 (Midpoint rule). *Let k be a query key and A be the (partially ordered) set of keys stored in a sub-tree. Let $a_l \leq a \; \forall \; a \in A$ and $a_u \geq a \; \forall \; a \in A$. To add a new node at the root of the sub-tree, assign a key k_p at the root of the new sub-tree such that $k_p = \frac{k+a_u}{2}$ if $k > a_u$ and $k_p = \frac{k+a_l}{2}$ if $k < a_l$.*

The mid-point rule maintains the symmetric order of the BST. Intuitively, rule 3.1 optimizes the average hight of the BST. We do not delve into an analytical discussion of this rule in the present work. In experiments, we observed that this approach improves the average throughput.

An ADD(k), line 25 to 43, performs a traversal using `Search` to reach a leaf node ℓ. If ℓ is neither `Dead` nor `DeadBl`, we find the regular leaf node using `GetNxt`(ℓ). It calls the `NewNod` method to create a new internal node pointed by nl. We apply rule 3.1 at line 31. If ℓ is a regular node, it perform as in the Algorithm 3.1. However, if ℓ is a splice node, it does not take steps to help the pending REMOVE operation and behaves in a selfish manner to directly update $g(k) \rightsquigarrow n(k)$ to nl using a `CAS`. If `CAS` succeeds, it not only ensures success of ADD(k), but also guarantees the completion of some pending REMOVE operations. If `CAS` to connect nl at line 33 or 34 succeeds, we return true.

On the other hand, if ℓ is found `Dead`, ADD(k) behaves along the lines of the Algorithm 3.1, see line 42. And finally, if ℓ is `DeadBl`, to ensure progress, we first fix the sibling of ℓ using the method `AddSp`, line 37, and then add either a new node, line 41, or a new internal node, line 40, at $g(k)$ in a selfish fashion. The call of `AddSp` at line 37 may assimilate the steps of a concurrent pending REMOVE operation, which being help-aware, terminates immediately, as discussed before. Note that, to apply rule 3.1 here, we use p·k instead of `GetKey`(sib) because the latter may not provide the required bound of the set of keys stored in the sub-tree rooted at sib. On a `CAS` failure, we perform a `BckTrck` at line 43 to get a fresh set of thread-local variables and reattempt.

3.4.2 Correctness and Lock-freedom

Proving that the modify operations maintain a valid external BST requires similar approach as that in the Algorithm 3.2. Therefore, without repeating them, we mention that we derive an induction based proof building on the arguments

that the sentinel nodes form a valid BST at the initialization and no modify operation invalidates the symmetric order of the BST.

In this algorithm, the linearization points of a successful ADD, REMOVE and CONTAINS operations and an unsuccessful ADD operation are similar to their counterparts in the Algorithm 3.2. A CONTAINS or a REMOVE operation returns false also in case the node containing query key is found `Dead`, in addition to the cases already discussed in the Algorithm 3.2. The linearization point of such a CONTAINS or REMOVE operation is taken at own invocation point.

Finally, we can prove the lock-freedom of the Algorithm 3.3 using arguments which are parallel to those used in Algorithm 3.2. However, unlike a linked-list, in an external BST even though a CONTAINS is not engaged in any helping activity, it is not wait-free, even for a finite key universe. This is a very interesting aspect of an external BST structure, which entails an entirely different and innovative approach to designing a wait-free algorithm for this data structure.

3.5 Help-optimality: Specification

We consider a *shared memory system* U comprising of a finite set of *threads* Λ and a finite set of *shared variables* V. At time t, the *states* of Λ and V are denoted by Λ_t and V_t, respectively. Let Υ be a lock-free data structure formed by variables $v{\in}V$. Let $\mathcal{O}_\lambda$ be the set of *operations* performed by a $\lambda{\in}\Lambda$ on Υ. A *step* s of an operation $o{\in}\mathcal{O}_\lambda$ comprises local computations in λ and at most a single execution of an atomic-primitive $a{\in}\{\texttt{read}, \texttt{write}, \texttt{CAS}\}$ on a shared variable $v{\in}V$. A state Υ_t of Υ is formed by variables $v{\in}V_t$. On execution of a step s, Υ can change from a state Υ_t to another state $\Upsilon_{t'}$. We denote such a state change by $\Delta\Upsilon_{t,t'}$. Let S_o denote the set of steps to complete an operation $o{\in}\mathcal{O}_\lambda$.

We call $s{\in}S_o$ an *altruistic step* of o, if (a) it is executed to apply a state change $\Delta\Upsilon_{t,t'}$ (b) $\Delta\Upsilon_{t,t'}$ is necessary for completion of a concurrent operation $o'{\in}\mathcal{O}_{\lambda'}$ and (c) $\Delta\Upsilon_{t,t'}$ is *not* necessary for completion of o. We call an operation *o selfish* if *no* step $s{\in}S_o$ is altruistic.

We call $s{\in}S_o$ a *wasted step* of o, if (a) it is executed to apply a state change $\Delta\Upsilon_{t,t'}$ (b) $\Delta\Upsilon_{t,t'}$ is necessary for completion of o (c) $\Delta\Upsilon_{t,t'}{\subseteq}\Delta\Upsilon_{t,t''}$ (c) $\Delta\Upsilon_{t,t''}$ has already been applied by a set of steps $\{s'_1, \ldots, s'_n\}$, where $s'_i{\in}S_{o'_i}$ is a step of a concurrent operation $o'_i{\in}\mathcal{O}_{\lambda'_i}$. We call $o{\in}\mathcal{O}_\lambda$ *help-aware* if it performs *no more than one* wasted step.

A lock-free data structure Υ is called *help-optimal* if every operation $o{\in}\mathcal{O}_\lambda$

for each $\lambda \in \Lambda$ is both selfish and help-aware. In the Algorithms 3.1 to 3.3, we can observe that every operation satisfies the requirements of both selfishness and help-awareness. We skip a rigorous definitional discussion on selfishness, help-awareness, and help-optimality to a future work.

Censor-Hillel et al. [20] defined *help-freedom*, which intuitively implies that an operation does not *altruistically* help a concurrent (slow) operation to guarantee wait-freedom. In contrast to that, in lock-free algorithms, help-optimality not only implies an absence of altruistic helping but also indicates that an operation is aware of intended modification getting applied as a part of a modification by a concurrent operation. Thereby, on account of helping, the aggregate number of steps is minimized. In this work, we do not delve into a formal comparison between help-optimality and help-freedom.

As a rationale behind the term help-optimality, we would like to underline our aim to optimize a lock-free design with respect to the number of (CAS execution) steps incurred in helping under the constraints such as an optimal memory footprint and an optimal amortized step complexity.

3.6 Experimental Evaluation

3.6.1 Experimental Set-up

In this section, we present a detailed performance analysis of our implementations in both C/C++ and Java. The following concurrent linked-lists and lock-free BSTs are compared:

1. **HO-LL:** An optimized variant of lock-free linked-list of [46], where a CONTAINS does not perform help.
2. **Lazy-LL:** Lock-based linked-list of [48], in which logically removed nodes are ignored during traversal.
3. **CWT-LL:** Lock-free linked-list described in section 3.3.
4. **EFRB-BST:** Lock-free external BST of [33], in which both ADD and REMOVE require help to complete at a conflict.
5. **LF-SKIPLIST:** Concurrent skip-list implementation that is part of the java.util. concurrent package [62].
6. **NM-BST:** Lock-free external BST of [71], in which multiple nodes under REMOVE by different threads are cleaned out together similar to algorithm 3.3.

7. **CWT-BST:** Lock-free BST described in section 3.4.

8. **CWT-Simple-BST:** Lock-free BST of section 3.2.

We performed our evaluations on a dual-socket server with a 3.4 GHz Intel (R) Xeon (R) E5-2687W-v2 having 16 physical cores (32 hardware threads by hyper-threading), 16 GB of RAM, running Ubuntu 13.04 Linux (3.8.0-35-generic x86_64) with Java HotSpot (TM) 64-Bit Server VM (build 25.60-b23, mixed mode). We compiled all Java implementations with `javac` version `1.8.0_60` and used the runtime flags `-d64 -server`. C/C++ implementations were compiled with `g++` version `4.9.2`, `O3` optimization and TCMalloc [38] to reduce dynamic memory allocation overheads.

We compared the performance in terms of the throughput as Million operations per second (Mops/s). We measured the memory consumption as the change in heap-size of the JVM on loading the set with initial elements and on execution of the workload. Each experiment was run for 5 seconds, then the average over 6 trials was taken under the following parameters:

1. **Workload Distribution:** Similar to [71], we considered three workload distributions: (a) *write-dominated*: 0% CONTAINS, 50% ADD and 50% REMOVE. (b) *mixed*: 70% CONTAINS, 20% ADD and 10% REMOVE, and (c) *read-dominated*: 90% CONTAINS, 9% ADD and 1% REMOVE.

2. **Set Size:**. The maximum set size depends on the range of the keys. We consider the following key ranges for the linked-lists: 2^7, 2^9, 2^{10} and 2^{12}. For the BSTs we consider the ranges: 2^{10}, 2^{14}, 2^{17} and 2^{20}. In each experiment, the set is pre-loaded with roughly half the keys in the key range.

C/C++ and Java Implementations: As we pointed out in the section 3.1, many concurrent data structures are designed with language specific dependencies and as such offer varying performance in different languages. Additionally, original implementations of the algorithms are available either in Java or C/C++. With this in mind, we implemented our new language-portable lock-free algorithms in both C/C++ and Java. The code is available at `https://github.com/bapi/ConcurrentSet`.

To ensure a fair comparison, we implemented our C/C++ versions of the algorithms as part of the ASCYLIB library [29], with SSMEM - a memory allocator with epoch-based garbage collection. We used the same benchmarks which are part thereof. HO-LL in Java employs RTTI. For locking, Lazy-LL uses ReentrantLock in Java and a ticket lock in C/C++.

3.6.2 Performance Results and Discussion

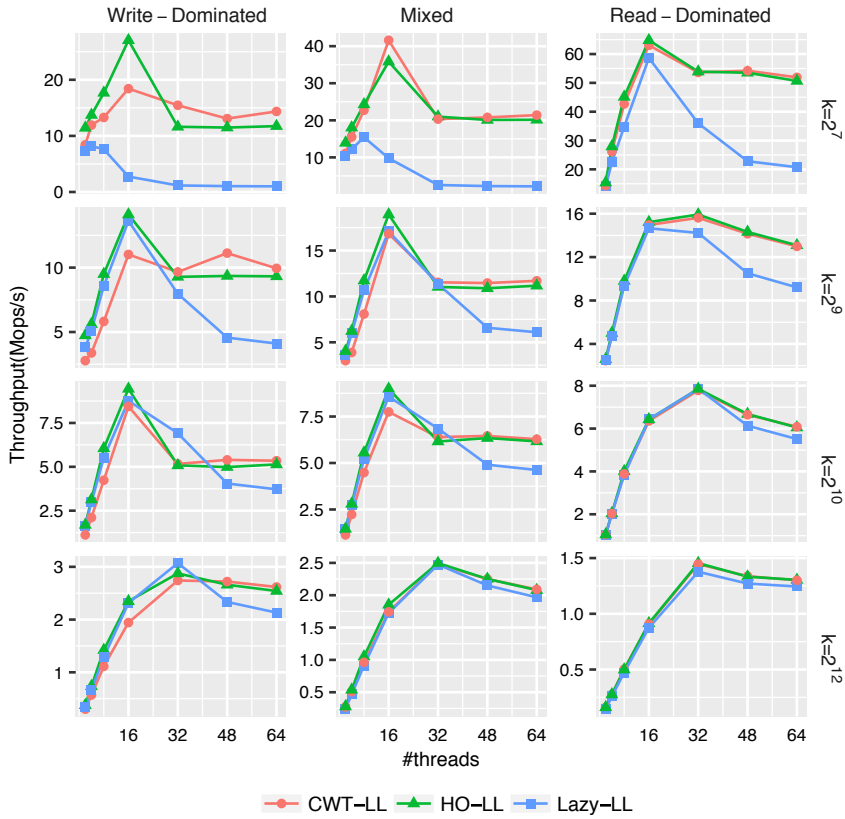


Fig. 3.7: Concurrent linked-list algorithms: Java Implementation

Figures 3.7 and 3.8 depict the comparative performance of linked-list-based Set algorithms in Java and in C/C++, respectively. At low contention i.e. with read-dominated workloads and large key space sizes, the lists scale with increasing thread count. CWT-LL performs on a par with HO-LL in both Java and C/C++. In the high contention cases, mainly write-dominated and small key space sizes, Lazy-LL degrades significantly with increasing thread count. This is mainly due to the increased contention on the locks and cache misses resulting from the lock migrations. Contention increases as the list gets shorter in size with a smaller key space size. At high contention CWT-LL outperforms

HO-LL by 5% for Write-Dominated and 3%-6% for Mixed workloads. This can be attributed to the local restart and the ability to clean out multiple nodes in a single step.

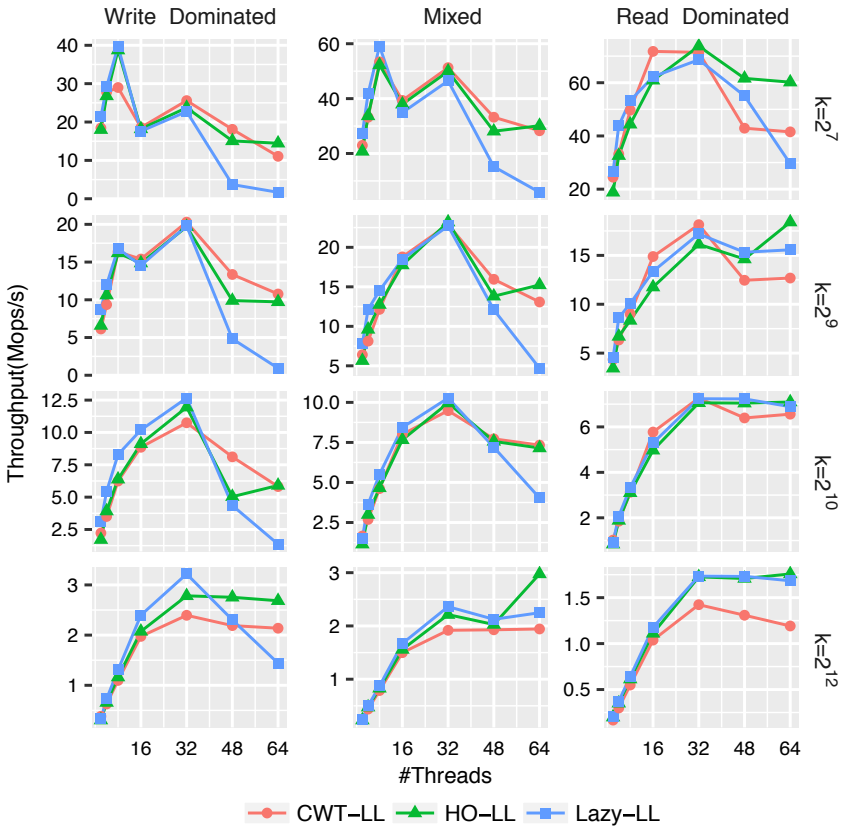


Fig. 3.8: Concurrent linked-list algorithms: C/C++ Implementation

In C/C++ we observe similar relative performance, however, the list performance degrades significantly when the cores are saturated with threads (most especially in the write-dominated workload). The effect of oversubscribing the cores with more threads is bigger in Lazy-LL than that in other algorithms as a result of increased lock-contention.

Figures 3.9 and 3.10 shows the comparative performance of considered lock-free BST and skip-list algorithms in Java and C/C++, respectively. We

have not included CWT-Simple-BST here considering its incomparable memory footprint. It is clear that among the Java implementations, CWT-BST offers the best throughput for all key space sizes and workloads. CWT-BST outperforms EFRB-BST by 10%- 50% and LF-SKIPLIST 20%-100% over Write-Dominated and Mixed workloads.

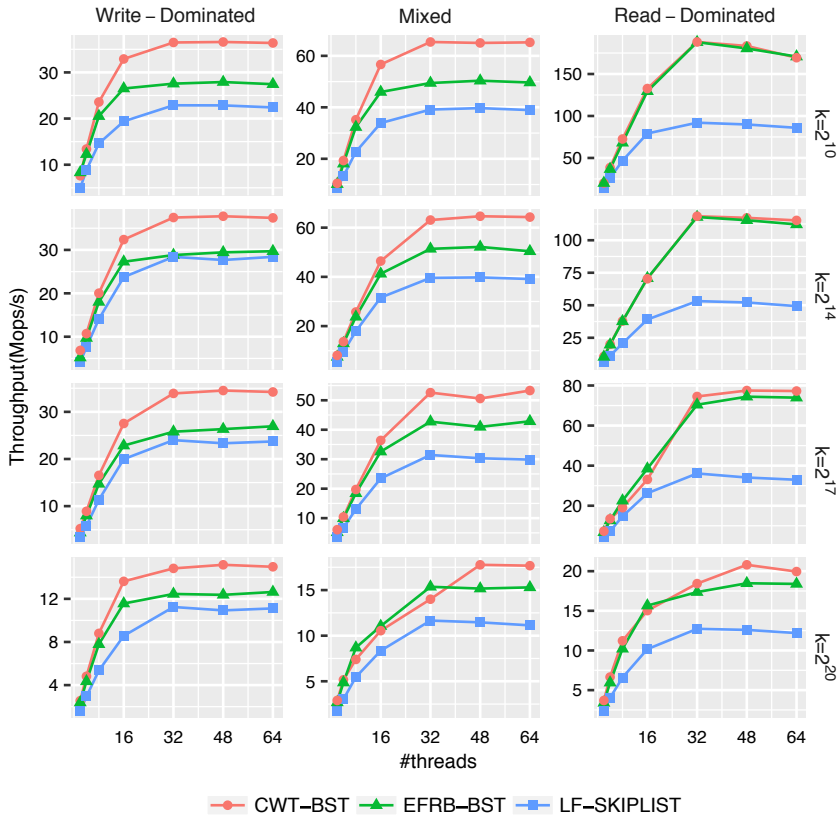


Fig. 3.9: Lock-Free BST algorithms: Java Implementation

In C/C++, NM-BST outperforms others at high contention. This can be attributed to the advantage of bit-stealing over explicit object allocations. Bit masking, unmasking and other bitwise operations in C/C++ are simple and faster than object creation, however not portable to other high-level languages. As we increase the key space size, CWT-BST offers performance similar to NM-BST, especially in Mixed and Read-Dominated workloads, even dominat-

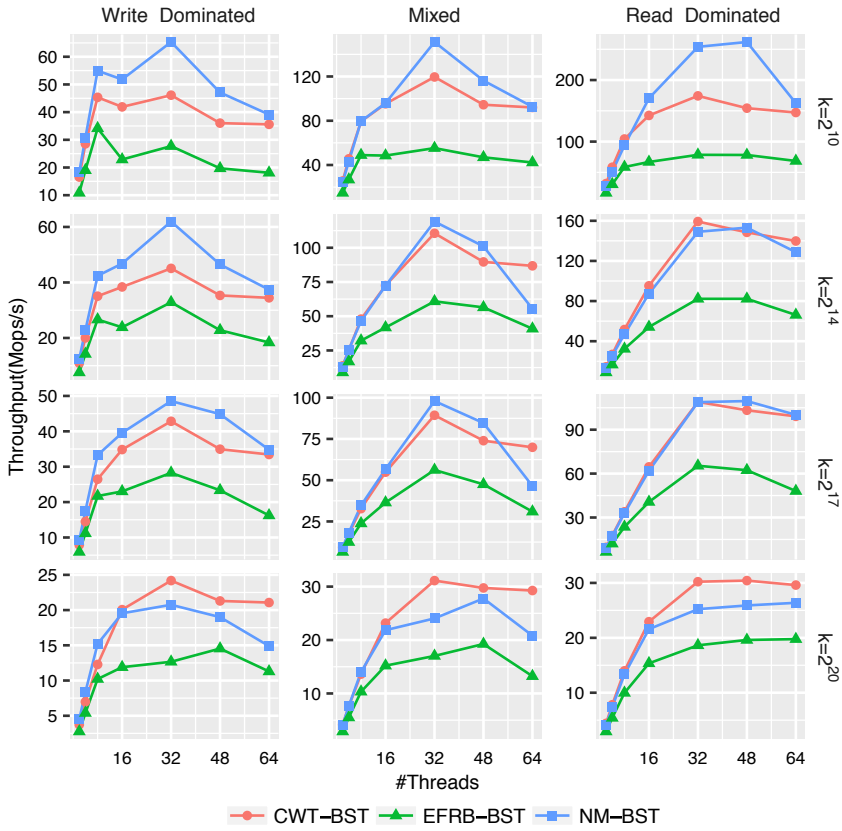


Fig. 3.10: Lock-Free BST algorithms: C/C++ Implementation

ing in the low contention case with key space (2^{20}) by 3%-15%. This can be attributed to a comparative cost of object allocation but lowered cost of reading a pointer without bit unmasking. It can be noted that although EFRB-BST implementation is based on bit-stealing, CWT-BST outperforms it in every case scenario by 10%-50%.

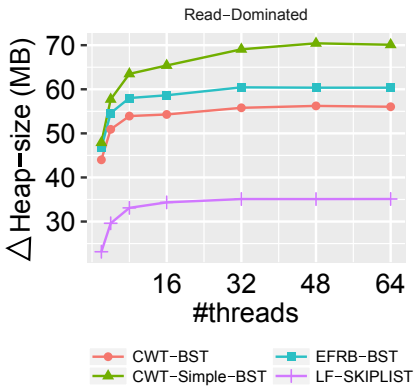


Fig. 3.11: Heap size change

Memory Reclamation: As a REMOVE operation allocates a splice node, the load on garbage collector certainly increases. However, as illustrated in the Figure 3.11, in a garbage collected environment, CWT-BST experiences no unexpected growth in heap-memory usage. In fact, on this account, it outperforms EFRB-BST. Though the figure presents a case for one workload setting, we observed similar relative memory usage with every workload settings. Nevertheless, we do advise that these techniques should not be used without memory reclamation.

Chapter Summary

In this chapter, we introduced the notion of help-optimality in a lock-free algorithm. Intuitively, in a lock-free data structure, which satisfies help-optimality, at a conflict over modification of a shared variable, we avoid both offer and acceptance of help in form of a step comprising a CAS *execution. Help-optimality consists of the notions of selfishness and help-awareness. Selfishness implies optimization of the count of steps of* CAS *executions by an obstructed operation, whereas help-awareness implies the same for an obstructing operation.*

The present work is mostly experimental in nature to demonstrate the utility of the concept of help-optimality in a lock-free linked-list and a BST. In future, we plan to develop formal specifications of the introduced notions.

Following a state-of-the-art implementation of the lock-free skip-list in Java library, in this chapter, we designed the lock-free data structures to provide a language-portable implementation. We experimentally showed that such an implementation performs on a par with highly optimized implementations in C++ which use the technique of bit-stealing. The Go programming language, which reasonably focuses on concurrency, provides pointers without pointer-arithmetic and does not provide type-inheritance. We believe that with growing popularity of such languages, designing language-portable lock-free data structures will be increasingly significant.

4

AMORTIZED COMPLEXITY OF LOCK-FREE INTERNAL BINARY SEARCH TREE

Chapter Abstract

*In this chapter, we present an algorithm for the lock-free internal binary search trees (BST). We prove that the Set ADT operations–*ADD, REMOVE *and* CONTAINS*–implemented by the algorithm are linearizable. We show that the amortized step complexity of each of the operations is $O(H(n) + c)$, where c is the contention during the execution and $H(n)$ is the height of the BST (with n number of nodes) at its invocation. The operations* ADD *and* REMOVE *use the single-word compare-and-swap (*CAS*) atomic primitive, which is readily available in common multi-core architectures.*

4.1 Introduction

In literature, there are lock-free as well as wait-free singly linked-lists [35, 83], lock-free doubly linked-list [82], lock-free hash-tables [67] and lock-free skip-lists [35, 81]. In case of non-blocking concurrent BSTs, there have been some recent publications. A multi-word compare-and-swap (MCAS) based lock-free BST implementation was presented by Fraser in [36]. However, MCAS is not a native atomic primitive provided by available multi-core chips and is very costly to be implemented using single-word CAS. Bronson et al. proposed an optimistic lock-based partially-external BST with relaxed balance [15]. Ellen et al. presented lock-free external binary search tree [33] based on co-operative helping technique presented by Barnes [11]. Though their work did not include an analysis of complexity or any empirical evaluation of the algorithm, the contention window of update operations in the data-structure is large. Also, because it is an external binary search tree, REMOVE is simpler at the cost of extra memory to maintain internal nodes without data. Howley et al. presented a lock-free

internal BST [53] based on similar technique. A software transactional memory based approach was presented by Crain et al. [28] to design a concurrent red-black tree. While it seems to outperform some coarse-grained locking methods, it easily falls behind in performance when compared to a carefully tailored locking scheme as in [15]. Recently, two lock-free external BSTs [71, 17] and a lock-based internal BST [30] have been proposed. All of these works lack theoretical complexity analysis.

A common predicament for the existing lock-free BST algorithms is the following. Consider multiple modify operations contending at a leaf node. Now, if a REMOVE operation among them succeeds then all other operations have to restart from the root. It results in the complexity of a modify operation being $O(cH(n))$ where $H(n)$ is the height of the BST on n nodes and c is the measure of contention. It may grow dramatically with the growth in the size of the tree and the contention. In addition to that, CONTAINS operations have to be aware of an ongoing REMOVE of a node with two children, otherwise it may return an invalid result. Hence in the existing implementations of lock-free internal BST [53], a CONTAINS operation may have to restart from the root on realizing that the return may be invalid if more nodes are not scanned. The external or partially-external BSTs remain immune to this problem at the cost of extra memory used by the routing internal nodes. Our algorithm solves both these problems elegantly. The CONTAINS operations in our BST enjoy oblivion of any kind of modify operation. And, the modify operations after helping a concurrent modify operation restart not from the root, rather from a level in the vicinity of failure. It ensures that all the operations in our algorithm run in $O(H(n) + c)$. This is our main contribution. Ellen et al. [32] improved the external BST by Ellen et al. [33] using thread local stacks for each thread to achieve similar complexity as ours.

Given the requirements of a concurrent data-structure in terms of number of shared-memory words to modify for an update, we always strive to exploit maximum possible disjoint-access-parallelism [57]. It facilitates in maximizing the progress of concurrent operations if an operation needs to modify multiple shared memory words to complete its steps. The lock-free methods for BST [33, 53], in order to use single-word CAS for atomically modifying the links outgoing from a node, and yet maintain correctness, store a flag as an operation field or some version indicator in the node itself, and hence a modify operation "holds" a node. This mechanism of holding a node, specifically for a REMOVE, can reduce the progress of two concurrent operations which may be otherwise non-conflicting. In [71], a flag is stored in a link instead of a node in an external BST. We found that even in an internal BST it is indeed possible that a REMOVE operation, instead of holding the node, just holds the links connected to

and from a node in a predetermined order so that improved progress of concurrent operations working at disjoint memory words corresponding to the links could be achieved. The presented lock-free design using "storing a flag" at a link instead of a node significantly improves the disjoint-access-parallelism compared to the existing design as in [53]. This is our next contribution.

Helping mechanism which ensures non-blocking progress may prove counterproductive to the performance if not used judiciously. In a previous version of this work [24] we proposed an adaptive helping mechanism to choose depending on the read-write load. However, with empirical observations we found that helping during traversal always reduces the performance and so it is better not to help a pending REMOVE operation during traversal. Our algorithm requires only single-word atomic CAS primitives along with single-word atomic read and write which practically exist in all the widely available multi-core processors in the market. Based on our design, we implement a Set ADT. We prove that our algorithm is linearizable [50]. We also present complexity analysis of our implementation.

The contributions of this work are the following:

1. We present an algorithm for a lock-free internal BST that implements a Set abstract data types (ADT). The ADT operations ADD, REMOVE, and CONTAINS are provably linearizable. The presented algorithm is *help-aware* (see chapter 3).

2. On a CAS failure at a conflict, the modify operations in our algorithm restart locally to optimize the amortized step complexity per operation.

3. We prove that the amortized number of steps per operation in our algorithm is $O(H(n) + c)$, where $H(n)$ is the height of the BST on n nodes and c is the measure of contention.

Further in this chapter, first, we present the basic tree terminologies (section 4.2,). Thereafter, the proposed algorithm is described (section 4.3). Having described the algorithms, we present a discussion on the correctness and the progress of our concurrent implementation along with an amortized analysis of its time complexity (section 4.4).

4.2 Preliminaries

A *binary tree* is an ordered tree in which each *node* x has a *left-child* and a *right-child* denoted as $left(x)$ and $right(x)$ respectively, either or both of which may be *external*. When both the children are external the node is called a *leaf*, with

one external child a *unary* node and with no external child a *binary* node, and all the non-external nodes are called *internal* nodes. If a node is used to store data it is called a *data node* else it remains in the tree for just facilitating a traversal and is called a *routing-node*. We denote the parent of a node x by $p(x)$ and there is a unique node called *root* s.t. $p(root) = null$. Each parent is connected with its children by links. We indicate the link (often we shall be using the terms pointer and link interchangeably) emanating from a node *k* and incoming to a node *l* by $k \rightsquigarrow l$.

We are primarily interested in implementing an ordered *Set* ADT - *binary search tree* using a binary tree in which each node is associated with a unique key k selected from a totally ordered universe. A node with a key k is denoted as $x(k)$ and x if the context is otherwise understood. Determined by the total order of the keys, each node x has a *predecessor* and a *successor*, denoted as $pre(x)$ and $suc(x)$, respectively. We denote height of x by $ht(x)$, which is defined as the distance of the deepest leaf in the *subtree* rooted at x from x. Height of a BST is $ht(root)$. In this chapter, we consider an *internal BST*, in which all the internal nodes are data-nodes and the external nodes are usually denoted by null. There is a *symmetric order* of arranging the data - all the nodes in the *left subtree* of $x(k)$ have keys less than k and those in its *right subtree* have keys greater than k, and so no two nodes can have the same key.

The ADT operations in our BST design are as described here. To query whether a BST CONTAINS a data with key k, at every *search-step* we utilize its symmetric order to look for the desired node either in the left or in the right subtree of the *current node* if the key not matched at it, unless we reach an external node. If the key matches at a node, we return true (or address of the node if needed), otherwise false. To ADD data we query by its key k. If the query reaches an external node we replace this node with a new leaf node $x(k)$. The REMOVE operations vary with the location of the node containing the data to remove. First of all, we check whether $x(k)$ is in the BST. If the BST does not contain $x(k)$, false is returned. On finding $x(k)$, we perform delete as following. If it is a leaf then we just replace it with an external node i.e. null. In case of a unary node its only child is connected to its parent. For a binary node $x(k)$, it is first replaced with a copy of $pre(x(k))$, which may happen to be a leaf or a unary node, and then $pre(x(k))$ is removed. Thus, the ADT operations do maintain the symmetric order of the BST.

4.3 Our Algorithm

4.3.1 Design Fundamentals

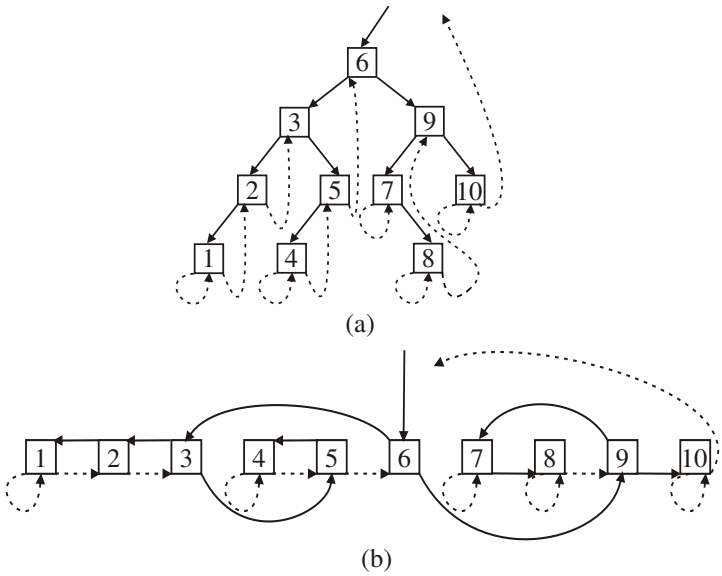


Fig. 4.1: Basic Design of the Binary Search Tree.

The basic design of the lock-free internal BST is shown in the Figure 4.1. To implement a lock-free BST, we represent it in a *threaded* format [75]. In this format, if the left or the right child pointers at x is null and so corresponds to an external node, it is instead connected to $pre(x)$ or $suc(x)$, respectively. Some indicator is stored to indicate whether a child-link is used for such a connection. This is called *threading* of the child-links. In our design, we use the null child pointers at the leaf and unary nodes as following - right child pointer, if null, is threaded and is used to point to the successor node, whereas, a similar left child pointer is threaded to point to the node itself, see Figure 4.1(a). In this representation a binary tree can be viewed as an ordered list with exactly two outgoing and two incoming pointers per node, as shown in Figure 4.1(b). Also, among two incoming pointers, exactly one is threaded and the other is not. Further, if $x(k_i)$ and $x(k_j)$ are two nodes in the BST and there is no node $x(k)$ such that $k_i \leq k \leq k_j$ then the interval $[k_i,\ k_j]$ is called *associated* with the threaded link incoming at k_j.

We exploit this symmetry of the equal number of incoming and outgoing pointers. A usual traversal in the BST following its symmetric order for a predecessor query, is equivalent to a traversal over a *subsequence* of the ordered-list produced by an in-order traversal of the BST, which is exactly the one shown in Figure 4.1(b). This is made possible by the threaded right-links at leaf or unary nodes. Though in this representation, there are two pointers in both incoming and outgoing directions at each node, a single pointer needs to be modified to ADD a node in the list. To REMOVE a node we may have to modify up to four pointers. Therefore, ADD can be as simple as that in a lock-free single linked-list [35], and REMOVE is no more complex than that in a lock-free double linked-list [82]. A traversal in a lock-free list may enjoy oblivion from a concurrent REMOVE of a node. Also in our design of internal BST, a traversal can remain undeterred by any ongoing modification, unlike that in existing lock-free implementations of internal BSTs [53, 36].

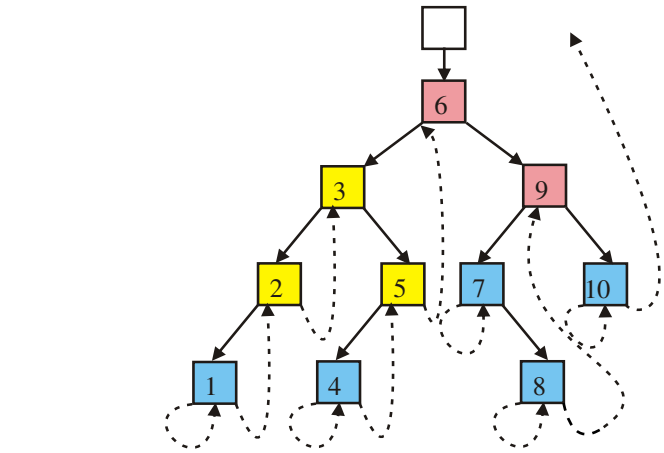
In designing an internal BST in a concurrent set-up, the most difficult part is to perform an error-free REMOVE of a binary node. To remove a binary node we replace it with its predecessor, and hence, the incoming and outgoing links of the predecessor also need to be modified in addition to the incoming and outgoing links of the node itself. According to the number of links needed to be modified in order to remove a node, unlike traditional categorization of nodes of a BST into leaf, unary and binary, we categorize them into three categories as shown in Figure 4.2. The categorization characteristic is the origin of the threaded incoming link into the node, hereafter we call this link the *order-link*. Nodes belonging to category 1 are those whose order-link emanates from themselves; to category 2, it emanates from the left-child of the node, and to category 3 are those whose incoming order-link emanates from a "distant" node in its left-subtree. We name the node where the order-link emanates from, an *order-node*.

Given this categorization of nodes, we are now ready to describe the operations in the lock-free internal BST.

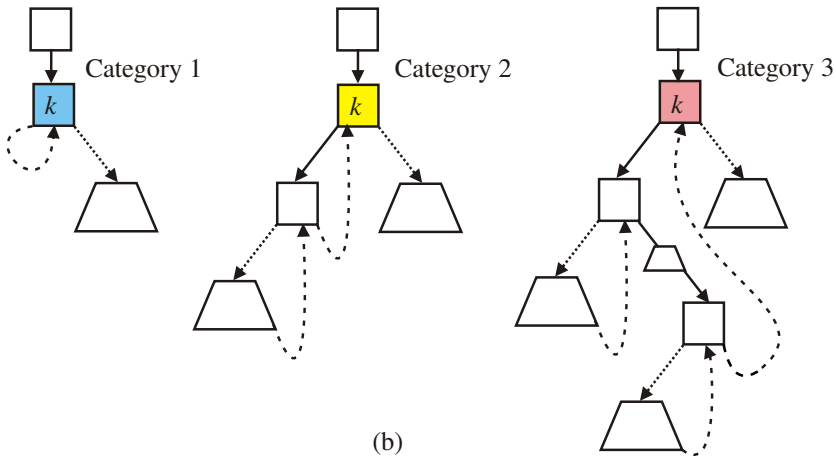
4.3.2 The Lock-free Algorithm

(A) The basic lock-free design

We described in the section 4.2, that if the node under REMOVE is binary, it is first replaced with its predecessor and then the predecessor is removed. However, in a concurrent setting, that structural change leads to a large number of `CAS` operations, as can be seen in our previous work [24]. Specifically, that procedure does not support proving the claim of the upper bound of amortized



(a)



(b)

Fig. 4.2: Category of Nodes in the BST for REMOVE.

number of steps per operation in the lock-free internal BST, which we present in section 4.4. Therefore, here we first describe a simple method to remove a node from the internal BST in a sequential setting. Thereafter, we shall describe the lock-free implementation.

For a node $x(k)$ of category 1, where the left child-link is its order-link emanating from and terminating at itself, (a) we just connect the incoming link from the parent of $x(k)$, $p(x(k))\rightsquigarrow x(k)$ to the right-child if the right-child-link points to a node in the right-subtree, and (b) we change the incoming link $p(x(k))\rightsquigarrow x(k)$ to a threaded link and connect it to $suc(x(k))$, if the right-child-link is threaded and pointing to $suc(x(k))$. We can notice that it is in fact equivalent to replacing the value of link $p(x(k))\rightsquigarrow x(k)$ in the node $p(x(k))$ with the value of the right-child-link of $x(k)$. For a category 2 node $x(k)$, where the order-link emanates from the left-child of $x(k)$, we connect the right-child or the $suc(x(k))$ with the left-child by a non-threaded or a threaded link, respectively. Additionally, we update the link $p(x(k))\rightsquigarrow x(k)$ at the node $p(x(k))$ with the value of the left-child-link of $x(k)$. For a category 3 node $x(k)$, we do not follow the traditional approach of replacing it with a copy of $pre(x(k))$ and removing the node $pre(x(k))$. In place of that, we follow a simple approach and in effect treat a category 3 node as a category 2 node. Accordingly, here as the order-link of $x(k)$ emanates from $pre(x(k))$, we connect the right-child or the $suc(x(k))$ with the node $pre(x(k))$ by a non-threaded or a threaded link, respectively, and update the link $p(x(k))\rightsquigarrow x(k)$ at the node $p(x(k))$ with the value of the left-child-link of $x(k)$.

In a sequential setting, when REMOVE($x(k)$) modifies the pointer $p(x(k))\rightsquigarrow x(k)$, no operation is executed concurrently with a possibility to modify either of the child-links of $x(k)$. However, in a concurrent setting, where these pointers are shared by multiple operations, an ADD operation can concurrently modify any of these pointers. It may result into the newly added node not being a part of the BST. Similarly, a concurrent REMOVE operation trying to remove a child of right-child of $x(k)$ may end up connecting $p(x(k))$ to the removed child which results into a wrong outcome. Essentially, for a correct concurrent implementation of modify operations in a BST, we need to keep the child links *fixed* when $p(x(k))\rightsquigarrow x(k)$ is updated or for that matter the link $pre(x(k))\rightsquigarrow x(k)$ i.e. the order link of $x(k)$ is updated.

For a lock-free synchronization we can not use locks to keep these shared links fixed. Instead of locks, we use a protocol for operations to perform *helping*, whenever they encounter shared pointers fixed (although not by a lock) by concurrent operations, i.e. obstructed. This ensures that no non-faulty thread is blocked due to a delayed or crashed thread and thereby providing progress guarantee.

More precisely, we use special *descriptors* over links to inform any obstructed operation about the stage of REMOVE operation indicating the links that are already modified. Along the same lines as in the last chapter, to keep the algorithms language-portable, we use splice-nodes to delegate a link with descriptor or that is *not clean*. The splice nodes are identified by their keys. Additionally, as we saw in the last chapter, we can use these nodes to keep an extra pointer to a node up in the BST to facilitate local restart. Furthermore, we also save some new memory allocation by way of splice nodes storing the original links on which descriptors are put during a REMOVE operation.

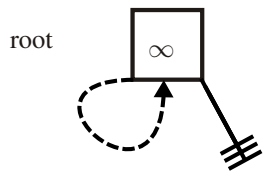


Fig. 4.3: Sentinel Node: Internal lock-free BST

The node structure of a typical node is shown at line 1 in Algorithm 4.1. A node consists three pointer fields lt, rt and preLink in addition to a key field k. Without ambiguity, we shall use k to denote a node with key k. The pointer fields lt and rt connect a node to its left and right children respectively. The preLink is a special pointer that connects a node to its predecessor when it is under REMOVE. We use the method `Dir` to find the child direction **L** or **R** of a node containing a query key with respect to a node in the BST. The method `Child` returns the child link of a node given the address of the node and the child direction. Whereas, the method `ChCAS` performs a `CAS` at the child link of a node, given its address and the child direction. To avoid special cases, we use a sentinel node root as given in line 2 and shown in Figure 4.3. The special key ∞ relates to any other key in the dataset as $k < \infty$ for all keys k in the dataset.

Following the technique used in the last chapter, we essentially use descriptors over the links for concurrent operations to synchronize. A link is not "clean" if it has a descriptor over it. A special descriptor - thread (using the same terminology as in the thread-based BST design) - is used for the threaded-links. A link with descriptor is represented by a *splice node*, as described in the last chapter. Splice nodes are distinguished based on their key. The pointer field rt is used to store the original pointer value of the link, whereas, the fields lt and preLink are used to store additional addresses which help in local restart of a modify operation and some additional programming optimizations. We use

Algorithm 4.1. Lock-free Internal BST: Basic Operations

```
1  struct Node {K k; Node* lt, rt, preLink;};
2  root := Node(∞1, GetTh(root), null, null);
3  Dir(Node* par, K k) {return k ≤ par·k ? L : R};
   Child(Node* par, dir cD)
4  | return cD == L ? par·lt : par·rt;
   ChCAS(Node* par, Node* exp, Node* new, dir cD)
5  | if (cD == L) and par·lt == exp then
6  | | return CAS(par·lt·ref, exp, new);
7  | else if (cD == R) and par·rt == exp then
8  | | return CAS(par·rt·ref, exp, new);
9  | else return false;
10 GetTh(Node* n) {return Node(∞2, null, n, null);}
   GetMark(Node* n, Node* p)
11 | return Node(−∞0, p, n, null);
   GetThMark(Node* n, Node* p)
12 | return Node(∞0, p, n·rt, n);
   GetThFlag(Node* n, Node* p)
13 | return Node(∞1, p, n, null);
14 IsTh(Node* n) {return n·k ≥ ∞2;}
15 IsMark(Node* n) {return n·k == −∞0;}
16 IsThMark(Node* n) {return n·k == ∞0;}
17 IsThFlag(Node* n) {return n·k == ∞1;}
```

following category of links with descriptors and the corresponding splice node types.

1. `Thread`: Used for order-links of a node. The key is ∞_2. The rt pointer stores the node that the link is order-link of.

2. `Mark`: Used to fix the left or right child-link of a node under REMOVE. The child-link is originally *un-threaded*. The key of the splice node is $-\infty_0$. The rt pointer stores the original child-link. The lt pointer points to the parent of the node under REMOVE.

3. `ThMark`: Used to fix a right child-link of a node under REMOVE which is originally *threaded*. The key of the splice node is ∞_0. The rt pointer points to the successor of the node under REMOVE. The lt pointer points to the parent of the node under REMOVE. The preLink pointer stores the original threaded right child-link.

4. `ThFlag`: Used to fix the order-link of a node under REMOVE. The key of the splice node is ∞_1. The rt pointer stores the address of the node under REMOVE. The lt pointer points to the parent of the node under REMOVE.

The special keys in the splice nodes are such that $|k|<\infty_2<\infty_1<\infty_0<\infty$ for all keys k in the BST. The descriptors `ThMark` and `ThFlag` are considered as composite descriptors. The links with descriptors are created using the methods `GetTh`, `GetMark`, `GetThMark` and `GetThFlag`, see line 10 to 13 in algorithm 4.1. The methods `IsMark`, `IsThMark` are `IsThFlag` used to check whether a link has a descriptor `Mark`, `ThMark` or `ThFlag`, respectively, see lines 15 to 17. Whereas, the method `IsTh`, line 14, checks whether a link has either the descriptor `Thread` or any composite descriptor.

(B) The Steps of the REMOVE *Operation*

With that, we now describe the REMOVE operation. As the number of links to be modified in the case of category 2 and category 3 nodes are equal, we take them together and before that we describe the steps of REMOVE operation of category 1 nodes.

As shown in the Figure 4.4, given a node $x(k)$ with its order-link, see Figure 4.4(a), the first step is to replace the order-link with a link containing the descriptor `ThFlag`, see Figure 4.4(b), using a `CAS`. On that, the order-link of the node $x(k)$ is fixed and a new node can not be added there. Any ADD operation that aims to add a new node there, will help the pending REMOVE operation

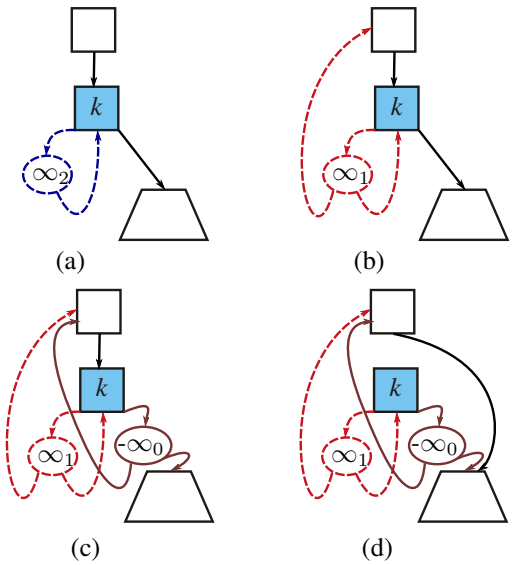


Fig. 4.4: REMOVE steps: Category 1 Node with the right child.

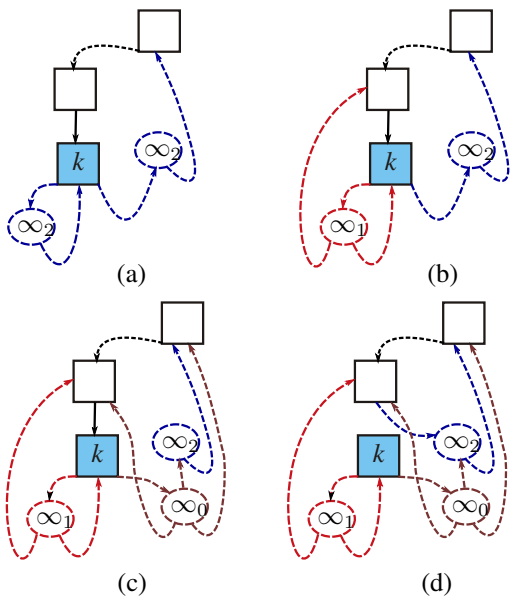


Fig. 4.5: REMOVE steps: Category 1 Node without the right child.

of $x(k)$. The descriptor ThFlag indicates that the next step is to replace the right child-link with a link containing the descriptor Mark and pointing to the right child as in Figure 4.4(c). With that, the only child link of $x(k)$ is fixed which makes any obstructed operation to help the REMOVE of $x(k)$. Having done that, the incoming link from the parent of $x(k)$ is replaced with the original right child-link of $x(k)$, see Figure 4.4(d), using a CAS.

In case the right child-link of $x(k)$ is itself a threaded link, which is order-link of the successor of $x(k)$, the steps are shown in the Figure 4.5. Here in step (c) the right child-link is replaced using a CAS to a link with descriptor ThMark, see Figure 4.5(c). Please note that the original right link of $x(k)$ is stored in the preLink of the link with ThMark, which provides an optimization to avoid a new node allocation. Finally the incoming parent link to $x(k)$ is updated using a CAS to the original right link of $x(k)$, see Figure 4.5(d).

For a category 2 node, its left-child itself is its order-node. However, the overall steps of REMOVE operation are same for category 2 and category 3 nodes. The steps of REMOVE operation for a node of category 2 or 3, with its right child-link pointing to its right-child, are shown in the Figure 4.6. Given $x(k)$, as shown in the Figure 4.6(a), the first step is to replace the order-link with a link with descriptor ThFlag using a CAS, see Figure 4.6(b). In the next step, the left child-link is replaced with a link containing the descriptor Mark using a CAS, see Figure 4.6(c). After that, the right child-link is replaced with a link containing the descriptor Mark using a CAS, see Figure 4.6(d). With that both the child links of the node under REMOVE are fixed. Also, the order-link is fixed and therefore no new node can be added at this link, and any ADD operation getting obstructed there helps the pending REMOVE operation.

The next step is to update the incoming order-link and the parent-link of the node $x(k)$. First we update the order-link using a CAS as shown in Figure 4.6(e), the predecessor of $x(k)$ is connected to the right-child of $x(k)$. Finally the incoming parent link is updated using a CAS to connect the parent to the left-child of $x(k)$ as shown in Figure 4.6(f). With that the node $x(k)$ is made unreachable from any node in the BST.

The case of REMOVE of a node $x(k)$, which is of the category 2 or 3, are shown in the Figure 4.7. The steps are absolutely same with only difference with the step of fixing the right link of $x(k)$, which is replaced with a link containing the descriptor ThMark.

Adding a new node in the BST requires a single CAS and always happens at a threaded-link. If the threaded-link is found marked or flagged, the ADD operation helps the pending REMOVE operation. To perform a CONTAINS, the query either terminates at a node containing the query key, in which case it returns true or at a threaded-link, in which case it returns false.

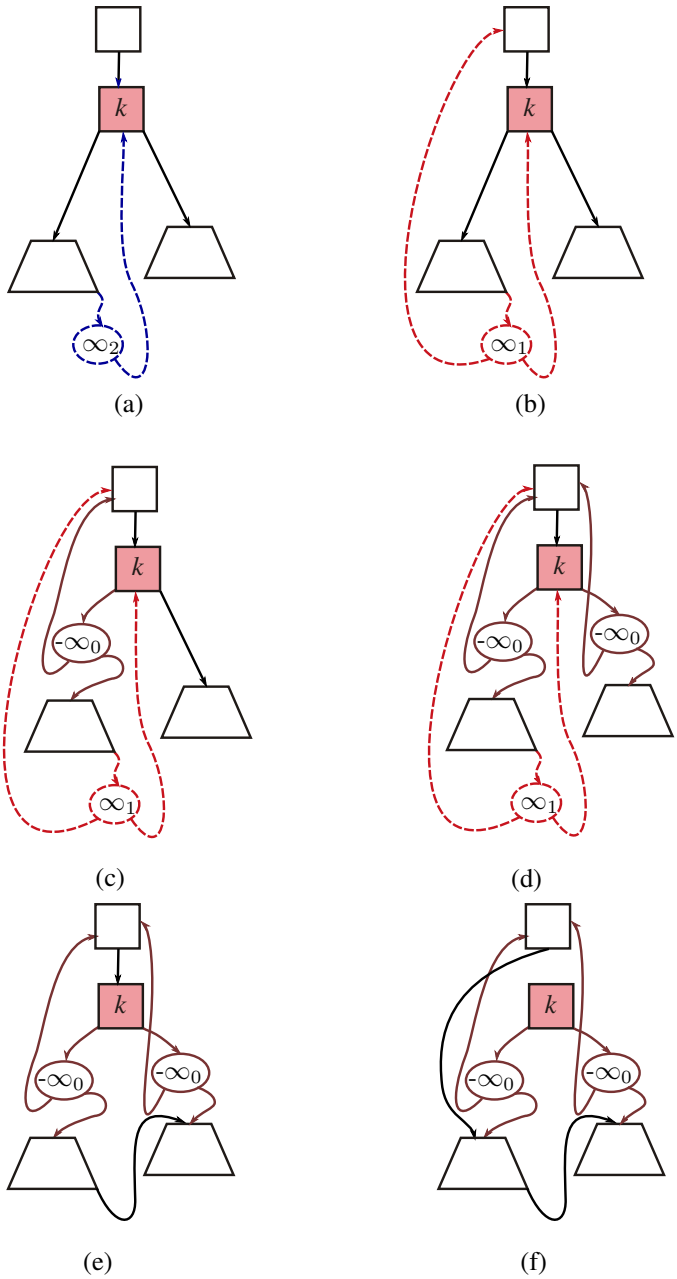


Fig. 4.6: REMOVE steps: Category 2 or Category 3 Node with right child.

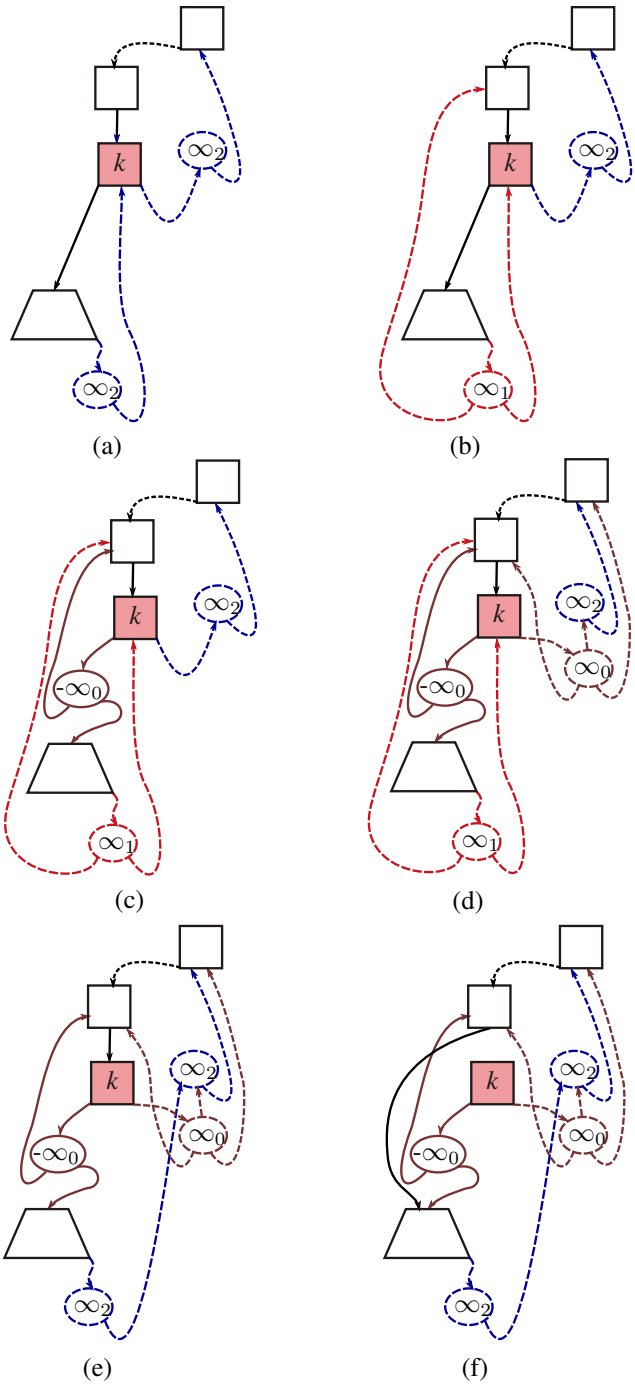


Fig. 4.7: REMOVE steps: Category 2 or Category 3 Node without right child.

Algorithm 4.2. Lock-free Internal BST: REMOVE operation

```
    REMOVE(K k)
1   mode := true; d := null; p := root·ref;
2   retry:
3   c := Child(p, Dir(p, k));
4   while IsMark(c) do {p := c·lt; c := Child(p, Dir(p, k));} ;
5   while true do
6     if !IsTh(c) then
7       n := Child(c, Dir(c, k));
8       while IsMark(n) do
9         pr := Pr(c); r := GetRepHelp(c, p, pr);
10        if !ChCAS(p, c, r, Dir(p, k)) then goto retry;
11        c := r; n := Child(c, Dir(c, k));
12    else {n := c; c := p;};
13    if c·k == k and d == null then {par := p; d := c;};
14    if IsTh(n) then
15      if mode == true then
16        if n·rt ≠ d then return false;
17        if IsThFlag(n) then
18          r := GetRep(c, n, p, false);
19          if r == null then return false;
20          else if ChCAS(p, d, r, Dir(p, k)) then return false;
21          p := n·lt;
22        else if IsThMark(n) then
23          pr := Pr(c); r := GetRepHelp(c, p, pr);
24          ChCAS(p, c, r, Dir(p, k));
25          if p == c then p := n·lt;
26        else
27          m := GetThMark(p, d);
28          if ChCAS(c, n, m, Dir(c, k)) then
29            mode := false; r := GetRep(c, m, p, false);
30            if r == null then return true;
31            else if ChCAS(p, d, r, Dir(p, k)) then return true;
32            p := par·lt;
33        goto retry;
34      else return true;
35    else {p := c; c := n;};
```

(C) The Lock-free Algorithm

Algorithm 4.3. Lock-free Internal BST: Help method

```
   GetRep(Node* oNode, Node* oLink, Node* p, bool isHelp)
1  | n := oLink·rt;
2  | if !CAS(n·preLink, null, oNode) and !isHelp then
3  | | return null;
4  | if n ≠ oNode then
5  | | lC := MarkLt(n, p, isHelp);
6  | | if lC ≠ null then
7  | | | rC := MarkRt(n, p, isHelp);
8  | | | if rC ≠ null then
9  | | | | ChCAS(oNode, oLink, rC, false); return lC;
10 | | | else return null;
11 | | else return null;
12 | else return MarkRt(n, p, isHelp);
   ---------------------------------------------------------
   MarkLt(Node* n, Node* p, bool isHelp)
13 | lC := n·lt;
14 | while true do
15 | | if !IsMark(lC) then
16 | | | m := GetMark(p, lC);
17 | | | if ChCAS(n, lC, m, true) then return lC;
18 | | else return isHelp ? lC·rt : null;
   ---------------------------------------------------------
   MarkRt(Node* n, Node* p, bool isHelp)
19 | rC := n·rt;
20 | while true do
21 | | if IsThMark(rC) then
22 | | | return isHelp ? rC·preLink : null;
23 | | else if IsMark(rC) then
24 | | | return isHelp ? rC·preLink : null;
25 | | else if IsThFlag(rC) then
26 | | | GetRep(n, rC, rC·lt, true);
27 | | else
28 | | | m := IsTh(rC) ? GetThMark(p, rC) : GetMark(p, rC);
29 | | | if ChCAS(n, rC, m, false) then return rC;
```

The REMOVE operation is given by the pseudo-code in the Algorithm 4.2 from the line 1 to 35. We start from the root with its address copying to the variable p. In case we face a link with Mark descriptor, see line 4, we keep on backtracking using the lt of the splice node delegating the link. Please note that the **While** loop does terminate because a child-link of root can not be marked as the key of the root can not be a key to remove. We actually start with attempting our own REMOVE operation where we enter the **While** loop at line 5. First we check, whether the current link has a descriptor Thread or not. It could be a composite descriptor. If the link is not having a Thread or a composite descriptor, we find the child-link of the node (in the direction of the query key) to check whether it is the node to terminate the traversal at. If the child link does not have a Thread or a composite descriptor over it, it can either be a clean link or a link with a Mark descriptor over it.

Algorithm 4.3. Lock-free Internal BST: Help method

```
   GetRepHelp(Node* n, Node* p, Node* oNode)
30 if n ≠ oNode then
31   oLink := oNode·rt;
32   if oLink·rt = n then
33     lC := MarkLt(n, p, isHelp);
34     rC := MarkRt(n, p, isHelp);
35     ChCAS(oNode, oLink, rC, false);
36     return lC;
37   else return n·lt·rt;
38 else return MarkRt(n, p, true);
```

In case there is a Mark descriptor over it, it indicates that the node pointed by c is undergoing a REMOVE and therefore the link p⇝c is attempted (in helping) to be updated to point to a replacement of c. The method GetRepHelp is called to get the replacement node. In case the CAS to update the p⇝c fails, it implies that the link itself has changed and therefore we go to the backtracking steps, see line 10. In all other cases, we check whether the query key match at the current node (represented by c) node, and if it does, the current node becomes the node to remove (stored at d), in case that is not already fixed (be checking whether d is null), and at this location the value of p becomes the parent of the node to remove, see line 13.

Now, if we found that the child-link of c has a Thread or a composite descriptor, it indicates that the traversal is over and it is time to start the REMOVE operation. If the node to remove and its parent were not fixed before this

Algorithm 4.4. Lock-free Internal BST: ADD and CONTAINS operations

```
   ADD(K k)
1  | node := null; p := root·ref;
2  | retry:
3  | c := Child(p, Dir(p, k));
4  | while IsMark(c) do {p := c·lt; c := Child(p, Dir(p, k));} ;
5  | while true do
6  |   if !IsTh(c) then
7  |     n := Child(c, Dir(c, k));
8  |     while IsMark(n) do
9  |       pr := Pr(c); r := GetRepHelp(c, p, pr);
10 |       if !ChCAS(p, c, r, Dir(p, k)) then goto retry;
11 |       c := r; n := Child(c, Dir(c, k));
12 |   else {n := c; c := p;};
13 |   cD := Dir(c, k); pr := Pr(c);
14 |   if c·k == k and pr == null then return false;
15 |   if IsTh(n) then
16 |     if IsThFlag(n) then
17 |       r := GetRep(c, n, ,·lt true);
18 |       if cD == L then ChCAS(p, c, r, Dir(p, k));
19 |       if p == n·rt then p := n·lt;
20 |     else if IsThMark(n) then
21 |       r := GetRepHelp(c, p, pr); ChCAS(p, c, r, Dir(p, c·k));
22 |       if p == c then p := n·lt;
23 |     else
24 |       if node == null then
25 |         node := Node(k);
26 |         node·lt := GetTh(node); node·rt := n;
27 |       if ChCAS(c, n, node, cD) then return true;
28 |     goto retry;
29 |   else {p := c; c := n;};

   CONTAINS(K k)
30 | c := root·lt;
31 | while !IsTh(c) do
32 |   if k == c·k then return Pr(c) == null;
33 |   else c := Child(c, Dir(c, k));
34 | return false;
```

point, it shows that the query key does not exist in the BST and therefore we return false, see line 16. After that we check if the link already has a descriptor `ThFlag`, and if it does, it indicates that a REMOVE of the node has already been initiated and therefore that REMOVE operation is helped before returning false, see line 20. In case the link is found to have the descriptor `ThMark`, it indicates that the order-node of the node is under a concurrent REMOVE and therefore that is helped before going to backtracking step, see line 25.

Now, in case the link is found to have the descriptor `Thread`, it is replaced by a link with descriptor `ThFlag` using a `CAS` and steps to physically clean the node are taken next, see line 28. The method `GetRep` starts with setting the preLink of the node under REMOVE to point to the order-node using a `CAS`, see line 2. At this point if the preLink is found to be already set, it indicates that there is a concurrent helping operation working, and the REMOVE operation being help-aware (see last chapter), returns. If the setting of preLink succeeds, it checks the category of node by comparing the order-node with the node itself, see line 4, and if that found matching it indicates a category 1 node and therefore the next step is to replace the right child-link of the node with a link containing the descriptor `Mark` or `ThMark` whichever appropriate. In case the node is found to be not of category 1, the next step is replace the left child-link of the node with a link containing the descriptor `Mark`.

The steps of replacing the left and right child links with a link containing `Mark` or `ThMark` descriptor, are taken in the methods `MarkLt`, line 13 to 18, and `MarkRt`, line 19 to 29, respectively, in the Algorithm 4.3. In these methods, in case the link is found already containing the desired descriptor, the REMOVE operation returns because of being help-aware. The return is ensured by returning null from the methods which further terminates the calling method. In the method `MarkRt`, in case the right child-link is found containing the descriptor `ThFlag`, it shows that a concurrent REMOVE of the successor of the node has already started and therefore that is first helped before reattempting the `CAS`, see line 25. The method `GetRepHelp`, from line 30 to 38, is called to clean a node whose one of the links has been already replaced with a link with the descriptor `Mark`, and therefore performs the helping steps without attempting setting the preLink.

In an ADD operation, line 1 to 29 in the Algorithm 4.4, the traversal and helping during that are absolutely same as those in a REMOVE operation. When we see a link with the descriptor `Thread`, it is attempted to be replaced with a new node at line 27 using a `CAS`. However, if a node containing the query key is found to be in the BST, whose preLink is still null, false is returned, see line 14.

The CONTAINS operation, line 30 to 34 in the Algorithm 4.4, returns true

only if a node containing the query key is found in the BST and the node must not have its preLink set, see line 32. Otherwise it returns false, line 34.

4.4 Correctness and Complexity

In this section we present the formal proof of correctness of the presented algorithm and show that the algorithm is linearizable to a sequential BST and then prove its lock-freedom. First we give some basic definitions that will be used in the proof.

We consider the shared memory system $\mathcal{U}$ as described in section 2.1. In $\mathcal{U}$ a binary search tree is a data structure that provides a partial implementation $\mathcal{I}_{\mathcal{O}}$, where $\mathcal{O} \subseteq \mathcal{M}$ is given by $\mathcal{O} = \{\text{ADD}, \text{REMOVE}, \text{CONTAINS}\}$. The definition of $\mathcal{M}$ is given in the section 2.1. The state of Υ at time t is a finite set of nodes denoted as $\Upsilon_t = \{x_i(k_i)\}_{i=0}^{r-1}$ forming a directed graph. The nodes are connected with links and the nodes and the links together have properties as described in the section 4.2. There is a specific node root$\in\Upsilon_t$ from where every node $x_i(k_i) \in \Upsilon_t$ is reachable following the lt and rt links of other nodes. It implies that to remove the node $x_i(k_i)$ from Υ_t, we need to ensure that it can not be reached from any other node and hence from the node root.

Definition 4.1. *At time t the BST state Υ_t is called valid iff $\forall$ nodes $x(k_x), y(k_y) \in \Upsilon_t$, $x(k_x) \neq y(k_y) \Rightarrow k_x \neq k_y$. Further if $y(k_y)$ is in the left-subtree of $x(k_x)$ then $k_y < k_x$ and if that is in the right-subtree of the same then $k_y > k_x$.*

At time $t = 0$, the BST state $\Upsilon_0 = \{\text{root}\}$ satisfies the above requirement in which both the subtrees of the node of root are null. A valid BST state $\Upsilon_t = \{x_i(k_i)\}_{i=0}^{r-1}$ corresponds to an abstract set $K = \{k_i\}_{i=0}^{r-1}$ with r elements. The ADT operations ADD(k), REMOVE(k) and CONTAINS(k) follow the sequential specifications as given in their definition in the mapping $\mathcal{M}$ in section 2.1 in chapter 2.

Now we prove some invariants of the presented algorithm in form of lemmas in order to prove that at all time $t \geq 0$ it maintains a valid BST. Essentially, the lemmas will show that no null variable is dereferenced and further if in an execution α, a step s is performed by a thread $p \in \mathcal{P}$ over a configuration $U_t \supseteq \Upsilon_t$ so that it changes to a configuration $U'_t \supseteq \Upsilon'_t$ then $\Upsilon_t \; is \; valid \; \Rightarrow \; \Upsilon'_t \; is \; also \; valid.$ After that we shall prove the linearizability of the set operations to prove a correct concurrent linearizable implementation. And finally we shall prove the lock-freedom and hence we shall prove that the efficient lock-free BST algorithm presented in this paper implements a correct linearizable lock-free concurrent BST.

It is trivial to observe that after a node is ADDed in a BST following our algorithm, its key never changes and all the links outgoing from it are modified using atomic CAS only. Because we can not ADD or REMOVE a key k such that $k \geq (\infty_2)$ or $k \leq (-\infty_0)$, the sentinel node or splice nodes can not come under a REMOVE. For that matter the lt link of root can not be marked.

Lemma 4.1. A null is never dereferenced at any step s during an execution α.

Proof. It can be observed that when a node is ADDed in the BST, all its fields except preLink are non-null. To turn a node to be unreachable the incoming pointers are turned away from it. The preLink field of a node is updated only once in the algorithm in the method GetRep. After that when we need to retrieve the fields of the order-node referred by the preLink, we first check if it is non-null throughout the algorithm. We can check that for the parameters passed to the methods, no null pointer is passed to them. Hence a null pointer is never dereferenced at any step s in α. □

Lemma 4.2. In a BST state Υ_t a link that connects a node to any of its children is not threaded.

Proof. In the initial state Υ_0 it is vacuously true. Now suppose it is true in a state Υ_{i-1} and $\text{ADD}(x)(\Upsilon_{i-1}) \rightarrowtail \Upsilon_i$. At line 27 after CAS succeeds, the link between the node to which x is connected and x, is a clean link. Hence, by induction the lemma proves. □

Corollary 4.1. A threaded link outgoing from a node ensures the subtree of the node in that direction is null.

Proof. We can observe in the operation ADD that when a new node that has been initialized at line 25 its left and right outgoing links are threaded and thus it has null subtrees in both the directions. Using the induction it is proved. □

Corollary 4.2. An ADD always happens at a clean and threaded link.

Proof. Follows from above. □

Lemma 4.3. At any step s in an execution α, a call of CONTAINS(key) over a valid BST state Υ_t at a $t \geq 0$ satisfies the sequential specifications.

Proof. We can observe that in no step s during CONTAINS any change in the BST is performed. The return then matches the sequential specification. □

Lemma 4.4. At any step s in an execution α, a call of ADD(key) over a valid BST state Υ_t at a $t \geq 0$ satisfies the sequential specifications.

Proof. From initial conditions and corollary 4.2. □

Lemma 4.5. At any step s in an execution α, from a valid BST state Υ_t at a $t \geq 0$, the REMOVE of a node $x(k) \in \Upsilon_t$ starts with flagging its order link.

Proof. By observation. □

Lemma 4.6. For a link belonging to a node $x(k) \in \Upsilon_t$, once it is marked it can not be updated again.

Proof. Trivial by observations at the `CAS` operations performed in the algorithm. □

Lemma 4.7. At any step s in an execution α, in a valid BST state Υ_t at a $t \geq 0$, the rt link of a node $x(k) \in \Upsilon_t$ can not be marked unless its order-link is flagged and its preLink must points to the correct order-node.

Proof. rt link of a node is marked only in the method `MarkRt`. However, that method is called from `GetRep` only after setting the preLink and even before that the order link is flagged. □

Lemma 4.8. At any step s in an execution α, before the parent-link incoming to a node $x(k) \in \Upsilon_t$ at a $t \geq 0$, is moved away from it all other links are fixed.

Proof. Using previous lemmas and observation in the operation REMOVE. □

Lemma 4.9. At any step s in an execution α, a call of REMOVE(key) over a valid BST state Υ_t at a $t \geq 0$, satisfies its sequential specifications.

Proof. Using lemma 4.8. □

Using Lemmas 4.3, 4.4 and 4.9, we arrive at proposition 4.1.

Proposition 4.1. *The presented algorithm in an execution α starting with initial configuration $U_0 \supseteq \Upsilon_0$ maintains a valid binary search tree state Υ_t, $\forall t \geq 0$.*

4.4.1 Linearizability

Having proved the above invariants of the lock-free BST algorithm we prove the linearizability.

Lemma 4.10. In a step s in an execution α, there exist linearization points of ADD, REMOVE and CONTAINS between their respective invocation and return, satisfying their sequential specifications

Proof. **CONTAINS** - A thread performing a CONTAINS(k) essentially terminates either at the node where it found the matching key or at a link with `Thread` or composite descriptor. If CONTAINS(k)(Υ_t) = true then the point at which the link pointing to $x(k)$ was read by CONTAINS, is taken as the linearization point. There is no other way that CONTAINS(k)(Υ_t) could return true as proved before. However, if CONTAINS(k)(Υ_t) = false then there could be two possibilities - (a) when CONTAINS(k) was invoked, $x(k) \in \Upsilon_t$ and its preLink was not null and (b) when CONTAINS(k) was invoked, $x(k) \notin \Upsilon_t$. In the first case because the linearization point of CONTAINS(k)(Υ_t) = false is just after the linearization point of the successful concurrent REMOVE operation that set the preLink of $x(k)$. In the second case the linearization point can well be taken as the invocation point of CONTAINS(k) returning false.

ADD - A thread performing an ADD(k) returns false if $x(k) \in \Upsilon_t$ and therefore as in case of CONTAINS, the linearization point of ADD(k)(Υ_t) = false is at the point where the link pointing to $x(k)$ was read, which is between the invocation and return ADD. For a successful ADD operation, the execution of the `CAS` at line 27 is where it takes effect and therefore it is the linearization point of ADD(k)(Υ_t) = true.

REMOVE - A thread performing a REMOVE(k) can return false in two ways (a) $x(k) \notin \Upsilon_t$ at the invocation of REMOVE and therefore is not located (b) $x(k) \in \Upsilon_t$ at the invocation of REMOVE but got REMOVEd by a concurrent successful REMOVE. These are the cases similar to those in the CONTAINS operation and so the linearization points are same as in that case. For a REMOVE(k) returning true, the linearization point is where the preLink of $x(k)$ is set that is at line 2 in the method `GetRep`. Please note that the setting of preLink can be performed either by the REMOVE operation that returns true in account of initiating the operation or by any helping concurrent operation. □

Using the Proposition 4.1 and the Lemma 4.10, the proposition 4.2 follows.

Proposition 4.2. *The algorithm Efficient Lock Free BST implements a valid and linearizable concurrent binary search tree.*

4.4.2 Lock-Freedom

Lemma 4.11. *Lock-freedom is guaranteed in the algorithm Efficient Lock Free BST.*

Proof. By the description of the algorithm, a non-faulty thread performing CONTAINS will always return unless its search path keeps on getting longer forever. If that happens, an infinite number of ADD operations would have successfully completed adding new nodes making the implementation lock-free.

So, it will suffice to prove that the modify operations are lock-free. Suppose that a thread $p \in \mathcal{P}$ performs a modify operation *op* on a valid BST state Υ_t and takes infinite steps and no other modify operation completes after that. Now, if no modify operation completes then Υ_t remains unchanged forcing p to retract every time it wants to execute its own modification step on Υ_t. This is possible only if every time p finds the injection point of *op* having a composite descriptor. This implies that a REMOVE operation is pending. It can be observed in our algorithm that in the function ADD if it gets obstructed by a concurrent REMOVE then before retrying after recovery from failure it helps the pending REMOVE by executing all the remaining steps of that. Additionally, whenever two REMOVE operations obstruct each other, one finishes before the other. It implies that whenever two modify operations obstruct each other one finishes before the other and so Υ_t changes. It is contrary to our assumption. Hence, by contradiction we show that no non-faulty thread shall remain taking infinite steps if no other non-faulty thread is making progress. □

The lemma 4.11 leads to the proposition 4.3.

Proposition 4.3. *The algorithm Efficient Lock Free BST implements a valid and linearizable lock-free concurrent binary search tree.*

4.4.3 Complexity

Having proved that our algorithm guarantees lock-freedom, though we can not compute the worst-case time complexity of an operation, we can definitely derive their amortized complexity. We derive the amortized step complexity of the set operations in our implementation by the accounting method along the similar lines as in [35, 74]. For an execution α, let $\mathcal{O}$ be the set of operations. First we show that a traversal visits a node only a constant number of times and hence we bound the length of the traversal path. Then in the execution α we amortize the step complexity of the operations $op \in \mathcal{O}$.

Lemma 4.12. A thread $p \in \mathcal{P}$ executing an operation *op* visits a node $x \in \Upsilon_t$ only an $O(1)$ times during the traversal in the shared memory system $\mathcal{U}$, if it does no modification.

Proof. Trivial to observe. □

Note that, all the set operations have to perform a predecessor query by a key k to LOCATE an interval $[k_i,\ k_j]$ associated with a link s.t. $x(k_i)$ and $x(k_j)$ are two nodes in the BST. Let us define the *access-node* of an interval as the node from which the link associated with the interval, emanates from. We

define *distance* of an interval from an operation op as the number of links that op traverses from its current location ($\mathsf{root} \; \forall \; t \leq t_i(op)$) to read the access-node of the interval. Suppose that at $t_i(op)$ there are n nodes in the valid BST state $\Upsilon_{t_i(op)}$. Clearly, distance of any interval from op at $t_i(op)$ is $O(H(n))$. Next we prove the following lemma.

Lemma 4.13. If at $t_{ref}(op) \in [t_i(op),\ t_r(op)]$, $x(k_j)$ is a category 1 node and the distance of the interval $[k_i,\ k_j]$ associated with the order-link of $x(k_j)$ from op at $t_{ref}(op)$ is d then to access an interval $[k,\ k'] \subseteq [k_i,\ k_j]$ op traverses no more than $d + ht(x(k_j)) + |\{op' \in \mathcal{O} : op' \; is \; a \; concurrent \; \text{ADD}\}|$ links.

Proof. Given that at $t_{ref}(op) \in [t_i(op),\ t_r(op)]$, $x(k_j)$ is a category 1 node and the distance of the interval $[k_i,\ k_j]$ associated with the order-link of $x(k_j)$ from op at $t_{ref}(op)$ is d. We can observe that if at a $t \in [t_{ref}(op),\ t_r(op)]$, $x(k_j)$ is still a category 1 node and it is REMOVEd then the interval associated with its order-link gets subsumed by the interval associated with the order-link of the leftmost child in its right subtree or with the order-link emanating from its parent if the right subtree is null. In the former case the distance of $[k_i,\ k_j]$ from op becomes $d + ht(x(k_j))$ and in the latter it decreases by 1. Also if a node $x(k_l)$ is ADDed by an operation op' then the extra distance apart from d traversed by op to access $[k_i,\ k_l]$ or $[k_l,\ k_j]$ is no more than 1. The observations made above imply that op does not traverse more than $d + ht(x(k_j)) + |\{op' \in \mathcal{O} : op' \; is \; a \; concurrent \; \text{ADD}\}|$ links to access a subinterval of $[k_i,\ k_j]$. □

Lemma 4.14. Length of the traversal path of a thread $p \in \mathcal{P}$ is bounded by $2H(n) + |\{op' \in \mathcal{O} : op' \; is \; a \; concurrent \; \text{ADD}\}|$.

Proof. When op traverses in the left subtree of a category 3 node x and if x gets REMOVEd, the interval associated with its order-link gets subsumed by the interval associated with the order-link of the leftmost node in the right subtree of x which is a category 1 node. On removal of a category 2 node, the interval associated with its order-link is subsumed by the interval associated with the order-link of its parent which can be a category 2 or category 3 node. Lemma 4.12 shows that a node is visited at maximum $O(1)$ times by a thread during a traversal. And, $(d + ht(x)) \leq 2H(n) \; \forall \; x \in \Upsilon$ and for any d that is distance of an interval from op its present position. Therefore, lemma 4.13 along with these observations show that the path length of a traversal in our lock-free BST is bounded by $2H(n) + |\{op' \in \mathcal{O} : op' \; is \; a \; concurrent \; \text{ADD}\}| + O(1)$. □

Having shown that the traversal path of a thread for any operation is bounded by $O(H(n) + |\{op' \in \mathcal{O} : op' \; is \; a \; concurrent \; \text{ADD}\}|)$ we prove that an obstructed operation incurs only a constant number of extra steps in helping an obstructing operation.

Lemma 4.15. An obstructing operation op makes an obstructed operation op' take only a constant number of extra steps for recovery from failure in order to finish its execution.

Proof. An ADD operation does not have to hold any link and so does not obstruct an operation for itself so only a REMOVE operation can obstruct another operation. Let op be a REMOVE operation. We observe that after flagging the order-link of a node, op takes only a constant number of atomic steps to flag, mark, tag and swap links connected to the node and to its order-node in addition to setting the preLink pointer of the node under REMOVE, if not obstructed by a concurrent operation. To start helping op after an unsuccessful CAS in order to complete an operation op', a thread p reads the preLink pointer of a node. It is trivial to observe that from a node pointed by preLink the distance of node is no more than a single directed link. That shows that a thread needs to take only a constant number of extra steps in order to perform helping. Hence the recovery from failure due to a concurrent obstructing operation needs only a constant number of links to traverse. That proves the lemma. □

Now we amortize the step complexities of the operations during an execution α. In the shared memory system $\mathcal{U}$, let $t_i(op)$ be the *invocation point* of op which is the time it reads the $pRoot$, and $t_r(op)$ be the *return point* of op which is the time it reads or writes at the last link before it leaves the BST. The *point contention* $c_p(op)$ during the execution interval of op is defined as the maximum number of threads running concurrently at any point $t \in [t_i(op),\ t_r(op)]$ [3] to execute any operation. Some authors also call it *concurrent contention* [52]. In order to perform an operation $op \in \mathcal{O}$, a number of atomic steps a are taken. The amortized complexity of $op \in \mathcal{O}$ is computed as following

$$
\begin{aligned}
& Amortized\ step\ complexity\ \hat{c}(op)\ of\ op \in \mathcal{O} \\
= \ & Actual\ step\ complexity\ c(op) \\
& + number\ of\ extra\ steps\ charged\ to\ op\ on\ behalf\ of\ op' \\
& - number\ of\ extra\ steps\ charged\ on\ behalf\ of\ op\ to\ op'' \\
& where\ op, op', op'' \in \mathcal{O}\ and\ op' \neq op \neq op''
\end{aligned}
$$

Let $\mathcal{A}$ be the set of atomic steps taken by all $p \in \mathcal{P}$. We define a function $f : \mathcal{A} \mapsto \mathcal{O}$ such that if $f(a) = op$ then a is charged to the account of op and

(a) In case of no contention, all the atomic steps a representing atomic read, write and CAS taken by op is mapped to op by f.

(b) In case of contention, any failed CAS by op is mapped by f to the operation op' whose successful CAS causes the failure.

(c) If an extra read is performed during the traversal in op due to an ADDed node at $t \in [t_i(op),\ t_r(op)]$ to the set of existing nodes by a concurrent ADD operation op' then it is mapped by f to op'.

(d) Any read, write or CAS step a taken by an operation op after the first failed CAS and before retrying at the same link i.e. during helping and recovery from failure is mapped by f to the operation op' that performed the successful CAS in order to make op help it, provided op' further does not help some op'' so that op helps op'' recursively. This includes resetting of prelink, if needed.

(e) In case of recursive helping, the extra atomic steps by all the operations helping op is mapped by f to op.

With the definition of the function for accounting the steps, we prove that the upper bound of amortized complexity of operations in the following Proposition.

Proposition 4.4. *In a BST with n nodes at the start of a finite execution α, the amortized step complexity of each operation op in α is $O(H(n) + c_p(op))$, where $c_p(op)$ is the point contention during the execution of op.*

Proof. Clearly for two operations $op, op' \in \mathcal{O}$ *and* $op \neq op'$, if they are not concurrent, f can not charge any extra step to op or op' on behalf of either. Now we take the three set operations separately.

(a) A CONTAINS operation op, as it does no modification in the BST no extra step can be charged to it other than the essential steps that it would take on account of the traversal. If a node is ADDed by op' to the BST at $t \in [t_i(op),\ t_r(op)]$ and it comes in the traversal path of op then the read of this node is charged to op' by f. In Lemma 4.12 we proved that a traversal visits a node only $O(1)$ times. From the discussion in the proof of Lemma 4.12 it can be seen that during the traversal an operation may possibly visit the node p more than once if the node $curr$ is shifted by a concurrent REMOVE operation during its LOCATE. The extra read of p is charged to the concurrent REMOVE operation that shifts $curr$ and that is at most once as in a REMOVE the shift of an order-node happens only once. Because the traversal path length is at most $2H(n) + |\{op' \in \mathcal{O} : op' \ is\ a\ concurrent\ \text{ADD}\}|$ and the number of extra reads, if any, counted in $|\{op' \in \mathcal{O} : op' \ is\ a\ concurrent\ \text{ADD}\}|$ is charged to concurrent ADD operations, the amortized step complexity of a CONTAINS operation during a finite execution α is $O(H(n))$ where n is the number of nodes in the BST in the initial configuration in α.

(b) An ADD operation does not perform any flag or mark of a link which can block a concurrent operation so an extra step can not be charged to an ADD by f on account of getting helped by any concurrent operation. When a node is ADDed to the BST by an operation op which was not at the invocation point of a concurrent operation op' and the new node comes in the path of the traversal of op', the read of the new node by op' is charged to op and it can be at most 1. This infers that at most $c_p(op)$ can be charged to an ADD operation op by f on account of the added node. An ADD operation's successful CAS can cause failure to the CAS of a concurrent ADD or the flagging or marking of a concurrent REMOVE and for that at most 1 CAS step can be charged by any concurrent modify operation. After the failure any concurrent modify will have to travel only one extra link which has been counted before. So on the account of failed CAS of concurrent operations f can charge at most $c_p(op)$ to an ADD operation op. Finally, an ADD operation helps a concurrent REMOVE operation if the threaded link that the new node is required to be added to is found marked or flagged. The failed CAS step and the extra steps in helping the concurrent REMOVE is charged to that. By Lemma 4.15 only a constant number of extra steps are taken in the helping. On summarizing these observations and by the upper bound of the traversal path by Lemma 4.12, the amortized complexity of an ADD operation op during a finite execution α is $O(H(n) + c_p(op))$ where n is the number of nodes in the BST in the initial configuration in α.

(c) For a REMOVE operation other than the essential steps that it takes in the traversal to track the order-link of the node to be deleted, it obstructs a concurrent modify operation and can make a concurrent traversal read extra node because of shifting of the order-node. The steps charged on account of traversal is similar to as discussed in case of CONTAINS. By Lemma 4.15 only a constant number of extra steps are needed by an obstructed concurrent modify operation. Therefore after locating the order-link of the node and its successful flagging a REMOVE operation op can be charged only $O(c_p(op))$ extra steps if it itself is not obstructed by another concurrent REMOVE forcing a recursive helping. In case of recursive helping the charges of the extra steps on behalf of obstructed operations that is taken to help another obstructing operation is passed to that by f. Summarizing these observations the amortized complexity of a REMOVE operation op during a finite execution α is $O(H(n) + c_p(op))$ where n is the number of nodes in the BST in the initial configuration in α.

Summing up, in any finite execution α with the set of operations $\mathcal{O}$, threads perform at most $O\left(\sum_{op\in\mathcal{O}}(H(n)+c_p(op))\right)$ steps in total where n is the number of nodes in the BST in the initial configuration in α. □

It is straightforward to observe that the number of memory-words used by a BST with n nodes in our design is $5n$. That concludes the amortized analysis of our algorithm.

Chapter Summary

In this chapter we presented an algorithm for a lock-free internal BST. We proved that each operation in our design takes $O(H(n)+c)$ amortized number of steps, where $H(n)$ is the height of the BST with n nodes and c is the point contention. We solved the problem of "retry from scratch" in internal BSTs. We proved the linearizability and lock-freedom of the proposed algorithm.

Acknowledgments

We would like to thank Dr. Neeraj Mittal (Associate Professor, Department of Computer Science, The University of Texas at Dallas, USA) for many valuable comments on a previous version of this chapter published as a a paper in PODC 2014.

Part III

LOCK-FREE 1-DIMENSIONAL RANGE SEARCH

5

LOCK-FREE LINEARIZABLE 1-DIMENSIONAL RANGE QUERIES

Chapter Abstract

Efficient concurrent data structures that support range queries are highly sought-after in a number of applications. For example, the contemporary big-data processing platforms employ them as in-memory index structures for fast and scalable real-time updates and analytics, where analytics utilizes the range queries.

In this chapter, we present a generic algorithm to perform linearizable range queries in lock-free ordered 1-dimensional data-structures. The algorithm requires single-word `CAS` *primitives. Our method generalizes the lock-free data structure snapshot of Petrank et al. [76]. Fundamentally, we utilize a partial snapshot object derived from the snapshot object of Jayanti [58].*

We experimentally evaluate the proposed algorithm in a lock-free linked-list, skip-list and binary search tree (BST). The experiments demonstrate that our algorithm is scalable even in the presence of high percentage of concurrent modify operations and outperforms an existing range search algorithm in lock-free k-ary trees in several scenarios.

5.1 Introduction

5.1.1 Background

An ordered set abstract data type (ADT), which supports set operations - ADD, REMOVE, CONTAINS and RANGESEARCH, is a widely used programming interface. Sequential data structures such as linked-lists, skip-lists and BSTs, which provide predecessor queries, implement this ADT. With the ubiquity of multi-core processors, efficient concurrent version of these data structures are ever more important.

An extremely important application area for such concurrent data structures is a modern big-data processing platform such as Google's Bigtable [22] and Apache HBase [47]. These technologies, in addition to the on-disc storage for persistence, employ a concurrent data structure as an in-memory index for fast real-time data updates and analytics. Specifically, the analytics components thereof are based on the RANGESEARCH operations. Therefore, to ensure the correctness of the analytics, the consistency of a RANGESEARCH, in presence of other concurrent set operations, becomes paramount. Additionally, the overall performance of such applications get significant impact from the scalability of the utilized concurrent index structure.

The most common consistency framework used in concurrent settings is linearizability [50], which makes a user perceive an operation to take effect instantaneously at a point between the invocation and response of the operation. In effect, linearizability of range search is highly desired because it ensures that the output provides an "aligned view", with respect to a global real-time order, of the targeted entries in the data structure.

Traditionally, concurrent data structure use mutual exclusion locks for consistent set operations. However, locks succumb to high contention and often offer poor scalability. In addition to that, in an asynchronous shared-memory system, where an infinite delay or a crash failure of a process is possible, a lock-based concurrent data structure is vulnerable to pitfalls such as deadlock, priority inversion and convoying. On the other hand, in a lock-free data structure, processes do not hold locks and at least one non-faulty process is guaranteed to finish its operation in a finite number of steps. Therefore, lock-free data structures foster both scalability and fault tolerance.

A number of lock-free ordered data structures exist in the literature: singly-linked lists [46, 35], doubly-linked lists [82], skip lists [81, 62], BSTs [33, 53, 71, 24, 32], etc. However, in general, the ADT implemented by these data structures do not support RANGESEARCH operations. A linearizable RANGESEARCH outputs a consistent view of the concurrent data structure with respect to multiple points stored in it and thus is inherently different from other set operations that essentially require single point queries.

This chapter makes the following contributions:

1. We present a generic method to implement linearizable RANGESEARCH operations in a lock-free ordered search data structure that supports linearizable ADD, REMOVE and CONTAINS operations.

2. Our method can be seamlessly integrated to any lock-free 1-dimensional ordered data structure.

3. We experimentally show that the proposed range search algorithm achieves good scalability and outperforms an existing range search method [16] in high contention scenarios.

Further in this chapter, first, we describe the algorithm (section 5.2). Thereafter, we present a proof of linearizability of the introduced range-search algorithm (section 5.3). Finally, we present details of the experiments and discussion on the observations thereof (section 5.4).

5.1.2 Related work

To our knowledge, the only work existing in the literature with regard to a linearizable range query in a lock-free data structure is by Brown et al. [16], who implemented it in the lock-free k-ary search trees. Their method requires range scanning using a depth-first-search followed by validating the scan, and if any node is found outdated, the range scan is retried, often repeatedly. Thus, in the cases of multiple concurrent updates, this method lets the range search starve.

The `ConcurrentSkipListSet` in Java concurrency library supports `subSet` operations that are effectively non-linarizable range queries. Avni et al. [10] used software transactional memory (STM) to encapsulate the linearizable range queries in a lock-based concurrent skip list, which they call a Leap list. The scalability of their method mainly depends on the efficiency of the underlying STM. Moreover, an STM based approach for a range query incurs increasing overhead with the growth in the number of target shared-memory words covered by a transaction and thus may not be an efficient approach to perform queries for long ranges in a reasonably populated concurrent data structure. Sagonas et al. [79], proposed to implement linearizable range queries by way of locking each of the nodes, which contain the points in the target range, in a lock-based data structure. Recently, Basin et al. [12] published a concurrent key-value store that supports range queries.

5.1.3 A summary of our work

(A) Background

A simple approach to perform a linearizable range search in a lock-free data structure is to collect a lock-free linearizable snapshot of the data structure and output the appropriate subset thereof. However, not many lock-free data structures support linearizable snapshot collection. Exception is - lock-free hash trie by Prokopec et al. [77]. They used a variant of double-compare-single-swap

(DCSS) primitive, which is implemented using single-word CAS. Here, an update with a concurrent snapshot requires copying each updated node together with the path from the root to it to help the snapshot collection, which results to a performance deterioration of concurrent updates with snapshot. Also, the hash mapping of the keys makes it hard to compute a range search directly from a collected snapshot.

Petrank and Timmat [76] presented another approach for collecting a linearizable snapshot of lock-free linked lists and skip lists. Their method is based on the wait-free multi-writer multi-scanner *snapshot object* of Jayanti [58]. A snapshot object, which consists of multiple shared-memory words, supports concurrent `scan` and `update` methods. An `update` writes a new value at one of the words, and a `scan` returns an atomic view of all the words.

Nonetheless, directly using a "full" snapshot collection for a RANGESEARCH is undesirable as it discards the advantage of the size of a range query output, which may not necessarily be equal to the size of the entire data structure.

A way to align the snapshot collection with the size of the queried range can be derived from the notion of the *partial snapshot* introduced by Attiya et al. [9]. A partial snapshot collection is essentially a generalization of the snapshot collection, in which the target is restricted to only a part of the multi-word object. This work underlines the motivation that a partial snapshot collection must be cheaper than a (full) snapshot collection.

The fundamental idea of a partial snapshot is based on the multiple concurrent announcements of *partial* `scan` operations. To handle the announcements, Attiya et al. used the idea of an *active set object* of Afek et al. [3]. An active set provides the methods - `join`, `leave` and `get_set`. A `join` method is called to join the set of "active" processes, a `leave` is called to leave such a set, and `get_set` outputs the list of processes which are part of the active set. Thus, an active set implementation keeps track of the processes that "currently" perform a partial `scan`.

(B) Our approach

To implement a lineaeizable range search, we combine the method of lock-free data structure snapshot of [76] with an implementation of an active set.

The key element of the lock-free data structure snapshot of [76] is a special object called *snap-collector*. The data structure is augmented with a pointer to snap-collector allocated by a process (called a scanner) collecting a snapshot. Concurrent scanners use a single snap-collector to return the same snapshot. An operation such as ADD, REMOVE or CONTAINS *reports* a modification to a concurrent scanner using its snap-collector.

We modify a snap-collector by equipping it with the lower and upper limits of a target range and call the modified object a *range-collector*. However, it does not provide a scalable solution for a common scenario in which concurrent range queries target disjoint ranges and thus can not use the same range-collector. Naturally, to solve that issue, we need multiple "simultaneously active" range-collectors corresponding to the concurrent range queries targeting disjoint ranges. Yet, there can be scenario where the concurrent range queries have overlapping ranges and for them ensuring linearizability needs extra care. We explain it below.

Suppose op_1 and op_2 are two concurrent RANGESEARCH operations with overlapping but non-identical target ranges. Now, if op_1 has its linearization point before that of op_2, and it includes a data-point k in its output, which belongs to the intersection of the associated target ranges, and there is no linearization point of a REMOVE(k) between the linearization points of op_1 and op_2, op_2 must also include k in its output for consistency. That effectively means that op_1 and op_2 must synchronize before their returns. Furthermore, if for concurrent and independent progress we use different range-collectors for op_1 and op_2, a concurrent ADD/ REMOVE/ CONTAINS operation, say op_3, will have to report to both the range-collectors, if its return point happens to belong to the intersection of the two associated ranges. Naturally, it significantly increases the overhead of reporting, which is undesirable.

To overcome these issues, we augment a lock-free concurrent data structure with a lock-free linked-list of range-collector objects. Functionally, the linked-list implements an active set of processes performing concurrent range queries. The data structure holds a pointer to one of the ends of this list, say the *head*. A new range-collector is allowed to be added only at the other end, say the *tail*. Thus, the addition to, removal from and traversal through this lock-free list delegate the active set methods `join`, `leave` and `get_set`, respectively.

A RANGESEARCH operation starts with traversing through the list and checks whether there is an *active* range-collector with an identical target range. If in the list such an active range-collector is found, it is used for a concurrent *coordinated* range scan, which is analogous to using same snap-collector by multiple concurrent scanners in [76]. A range-collector is removed from the lock-free linked-list as soon as the range scan at it gets over. With that, if concurrent RANGESEARCH operations op_1 and op_2 have overlapping ranges, and op_1 successfully joins the active set before op_2 by adding its range-collector to the lock-free list, we first make op_2 help op_1 to finish, and then let it restart. Thus at any time-point, the augmented list contains active range-collectors with disjoint ranges only.

It is easy to see that in our method a RANGESEARCH does not require

multiple (repeated) scans followed by validation, and thus, presents a scalable method even in presence of high percentage of concurrent ADD/ REMOVE operations in the data structure.

Please note that, similar to [76], our approach to implement range search in a lock-free data-structure is independent of the actual data structure design, and thus is generic. Moreover, this work also practically presents an alternate method to implement a partial snapshot object derived from [58]. The existing algorithms for partial snapshot by Attiya et al. [9] and Imbs et al. [55] are based on repeated scan and validation that we aimed to avoid.

Lastly, because the range search in data structures storing multidimensional points is not similar to that in 1-dimensional data structures, for detail see Samet's book [80], we do not claim that our method can be directly adapted to multidimensional range search problems in a concurrent setting, which in its own merit is a very important but yet largely unexplored topic.

5.2 The Lock-Free Range Search

We consider an *asynchronous shared memory system* $\mathcal{U}$ which comprises a set of word-sized *objects* $\mathcal{V}$ and a finite set of processes $\mathcal{P}$ and supports *primitives* `read`, `write` and and `CAS` (compare-and-swap). $\mathcal{U}$ guarantees that the primitives are *atomic* i.e. they take effect instantaneously at an indivisible time-point [51]. Each object $v \in \mathcal{V}$ has a unique *address*, commonly known as a *pointer to* v, denoted by $v{\cdot}ref$. `CAS`($v{\cdot}ref$, exp, new) compares the value of v with exp and on a match updates it to new in a single atomic step and returns true; else it returns false without any update at v. Let $|\mathcal{P}|=n$. Processes $p_i \in \mathcal{P}$ communicate by accessing the objects $v \in \mathcal{V}$ using a primitive.

We consider an abstract data type (ADT) OSet that provides definition of operations ADD, REMOVE, CONTAINS and RANGESEARCH. The ADT OSet is implemented by an ordered 1-dimensional data structure. A formal definition of the ADT OSet is as given in the chapter 2.

Fundamentally, our algorithm presents a generalization of the lock-free iterator algorithm of [76]. Therefore, for the sake of completeness, we briefly recap the algorithm of [76] in the section 5.2.1 before describing the linearizable range query algorithm in the section 5.2.2.

5.2.1 Snap-collector implementation

The lock-free data-structures considered in [76] are - linked list [46] and skip list [62], in which each node x has a unique key $key(x)$ and it is straightfor-

ward to find $next(x)$. The data-structures implement an ADT that provides the operations ADD(x), REMOVE(x) and CONTAINS(x). Without any ambiguity, we denote ADD$(key(x))$ by ADD(x). In these data-structures, the operation REMOVE(x) takes more than one atomic CAS steps. One of the CAS steps, whose success ensures that x will be eventually removed, is generally known as *logical remove*, and is considered the linearization point of REMOVE(x).

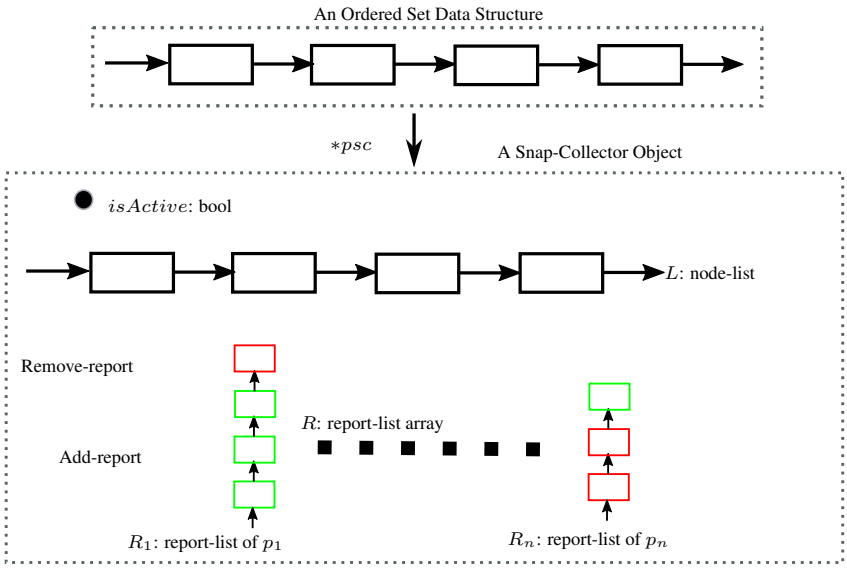


Fig. 5.1: A snap-collector implementation

To implement the snapshot collection, a pointer psc to an instance of a class *snap-collector* is maintained in the data-structure. The structure of the class snap-collector is shown in the Figure 5.1. It holds (a) a boolean variable $isActive$ (b) a list L of the data structure node pointers in which nodes are sorted by their keys and (c) an array R of lists of *reports*. A report comprises a node pointer and a *report-type* (ADD or REM). The size of the report-list array R is equal to the number of processes in the system. A snap-collector with false $isActive$ field is called *inactive*, otherwise *active*. If there is no ongoing snapshot collection, the pointer psc holds a reference to an inactive snap-collector. Initially psc points to a dummy snap-collector which is always inactive.

A snapshot collection starts with reading the pointer psc. If psc is found pointing to an inactive snap-collector, a new active snap-collector is allocated. The snap collector attempts to update psc to point to the new snap-collector

Algorithm 5.1. The basic methods of the range query algorithm

```
   ▷ data-structure-node.
35 struct DSNode {K k; ···;} *DSNPtr;

   ▷ report consisting a DSNPtr and report-type.
   struct Report {▷ RptType: ADD, REM.
36   RptType rt; DSNPtr node;
37 } *RptPtr;

   ▷ packet of a DSNPtr and auxilliary info to find next node.
38 struct Window {DSNPtr cur;···;};

39 struct RCNode {▷ range-collector-node.
40   K lo, hi; Window * L;
41   RptPtr * R;▷ Array-Size=#Threads.
     ▷ mark bit of next is used.
42   (RCNPtr ref, bool mark) next;
43 } *RCNPtr;

   ▷ Global shared variables to initialize.
44 RTail = RCNode(∞, ∞, null, null, (null, 1));
45 RHead = RCNode(-∞, -∞, null, null, (*RTail, 1));
46 ···;▷ Other global variables.

   ▷ Returns the Window of DSNPtr with key just ≥ lo; returns null
   if no such node present.
47 FindMinNode(K lo)

   ▷ Returns a Window of DSNPtr with key just ≥ key(x →cur).
48 Next(Window x)

   ▷ Returns true if x is not logically removed.
49 IsPresent(DSNPtr x)

   ▷ Returns true if range-collection at rnd is not linearized
   i.e. its next field's mark is 0.
50 IsActive(RCNode rnd)

   ▷ Returns the Window of DSNPtr with largest key in rnd·L
   after attempting to add x to it.
51 AddNode(RCNode rnd, Window x)
```

using a CAS. If the CAS fails, it helps the concurrent operation that successfully injected its own snap-collector. In both the situations (success in injecting own snap-collector or helping), the process scans the data-structure and collects the address of every node x that it discovers which is not logically removed. The list L (which has very similar semantics as the lock-free queue of Micheal et al. [66]) is used for the purpose in which the node $tail(L)$ satisfies $key(l) \leq key(tail(L))$ for every node $l \in L$.

Algorithm 5.1. The basic methods of the range query algorithm

```
▷ Returns whether the interval [lo,hi]
isConatined/contains/overlaps/isDisjoint with the interval of the
range-collector-node rnd.
52 ChkOvrlap(K lo, K hi, RCNode rnd)
---
▷ Returns a list of keys in interval [lo,hi] from the
range-collector-node rnd.
53 ProcessRange(K lo, K hi, RCNode rnd)
---
▷ Returns true if range-collection at rnd is not linearized
i.e.its next field's mark is 0.
54 Report(RCNode rnd, Report r, int tid)
---
▷ Collects the nodes in [lo,hi] to add to rnd·L.
55 SnapRange(K lo, K hi, RCNode rnd)
56   w = FindMinNode(lo);
57   if w·cur≠null then
58     while IsActive(rnd) and w·cur.key≤hi do
59       if IsPresent(w·cur) then w = Next(AddNode(rnd, w));
60   Deactivate(rnd); BlockFurtherReports(rnd);
---
▷ Adds a Window of DSNPtr with max-value key to rnd·L and
sets 1 at the mark of rnd's next.
   Deactivate(RCNode rnd)
61   AddNode(rnd, Window(*DSNode(∞),···));
62   (ref, m) = rnd·next;
63   while m≠1 or !CAS(rnd·next, (ref, 0), (ref, 1)) do
64     (ref, m) = rnd·next;
```

To optimize the scanning by multiple concurrent processes, a process is

allowed to insert a node-pointer ${}^*x$ in L only if $key(x) > key(tail(L))$. An attempt to insert ${}^*x$ in L returns $tail(L)$ if $key(x) \leq key(tail(L))$ (without making any changes in L) and ${}^*x$ is returned if it was successfully added to L. The $isActive$ variable is checked before every attempt to read a node from the data-structure and if it is found false then the snapshot collection is guaranteed to have finished. On completion of scanning the data-structure, $isActive$ is set false. This is the linearization point of the snapshot. Further, in order to ensure that all the processes have the same view of L, before attempting to set false at $isActive$, a null is inserted at L.

For a linearizable snapshot collection, a process performing ADD/ REMOVE/ CONTAINS requires to report the operation. A REMOVE(x), on finding x, performs steps up to the logical remove of x, then before taking further steps, adds a REM report consisting ${}^*x$ to its respective report-list in R. An ADD(x) after adding x with a successful CAS, reports ${}^*x$ as an ADD to its respective report-list in an active snap-collector. Before every ADD report, the logically remove status of ${}^*x$ is checked to avoid an unnecessary reporting. A CONTAINS(x) on finding a node reports (ADD or REM) to an active snap-collector. To ensure a consistent view of the report-lists of all the processes in the system, before the $isActive$ field of a snap-collector is set false, a null report is added to each of the report-lists. After the linearization of the snapshot, the reports are processed by sorting them and then merging them to L. A processed snapshot contains address to the nodes present in the data structure during the execution interval of the snapshot collection. The concurrent processes working at the same snap-collector return same snapshot.

5.2.2 Lock-free linearizable range search algorithm

Having presented the basic construction of the snapshot collection method, we are now prepared to present the linearizable range search by means of an active set of processes that collect concurrent snapshots targeting different subsets of nodes present in a lock-free data-structure.

In the description of the algorithm, we assume that the memory allocator always allocates a new shared-memory object at a new address and thus ABA problem (see the chapter 2) never occurs. We also assume that the memory reclamation is lock-free such as one using a lock-free garbage collector, for example, [72]. These are standard assumptions in the description of a lock-free data structure algorithm.

The pseudo-code of the algorithm is given in the Algorithms 5.1 to 5.3. We have sufficiently commented the pseudo-code to describe the functionality of the methods.

Algorithm 5.2. The linearizable range search algorithm

```
▷ Returns a list of keys in interval [lo,hi].
RANGESEARCH(K lo, K hi)
1  pre=RHead; cur=pre→next; mode=INIT; RN=null;
2  retry:
3  while true do
4      (ref, m)=cur→next;
5      while ref ≠ null and m==1 do
6          if !CAS(pre→next, (cur, 0), (ref, 0)) then
7              goto retry;
8          (ref, m)=cur→next; pre=cur; cur=ref;
9      if ref==null then
10         if mode==CLEAN then
11             return ProcessRange(lo, hi, RN);
12         RN=RCNode(true, lo, hi, null, null, (*RTail, 0));
13         if CAS(pre→next, (cur, 0), (*RN, 0)) then
14             SnapRange(lo, hi, RN);
15             (ref, ∗)=RN→next; mode=CLEAN;
16             if CAS(pre→next, (RN, 0), (ref, 0)) then
17                 return ProcessRange(lo, hi, RN);
18         else goto retry;
19     else if ChkOvrlap(lo, hi, cur)==isConatined then
20         SnapRange(cur·lo, cur·hi, cur);
21         RN=cur; mode=CLEAN; (ref, ∗)=RN→next;
22         if CAS(pre→next, (RN, 0), (ref, 0)) then
23             return ProcessRange(lo, hi, RN);
24     else if ChkOvrlap(lo, hi, cur)==isDisjoint then
25         (ref, m)=cur→next; pre=cur; cur=ref;
26     else
27         SnapRange(cur·lo, cur·hi, cur);
28         (ref, ∗)=cur→next;
29         CAS(pre→next, (cur, 0), (ref, 0));
```

We consider an ordered lock-free data-structure with nodes of type DSNode which are indexed by the keys k uniquely selected from a partially ordered universe. As before, we shall use k to denote a node if its key is k.

The data-structure is augmented with a lock-free linked list of instances of the class RCNode (range-collector node), see line 39 to line 43 in the Algorithm 5.1. An RCNode, similar to a snap-collector node, in addition to containing the data structure node list L and array of report-lists R, contains a pointer $next$ to connect to another RCNode in the list. It also records the range in terms of the lower limit lo and the upper limit hi. However, unlike a snap-collector node, a range-collector node does not contain the boolean field $isActive$ to fix the linearization point. Here we use a bit from the pointer $next$ for the purpose, which is a design optimization.

An important difference between an RCNode and a snap-collector node of [76] is the list L: we store additional info in a node (of type **Window**, line 38) of L that can facilitate traversal in the data-structures like BST, in which, unlike linked list and skip list, computing $next(x)$ is not straightforward. We describe in section 5.2.3 how we implemented the method `Next` that finds the successor of a node in a BST.

The list of RCNodes, which is unordered, has similar semantics as the lock-free linked list of Harris [46], except that here a new RCNode can be added only at one end. See Figure 5.2. The linked list is represented by two sentinel RCNodes $RHead$ and $RTail$ with ranges [-∞ -∞] and [∞ ∞], respectively, s.t. they are disjoint to any RCNode (-∞ and ∞ are the minimum and the maximum elements, respectively, in the partially ordered universe of keys). At the initialization of the data-structure, the $next$ of $RHead$ points to $RTail$ and L and R of both these RCNodes are null. Additionally, the $next$ of $RTail$ is always null.

A new RCNode is added using a `CAS` such that its $next$ pointer always points to $RTail$. On a `CAS` failure we restart from $RHead$. To remove an RCNode from the linked list, first its $next$ pointer is marked (one unused bit is set using a `CAS`) and then the $next$ pointer of the previous RCNode is updated. Any traversal through the list always helps a pending remove in the path. The method RANGESEARCH of the data-structure (see line 24 to line 23 in Algorithm 5.2) intrinsically uses add and remove of an RCNode in the unordered list together with the traversal through the list.

In effect, at any point of time, all the processes that perform RANGESEARCH use exactly one RCNode whose $next$ is not marked. A traversal through the list outputs the process-id of all the processes that perform RANGESEARCH. Because a new RCNode is added only at the $RTail$ (just before it), a traversal can not miss a new RANGESEARCH that started before and has not linearized

Algorithm 5.2. The linearizable range search algorithm: The operation `SyncWithRangeQuery`

```
▷ Finds appropriate active concurrent range-collection to
report node x.
SyncWithRangeQuery(DSNPtr x, RptType report, int tid)
30  (ref, *)=RHead→next;
31  while ref≠RTail do
32      if IsActive(ref) and ref.lo≤x.key≤ref.lo then
33          if report == ADD and IsPresent(x) then
34              Report(ref, Report(x, ADD), tid);
35          else Report(ref, Report(x, REM), tid) ;
36          break;
37      else (ref, *)=ref→next;
```

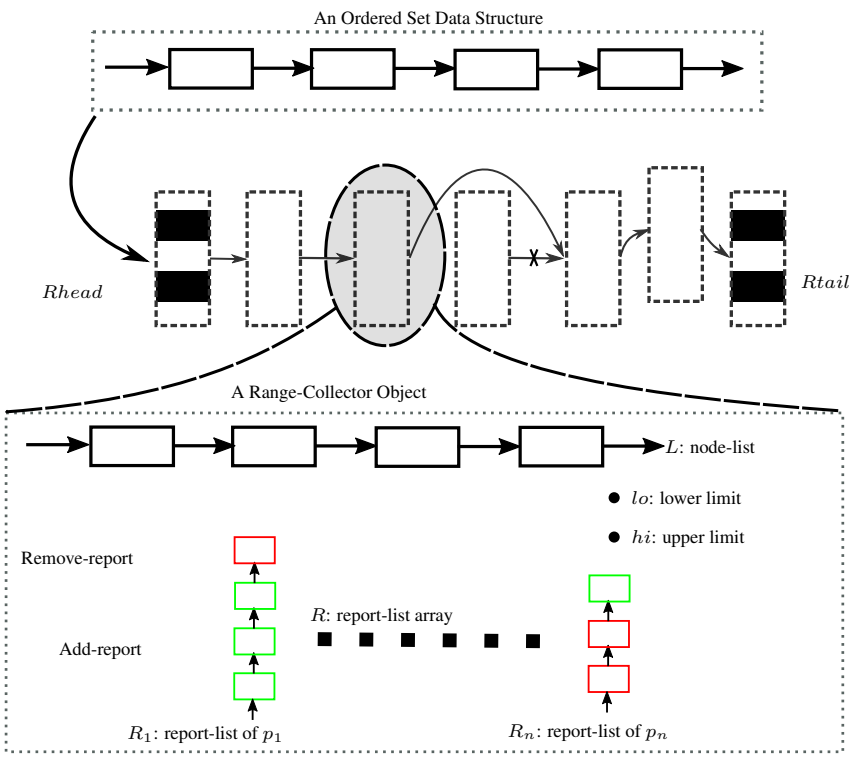


Fig. 5.2: A range-collector implementation

yet. Therefore, a process that starts a new RANGESEARCH always gets to know the concurrent processes that are active and performing RANGESEARCH. Also, "logically" leaving this set of active processes takes one CAS execution. Thus, the augmented lock-free list implements an active set.

A process intending to perform RANGESEARCH($[lo\ hi]$), line 24 to line 23 in the Algorithm 5.2, starts scanning the unordered list starting from $RHead$. A variable $mode$ is assigned value INIT to indicate that the process is yet to start the collection of desired data-structure nodes, line 1. On reading an RCNode cur with range $[x\ y]$, one of the following is done depending on how $[lo\ hi]$ relates to $[x\ y]$:

1. If the $next$ of cur is marked and not null (line 5), it indicates a pending but linearized RANGESEARCH operation. We help to clean the RCNode cur from the linked list.

2. If the $next$ of cur is null (line 9), i.e. cur is the node $RTail$, a new RCNode RN is allocated and added. After that, the method SnapRange (line 55 to line 60) is called to collect the snapshot of the desired subset of the data-structure.

   In an execution of SnapRange, the collection of nodes from the data-structure and storing them in the list L works exactly the same way as in a snap-collector. We begin with finding the node with smallest key in the data structure using the method FindMinNode, at line 47. In linear data structures such as linked list and skip list, it is straightforward: the first node (head or the successor of it in case head is a dummy node) is the node with the smallest key. In a BST we need to traverse down to locate the node with smallest key. Similarly, to find the node in the data structure with successor key the method Next, line 48, works directly in linear data structure, whereas, in a BST, we need to perform depth first search. The methods FindMinNode and Next for a lock-free BST are described in the section 5.2.3.

   The method Deactivate (line 60 to line 64), which consists of the steps - (a) adding a node with key ∞ to the list L and (b) a CAS to set the mark-bit of $next$, is called to deactivate a range-collector. Setting the mark bit of the $next$ works as the linearization point of the RANGESEARCH operation. After this step any call to IsActive returns false. Clearly, the terminal node of L with key ∞ causes termination of the collection of any further node. Additionally, the SnapRange method also calls the method BlockFurtherReports which works along the same lines as in a snap-collector and thus ensures that no further concurrent modify operation can be reported to RN.

On completion of SnapRange, $mode$ is changed to CLEAN and the RCNode RN is attempted to be detached from the unordered list. If the CAS to detach the node fails, the traversal is restarted. If $mode$ is set to CLEAN then reaching the node $RTail$ indicates that the targeted RCNode has been detached. Finally, the method ProcessRange, line 53, is called to return the desired data-structure-nodes. ProcessRange includes steps of sorting the added reports and combining the nodes from L to produce a set of nodes with keys in the range $[lo\ hi]$.

3. If the $next$ of cur is neither marked nor null, it indicates an active RANGESEARCH, therefore we check the relation between $([lo\ hi])$ and $([x\ y])$ by calling ChkOvrlap and

   (a) if the relation is isConatined (line 19), we help the undergoing SnapRange at cur and ProcessRange is used to return only those nodes which have the keys $\in [lo\ hi]$.

   (b) if the relation is isDisjoint (line 24), we simply move to the next RSNode as cur does not cover any data-structure-node with key in the range $[lo\ hi]$.

   (c) and finally, if the relation is contains or overlaps (line 26), we help the undergoing SnapRange at cur and the traversal is restarted from $RHead$.

Most commonly, the lock-free 1-dimensional ordered data structures available in literature [46, 62, 71] have a single successful CAS step as the completion of an ADD operation. Similarly, for a REMOVE operation, a successful CAS is required to inject a "mark" at a connecting link from the node to remove. This CAS step is commonly known as the logical remove step, and after that the logically removed node is eventually cleaned out of the data structure. Further, the CONTAINS operations terminate at a single atomic read step after performing the traversal. We have shown a generalized pseudo-code for these operations in the Algorithm 5.3. Now we describe how they synchronize with a concurrent RANGESEARCH operation.

The set operations ADD (line 2 to 12), REMOVE (line 13 to 22), and CONTAINS (line 24 to 28), which are concurrent to a RANGESEARCH, report the data-structure-nodes in a similar way as they do in [76]. An object **Report** consists of the address of the node to be reported and the type of report (ADD or REM). To report a node x, the method SyncWithRangeQuery, line 47 to 49, is called. SyncWithRangeQuery traverses through the unordered list to locate an RCNode which is active and has the range $\ni key(x)$. However, during the traversal no RANGESEARCH is helped. Before reporting an ADD, the

Algorithm 5.3. A Lock-free data structure with the linearizable range search

```
   ▷ Returns the DSNode with key equal k; returns null if no such
   node present.
1  Find(K k)

   ▷ Adds DSNode (k) to return true if no such node present else
   returns false.  tid is the unique id of the thread/process.
   ADD(K k)
2  while true do
3    if (x=Find(k))≠null then
4      if IsPresent(x) then
5        SyncWithRangeQuery(x, REM, tid); return false;
6      else
7        SyncWithRangeQuery(x, REM, tid);
8        ···;▷ Complete REMOVE.
9        cont;
10   else
11     ···;▷ Complete ADD.
12     SyncWithRangeQuery(x, ADD, tid); return true;

   ▷ Removes the DSNode (k) to return true; returns false if no
   such node present.  tid is the unique id of the thread/process.
   REMOVE(K k)
13 if (x=Find(k)) == null then return false;
14 if IsPresent(x) then
15   ···;▷ Logically remove DSNode (k).
16   SyncWithRangeQuery(x, REM, tid);
17   ···;▷ Complete REMOVE.
18   return true;
19 else
20   SyncWithRangeQuery(x, REM, tid);
21   ···;▷ Complete REMOVE.
22   return false;

   ▷ Atomically adds a dummy report to each of A[tid] and R[tid]
   lists of rnd.
23 BlockFurtherReports(RCNode rnd)
```

method `IsPresent` is called to check whether X is logically removed. If no relevant RCNode is found then nothing is reported.

A CONTAINS (line 24 to 28) operation on finding the target node reports it as `ADD` if it is not logically removed else `REM` is reported. If the node is not found, there is nothing to report. An ADD operation (line 2 to line 12) first adds the desired node, then calls the method `SyncWithRangeQuery` to report `ADD`. If a node with query key is found in the data-structure, which is not logically removed, then it behaves as a CONTAINS operations. On finding a node with the query key but logically removed, a `REM` is first reported, after that the pending REMOVE operation is helped and then the ADD is reattempted. A REMOVE (line 13 to line 22) on finding a node with the query key, attempts to logically remove the node, reports `REM` of the node, then completes the remaining steps to clean the node.

Algorithm 5.3. A Lock-free data structure with the linearizable range search

```
▷ Returns true if there exists a DSNode with key equal k else
false.  tid is the unique id of the thread/process.
CONTAINS(K k)
24 | if (x=Find(k)) == null then return false;
25 | if IsPresent(x) then
26 | | SyncWithRangeQuery(x, ADD, tid); return true;
27 | else
28 | | SyncWithRangeQuery(x, REM, tid); return false;
```

5.2.3 Range queries in a lock-free binary search tree

The BST that we used in this work is a modified form of the lock-free external BST of Natarajan et al. [71]. We included parent pointer in the node structure to facilitate depth first search (DFS). See line 3 to line 22 in the Algorithm 5.4 and the Figure 5.3.

Because of maintaining the parent pointers in nodes, obviously, the ADD and REMOVE operations become complex. The alternate approach to perform DFS is by way of maintaining a local stack in a thread. However, to perform a coordinated scan of the data structure, in which given any node, we find its successor, we can not use the method of stack for DFS. Using a stack will always require a scan to start from the root of the BST and if there is already a concurrent RANGESEARCH that has almost completed the range scan and

Algorithm 5.4. DFS implementation in the lock-free binary search tree.

```
▷ Structure of a BST node.
1  struct Node {K key; Node* lt, rt, parent;};

▷ Packet of address of a leaf-node and its parent.
2  struct Window {Node* cur; Node* par;};

▷ Returns the Window containing the node with key just ≥ lo
in the BST; returns null if no such node present.
   FindMinNode(K lo)
3  | prev=*root, cur=root→lt;
4  | while true do
5  | | while (cur→lt ≠ null) do
6  | | | prev=cur; turn=(k < prev→key);
7  | | | cur=turn ? prev→lt : prev→rt;
8  | | if cur→key<lo and cur→key<prev→key then
9  | | | cur=prev→rt
10 | | if (cur→lt == null) then break;
11 | if cur→key < lo then return null;
12 | else return Window(cur, prev);

▷ Returns a Window with BST node with key just greater than
or equal to the key(x →cur).
   Next(Window x)
13 | prev=x·par, cur=x·cur;
14 | if cur→key < prev→key then cur=prev→rt;
15 | else
16 | | gP = prev→parent;
17 | | while pre→key < gP→key do
18 | | | prev=gP; gP=prev→parent;
19 | | prev=gP; cur=prev→rt;
20 | while (cur→lt ≠ null) do
21 | | prev=cur; cur=prev→lt;
22 | return Window(cur, prev);
```

stored the nodes in the L list in an RCNode, we will have large wastage of work. To avoid that, we use parent pointers.

In an instance of **Window**, line 2, we store the address of a leaf-node, which contains data in an external BST, along with the address of its parent. Given these addresses we can always find the successor of the leaf-node. Given the lower limit of a target range, the method `FindMinNode` here, line 3 to 12, returns the **Window** that contains the first node to be stored in the list L as required in the method `SnapRange` at the line 56. The method `Next` for the BST called at line 59 in `SnapRange` performs one step of the DFS and is given in line 37 to 22.

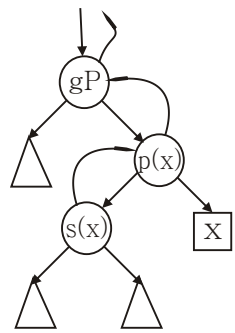


Fig. 5.3: A sub-tree of an external BST with parent pointers

5.3 Correctness proof

We prove that our algorithm is linearizable and lock-free. The system model and definition of basic notions are as given in the section 2.1 in chapter 2. The ADT that we implement here is defined by the set of mappings $\mathcal{M}' \subset \mathcal{M}$ such that $\mathcal{M} = \{\text{ADD}, \text{REMOVE}, \text{CONTAINS}, \text{RANGESEARCH}\}$.

5.3.1 Proof

First we list out the linearization points of the ADT operations as presented in the generalized pseudo-code of the Algorithms 5.2 and 5.3. We already mentioned that the linearization point of a RANGESEARCH operation is the atomic `CAS` step at line 63, where the $next$ of the RCNode is marked in the method `Deactivate` by one of the processes performing `SnapRange` at it. The introduction of RANGESEARCH in a linearizable partial implementation of the

ADT OSet with ADD, REMOVE and CONTAINS operations does not alter the usual linearization points of these operations, if the output of RANGESEARCH includes a modification performed or observed by a concurrent such operation. The linearization points for these operations in the generalized case are as below.

For a successful ADD operation, the execution of the CAS, where a new data structure node is added, is the linearization point. For an unsuccessful ADD and a successful CONTAINS, it is at the point where we read the address of the node with matching key in the method Find. For a successful REMOVE operation, the CAS of the logical remove step is the linearization point. Linearization arguments for an unsuccessful REMOVE and a similar CONTAINS have two cases - (a) if there existed a node containing the query key in the data structure at the invocation but was logically or completely removed by a concurrent REMOVE operation before the return of Find, the linearization point is placed just after the linearization point of that REMOVE operation (b) if no node containing the query key existed in the data structure at the invocation of the REMOVE or CONTAINS, the invocation point itself is taken as the linearization point.

However, if the output of the RANGESEARCH does not include a modification either performed or observed by a concurrent such operation, we linearize the concurrent operation just after the RANGESEARCH. In effect, the linearization points of all the concurrent operations are *anchored* at the linearization point of the RANGESEARCH, which is the atomic CAS step at line 63, if the output of the RANGESEARCH misses the observed or performed modification by a concurrent ADD, REMOVE or CONTAINS operations. We can order the linearization points anchored at the same CAS step in any arbitrary order, for example, in the order of their invocation points.

Now, because our range search algorithm is independent of the operations ADD, REMOVE and CONTAINS, we shall prove the linearizability of the full implementation $\mathcal{I}_{\mathcal{M}'}$, where $\mathcal{M}' = \{\text{ADD}, \text{REMOVE}, \text{CONTAINS}, \text{RANGESEARCH}\}$, of the ADT OSet by building upon a linearizable partial implementation $\mathcal{I}_{\mathcal{O}}$, where $\mathcal{O} = \{\text{ADD}, \text{REMOVE}, \text{CONTAINS}\}$.

Theorem 5.1. *(Correctness) The operations* ADD*,* REMOVE*,* CONTAINS *and* RANGESEARCH *are linearizable with respect to the ADT OSet.*

Proof. Let α be an arbitrary execution of the implementation $\mathcal{I}_{\mathcal{M}'}$. Let $\mathcal{H}$ be an arbitrary history of α. We show that a sequential history $\mathcal{S}$ obtained by following the steps: (a) in $\mathcal{H}$ append appropriate response (in any arbitrary order) of all the operations which have performed their linearization steps as stated above to obtain $ext(\mathcal{H})$, (b) drop the invocation steps without a matching response to obtain $complete(ext(\mathcal{H}))$, and (c) construct $\mathcal{S}$ by arranging the

invocation-response pair of operations according to their linearization points, is consistent.

Let $\rho{=}\mathcal{H}|_{\mathcal{O}}$ and $\sigma{=}\mathcal{S}|_{\mathcal{O}}$ be the projections of $\mathcal{H}$ and $\mathcal{S}$ such that they contain operations belonging to only $\mathcal{O}$. Please note that, for any operation $op{\in}\sigma$, $op(k)$, where $k{\in}\mathbb{R}$, returns either true or false, whereas, for a RANGESEARCH operation, RANGESEARCH(x, x')

Now, by the assumption that $\mathcal{I}_{\mathcal{O}}$ is linearizable, the sequential history σ is consistent. That directly implies that in σ and for a key $k{\in}\mathbb{R}$,

1. if $op_1, op_2{\in}\sigma$ and $op_1(k){\rightarrow_\sigma}op_2(k)$ and $op_1{=}$ADD and $op_2{=}$ADD and $op_1(k) {=}op_2(k){=}$true then $\exists$ $op_3{\in}\sigma$ s.t. $op_1(k){\rightarrow_\sigma}op_3(k){\rightarrow_\sigma}op_2(k)$ and $op_3{=}$REMOVE and $op_3(k)$ =true.

2. if $op_1{\in}\sigma$ and $op_1{=}$ADD and $op_1(k){=}$false then $\exists$ $op_2{\in}\sigma$ and $op_2{=}$ADD and $op_2(k){=}$true and $\nexists$ $op_3{\in}\sigma$ and op_3 =REMOVE and $op_3(k){=}$true s.t $op_1(k){\rightarrow_\sigma}op_3(k)$ $\rightarrow_\sigma op_2(k)$.

3. if $op_1, op_2{\in}\sigma$ and $op_1(k){\rightarrow_\sigma}op_2(k)$ and $op_1{=}$REMOVE and $op_2{=}$REMOVE and $op_1(k)$ $=op_2(k){=}$true then $\exists$ $op_3{\in}\sigma$ s.t. $op_1(k){\rightarrow_\sigma}op_3(k)$ $\rightarrow_\sigma op_2(k)$ and $op_3{=}$ADD and $op_3(k)$ =true.

4. if $op_1, op_2{\in}\sigma$ and $op_1(k){\rightarrow_\sigma}op_2(k)$ and $op_1{=}$REMOVE and $op_2{=}$REMOVE and $op_1(k){=}$false and $op_2(k){=}$false then either (a) $\nexists$ $op_3{\in}\sigma$ s.t. $op_1(k)$ $\rightarrow_\sigma op_3(k)$ $\rightarrow_\sigma op_2(k)$ and $op_3{=}$ADD and $op_3(k){=}$true or (b) $\nexists$ $op_3{\in}\sigma$ s.t. $op_3(k)$ $\rightarrow_\sigma op_1(k)$ and $op_3{=}$ADD and $op_3(k){=}$true.

5. if $op_1{\in}\sigma$ and $op_1{=}$CONTAINS and $op_1(k){=}$true then $\exists$ $op_2{\in}\sigma$ and $op_2{=}$ADD and $op_2(k){=}$true and $\nexists$ $op_3{\in}\sigma$ and $op_3{=}$ REMOVE and $op_3(k){=}$true s.t. $op_2(k){\rightarrow_\sigma}op_3(k)$ $\rightarrow_\sigma op_1(k)$.

6. if $op_1, op_2{\in}\sigma$ and $op_1(k){\rightarrow_\sigma}op_2(k)$ and $op_1{=}$REMOVE and $op_2{=}$CONTAINS and $op_1(k){=}$true and $op_2(k){=}$false then $\exists$ $op_3{\in}\sigma$ s.t. $op_1(k){\rightarrow_\sigma}op_3(k)$ $\rightarrow_\sigma op_2(k)$ and $op_3{=}$ADD and $op_3(k){=}$true.

The above expressions describe the consistency of the sequential history σ in the sense that in between two successful ADD operations with same key there must be a successful REMOVE of that key, or, for two successful REMOVE with same key there must be a successful ADD of that key or for a successful CONTAINS operation with a given key there must be a prior successful ADD with the same key, and so on. Here by success we mean a CONTAINS/ADD/REMOVE operation returning true in its response.

Now, with these assumptions in place, we need to show that ρ is consistent. Let ρ_n be a sub-history of ρ that contains the *first* n complete operations. Let

$\mathbb{A}_n$ be the dataset which was added to the data structure by the successful ADD operations in ρ_n. Let $\mathbb{B}_n$ be the dataset which was removed from the data structure by the successful REMOVE operations in ρ_n. Let $\mathbb{C}_n = \mathbb{A}_n/\mathbb{B}_n$. We use (strong) induction on n to show that ρ_n is consistent $\forall\ n{\geq}1$.

Suppose that ρ_n is consistent $\forall\ n\ :\ 1{\leq}n{\leq}i$. Let the $(i+1)^{th}$ operation in ρ_n be op, where $x, x'{\in}\mathbb{R}$. Then for ρ_{i+1} we prove the following:

1. Let op be RANGESEARCH(x, x') and RANGESEARCH$(x, x')\ \ni\ k$.

   (a) We show that if $\exists\ op_1{\in}\rho_i$ and $op_1(k){\rightarrow_\sigma}op(k)$ and $op_1{=}$ADD and $op_1(k){=}$true then $\nexists\ op_2{\in}\rho_i$ and $op_2{=}$ REMOVE and $op_2(k){=}$true s.t. $op_1(k){\rightarrow_\sigma}op_2(k){\rightarrow_\sigma}op(k)$.
   Proof: Suppose there $\exists$ such an $op_2{\in}\rho_i$. Then by the construction of the linearization points, the RANGESEARCH(x, x') must have noticed this REMOVE operation. That implies that either the REMOVE operation itself would have reported the logical removal of the node containing k or the RANGESEARCH operation started after REMOVE and they were not concurrent. That implies that $k\ \notin$ RANGESEARCH(x, x'), which is a contradiction.

   (b) We show that if $\exists\ op_1{\in}\rho_i$ and $op_1(k){\rightarrow_\sigma}op(k)$ and $op_1{=}$CONTAINS and $op_1(k){=}$true then $\nexists\ op_2{\in}\rho_i$ and $op_2{=}$ REMOVE and $op_2(k){=}$true s.t. $op_1(k){\rightarrow_\sigma}op_2(k)\ {\rightarrow_\sigma}op(k)$.
   Proof: Same as (a) above.

   (c) We show that if $\exists\ op_1{\in}\rho_i$ and $op_1(k){\rightarrow_\sigma}op(k)$ and $op_1{=}$ADD and $op_1(k){=}$true then $\nexists\ op_2{\in}\rho_i$ and $op_2{=}$ CONTAINS and $op_2(k){=}$false s.t. $op_1(k){\rightarrow_\sigma}op_2(k)\ {\rightarrow_\sigma}op(k)$.
   Proof: Suppose there $\exists$ such an $op_2{\in}\rho_i$. Then by the algorithm, either the CONTAINS operation found a node with key k as logically removed or it did not find it at all. If it found, the node logically removed then by the construction of the linearization points, the RANGESEARCH (x, x') must have either been notified of the same or the RANGESEARCH started after the completion of CONTAINS and they were not concurrent. Which in turn implies that $k\ \notin$ RANGESEARCH(x, x'). This is a contradiction. On the other hand, if CONTAINS did not find this node then there must have been a successful REMOVE operation after the ADD and before the CONTAINS. However, that is not possible by (a) above, hence contradiction.

2. Let $op{\in}\mathcal{O}$ then by the consistency of σ mentioned before, they are consistent among themselves.

(1) and (2) together prove that ρ_{i+1} is consistent. Thus, ρ_n is consistent for $n = i + 1$ provided ρ_i is consistent. Hence, by (strong) induction ρ_n is consistent for all $n \geq 1$. □

Proving lock-freedom of the full implementation $\mathcal{I}_{\mathcal{M}}$ of OSet is straightforward. Because $\mathcal{I}_{\mathcal{O}}$ is lock-free, at least one thread finishes its operation if none of them are performing RANGESEARCH. Now, even if RANGESEARCH is performed, the operations in the set $\mathcal{O}$ perform a single `CAS` for reporting without any failure-retry and thus even if one thread finishes its operation in $\mathcal{I}_{\mathcal{O}}$, it must finish its operation in $\mathcal{I}_{\mathcal{M}}$ as well. With regard to concurrent RANGESEARCH operations in $\mathcal{I}_{\mathcal{M}}$, they do not perform any write in the data structure. Further, in the augmented list, whenever a `CAS` fails to add a new RCNode, the traversal during the reattempt starts from the $RHead$ and any pending operation is helped to finish. Thus, for concurrent RANGESEARCH operations, it can not be possible that no operation finishes in finite number of steps. Thus we have the following theorem.

Theorem 5.2. *(Liveness) The operations* ADD, REMOVE, CONTAINS *and* RANGESEARCH *are lock-free.*

This concludes the proof of the presented algorithm.

5.4 Experimental Evaluation

5.4.1 Experimental Setup

We implemented the presented algorithms in Java using RTTI. We evaluated the introduced range search algorithm in three lock-free data-structures - (a) H_LinkedList: linked list of [46] , (b) Im_Ex_BST: a lock-free external BST in which a non-recursive traversal is facilitated using parent-links as described in section 5.2.3 (c) ConcSkipList: the skip list of [62].

We thankfully obtained the basic code of [76] from the authors in a personal communication. They used `AtomicMarkedReference` objects of java.util.concurrent.atomic library in the snap-collector implementation as well as in the linked list of [46], whereas the java library code of [62] was used for the skip list. To optimize the code, we aligned all the implementations to RTTI by replacing every instance of `AtomicMarkedReference` object with a volatile variable and used `AtomicReferenceFieldUpdater` on top of it for `CAS`.

We compared the algorithm with the range search implementation of [16] (BA_KST64), which is based on Java RTTI. We used the author's code available at their home-page. In the experiments in [16], $k = 64$ produced the best results among the k-ary search trees. Therefore, we chose to compare our implementation with the one with $k = 64$.

To simulate the variation due to the contention, we selected the combination of key-range, percentage of operations and the number of threads as following: (a) the % of (ADD, REMOVE, CONTAINS, RANGESEARCH) $\in \{(05, 05, 89, 01), (05, 05, 88, 02), (25, 25, 89, 01), (25, 25, 88, 02)\}$, (b) $|\{key \in K\}| \in \{10^2, 10^3\}$ and (c) the number of threads $\in \{2, 4, 8, 16, 28, 32\}$.

We used a machine with a dual chip Intel(R) Xeon(R) E5-2695 v3 processor with 14 hardware threads per chip (28 hardware threads in total with hyper-threading) running at 2.30 GHz. The machine has 64 GB of RAM and runs over CentOS Linux 7.1.1503 (Kernel version: 3.10.0-229.el7.x86_64) with Java HotSpot(TM) 64-Bit Server VM (1.8.0_51) with 1 GB initial heap size and 15.6 GB maximum heap size. All the implementations were compiled using `javac` version 1.8.0_51 and the runtime flags -d64 -server were used. We performed 10 repetitions of 5 seconds runs for each combination of the parameters shown in the graphs. The average over the 10 trials are recorded. The keys in all the data-structures are taken of Integer type. We ran the experiments for up to 32 threads to observe the scalability above the thread saturation limit - 28 threads, of the processor.

In the experiments, the keys are selected at random from the chosen key-ranges following a uniform distribution. Additionally, all the threads in the experiment perform all the operations selecting next operation at random with a probability expressed by the distribution percentage. In RANGESEARCH experiments, we recorded the throughput of the method `Size` which computes the size of the node-set returned by a call of RANGESEARCH. To call a RANGESEARCH($[\mathsf{lo}, \mathsf{hi}]$), we randomly selected two keys from the chosen key-range and passed their min as lo and their max as hi. For a linearizable `Size` method in BA_KST64, we used the length of the return of the `subset` method thereof.

5.4.2 Observations and Discussion

Figure 5.4 demonstrates the experimental observations. We observed a completely different behavior of our implementation in comparison to that of BA_KST64 with regards the performance and overhead.

- With the growth in the number of modify operations , our method substantially outperforms BA_KST64 (even linked list performs better than KST64 in smaller data-structure with high modification percentage). This

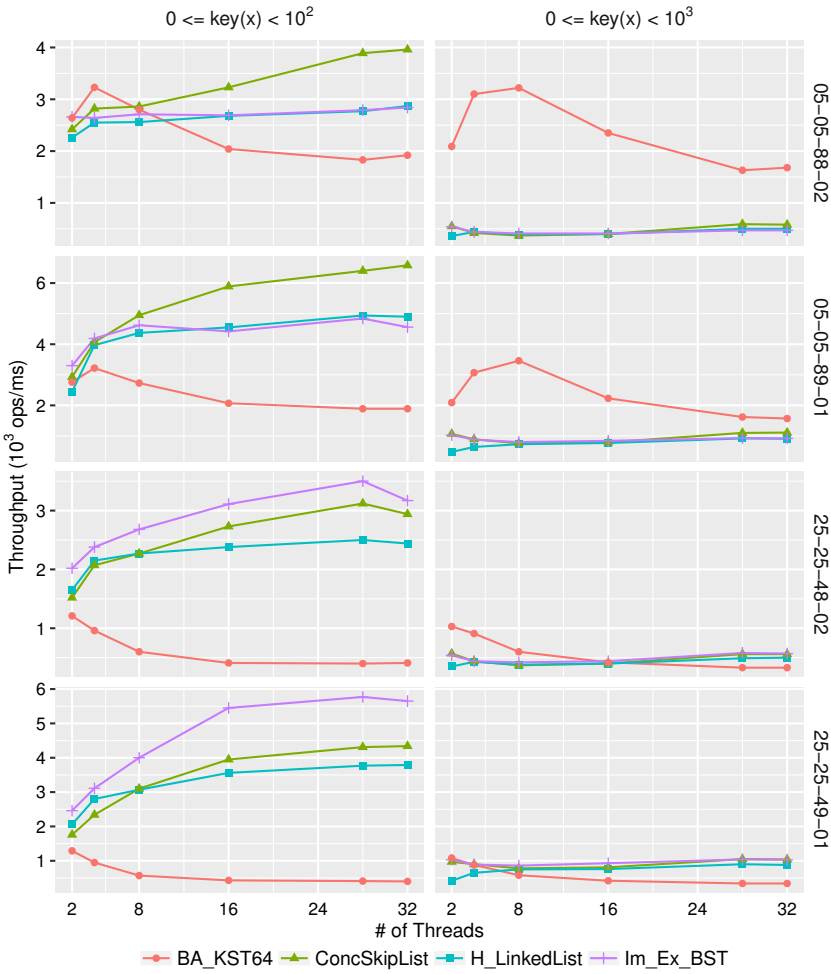


Fig. 5.4: Performance of the implementations.

is expected from the growing number of validations required in BA_KST64 with the growth in modify operations.

- In all the cases, as the number of threads increases, our method exhibits good scalability whereas it is opposite in BA_KST64. This again can be understood in terms of increasing number of validations required in high contention scenarios in BA_KST64.

- Among the data-structures considered for evaluating our generic method, in high contention cases BST outperforms the skip list, whereas in cases of low contention and smaller size of the data structure, skip list wins over the BST. This can be explained in terms of higher number of steps required to find the successor of a node in an external BST.

- On increasing the percentage of RANGESEARCH operations, the throughput of our algorithm decreases across the data-structure types, whereas we do not see similar throughput change in BA_KST64. It indicates that our method has marginally higher overhead compared to BA_KST64, specifically, when the number of concurrent threads and percentage of modify operations is low. It can be explained in terms of the fundamental difference in the snapshot collection strategies ([2] vs. [58]). We make CONTAINS operations report to the concurrent RANGESEARCH, whereas in BA_KST64, CONTAINS do not bother about RANGESEARCH. The experimental evaluations in [76] also showed somewhat similar behaviour with respect to the comparison between throughput and overhead of [76] and [77].

Chapter Summary

In this chapter we presented a generic method to perform linearizable range search in lock-free 1-dimensional ordered data structures. Our experiments showed that the proposed range search method scales well in high contention scenarios.

The k-ary tree by Brown et al. [16] used dirty bits in nodes. We observed that this method achieves better throughput in low contention scenarios compared to our method. We can design a hybrid method that uses similar strategy to achieve scalability of RANGESEARCH *together with lower overhead in low contention scenarios, whereas falls back to the presented method when percentage of concurrent modification is high. Also, we can explore the design of a lock-free BST which facilitates faster computation of successor of a given node, for instance, by connecting the leaves.*

Part IV

LOCK-FREE MULTIDIMENSIONAL POINT SEARCH

6

CONCURRENT LINEARIZABLE NEAREST NEIGHBOUR SEARCH IN LOCKFREE-KD-TREE

Chapter Abstract

The Nearest neighbour search (NNS) is a fundamental problem in many application domains dealing with multidimensional data. In a concurrent setting, where dynamic modifications are allowed, a linearizable implementation of NNS is highly desirable.

This chapter introduces the LockFree-kD-tree (LFkD-tree): a lock-free concurrent kD-tree, which implements an abstract data type (ADT) that provides the operations ADD, REMOVE, CONTAINS, *and* NNSEARCH. *Our implementation is linearizable. The operations in the LFkD-tree use single-word* `CAS` *atomic primitives.*

We experimentally evaluate the LFkD-tree using several benchmarks comprising real-world and synthetic datasets. The experiments show that the presented design is scalable and achieves significant speed-up compared to the implementations of an existing sequential kD-tree and a recently proposed multidimensional indexing structure, PH-tree.

6.1 Introduction

6.1.1 Background

Given a dataset of multidimensional points, finding the point in the dataset at the smallest distance from a given *target point* is typically known as the nearest neighbour search (NNS) problem. This fundamental problem arises in numer-

ous application domains such as data mining, information retrieval, machine learning, robotics, etc.

A variety of data structures available in the literature, which store multidimensional points, solve the NNS in a sequential setting. Samet's book [80] provides an excellent collection of data structures for storing multidimensional data. Several of these have been adapted to perform parallel NNS over a static data structure. However, both sequential and parallel designs primarily consider NNS queries without accommodating dynamic addition or removal (modifications) in the data structure. Allowing concurrent dynamic modifications exacerbates the challenge substantially. A typical real-life application is presented in the section 6.5.

The wide availability of multi-core machines, large system memory, and a surge in the popularity of in-memory databases, have led to a significant interest in the index structures that can support NNS with dynamic concurrent addition and removal of data. However, to our knowledge no complete work exists in the literature on concurrent data structures that support NNS.

Typically, a hierarchical tree-based multidimensional data structure stores the points following a space partitioning scheme. Such data structures provide an excellent tool to *prune* the subsets of a dataset that do not contain the target nearest neighbour. Thus, an NNS query *iteratively scans* the dataset using such a data structure. The iterative scan procedure starts with an initial guess, at every iteration visits a subset of the data structure (e.g. a subtree of a tree) that can potentially contain a better guess, and is unvisited until the last iteration, updates the current guess if required, and thereby finally returns the nearest neighbour.

In a concurrent setting, performing an iterative scan along with concurrent modifications, faces an inescapable challenge. Consider the case of an operation op performing an NNS query in a hierarchical multidimensional data structure that stores points from $\mathbb{R}^d$ and where Euclidean distance is used. Let $a{=}\{a_i\}_{i=1}^{d} \in \mathbb{R}^d$ be the target point of the NNS. Let us assume that $k^*{=}\{k_i^*\}_{i=1}^{d} \in \{k : k \text{ is key of a node}\}$ is the nearest neighbour of a at the invocation of op. In a sequential setting, where no addition or removal of datapoints occurs during the lifetime of op, k^* remains the nearest neighbour of a at the return of op. However, if a concurrent addition is allowed, a new node with key k^{**} may be added to the data structure in a subset that may already have been visited or got pruned by the completion of the latest iteration step. Clearly, op would not visit that subset. Suppose that k^{**} was closer to a compared to k^*, if op returns k^*, it is not consistent to an operation which observes that the addition of k^{**} completes before op.

We aim to design a lock-free linearizable data structure for NNS. In re-

cent years, a number of practical lock-free search data structures have been designed: skip-lists [81], binary search trees (BSTs) [33, 53, 71, 24], etc. Despite the growing literature on lock-free data structures, the research community has largely focused on one-dimensional search problems. To our knowledge, no complete design of any lock-free multidimensional data structure exists in the literature.

The challenge appears in two ways: designing a concurrent lock-free multidimensional data structure that supports NNS and ensuring the linearizability of NNS.

One of the most commonly used multidimensional data structures for NNS is the kD-tree, introduced by Bentley [14]. In principle, a kD-tree is a generalization of the BST to store multidimensional data. Friedmann et al. [37] proved that a kD-tree can process an NNS in expected logarithmic time assuming uniformly distributed data points. Various efforts, including approximate solutions, have contributed to improving the performance of NNS in kD-trees [70, 7]. Furthermore, several parallel kD-tree implementations have been presented, specifically in the computer graphics community, where the focus is on accelerating the applications, such as the ray tracing, in single-instruction-multiple-data (SIMD) programming model [86]. Unfortunately, these designs do not fit concurrent setting where we desire linearizable NNS with concurrent modifications. For robotic motion planning, Ichnowski *et al.* [54] used a kD-tree of 3-dimensional data in which they add nodes concurrently. However, this design does not support REMOVE and the canonical implementation of NNSEARCH, using recursive tree-traversal, is not linearizable.

The contributions of this work are the following:

1. We describe a linearizable implementation of an abstract data type (ADT) that provides ADD, REMOVE, CONTAINS and NNSEARCH operations for a multidimensional dataset.
2. To illustrate the implementation, we present LockFree-kD-tree (LFkD-tree) - an efficient concurrent lock-free kD-tree. LFkD-tree requires atomic single-word read and compare-and-swap primitives.
3. For experimental validation of the LFkD-tree, we use a 2-dimensional real-world dataset and several synthetic datasets representing extreme cases. We evaluate our implementation against an existing sequential kD-tree implementation and a recently proposed multidimensional index structure - PATRICIA-hypercube-tree implementation [85].

Further in this chapter, first, we present the basic design of the LockFree-kD-tree (LFkD-tree) (section 6.3). Thereafter, we detail the lock-free implementation (section 6.3). On describing the algorithm, we present the proof of its correctness (section 6.4). We describe an interesting real-life application of this

work (section 6.5). Finally, we describe experimental evaluation of our algorithm against an existing sequential kD-tree and the PATRICIA-hypercube-tree [85][1] (section 6.6).

6.1.2 A high-level summary of the work

The main challenge in implementing a linearizable NNSEARCH is to ensure that it is not oblivious to the concurrent modifications in the data structure. NNSEARCH requires an iterative scan, which collects, along with pruning, an atomic *snapshot*.

In general, concurrent data structures do not trivially support atomic snapshots. Some exceptions are - the lock-based BST by Bronson et al. [15], the lock-free Trie by Prokopec et al. [77] and lock-free k-ary search tree by Brown et al. [16].

Petrank et al. presented a method to support atomic snapshots in one dimensional lock-free ordered data structures that implement sets [76]. They illustrated their method in lock-free linked-lists and skip-lists. We extended [23] the method of [76] to propose lock-free linearizable range search in one dimensional data structures presented in chapter 5.

The main idea in [23, 76] is augmenting the data structure with a pointer to a special object, which provides a platform for an ADD/ REMOVE/ CONTAINS operation to *report* modifications to a concurrent operation performing a full or partial snapshot. Nevertheless, collecting an atomic snapshot of a multidimensional data structure to perform an NNSEARCH would be naive. We need to adapt the procedure of iterative scan, which benefits from an efficient hierarchical space partitioning structure, to a concurrent setting.

Our work proposes a solution based on augmenting a concurrent data structure with a pointer to a special object called *neighbour-collector*. A neighbour-collector provides a platform for *reporting* concurrent modifications that can otherwise *invalidate* the output of a linearizable NNSEARCH.

Essentially, an operation NNSEARCH(α) first searches for an exact match of α in the data structure, and if it succeeds, returns α itself as its nearest neighbour. If an exact match is not found, before starting the iterative scan, NNSEARCH(α) *announces* itself. The announcement uses a new *active* neighbour-collector that contains the target point α and the current best guess for the nearest neighbour of α. On completing the iterative scan, it *deactivates* the neighbour-collector. A concurrent operation, after completing its steps, checks

[1] In this work, we are not interested in an existing parallel or sequential implementation that does not provide a REMOVE operation, in which case lock-free design poses little challenge. We could find only these two existing implementations that provide REMOVE along with NNSEARCH.

for any active neighbour-collector, and if found, reports its output if it was a better guess than the current best guess available. Finally, NNSEARCH(α) outputs the best guess among the collected and the reported neighbours as the nearest neighbour of α.

Naturally, there can be multiple concurrent NNSEARCH operations with different target points, and we must allow each of them to continue its iterative scan, after announcing it as soon as it begins. To handle multiple concurrent announcements, we use a lock-free linked-list of neighbour-collector objects. The data structure stores a pointer to one end of this list, say the *head*. A new neighbour-collector is allowed to be added only at the other end, say the *tail*.

Consequently, before announcing a new iterative scan, an NNSEARCH operation goes through the list and checks whether there is an active neighbour-collector with the same target point. If an active neighbour-collector is found, it is used for a concurrent *coordinated* iterative scan(explained in the next paragraph). A neighbour-collector is removed from the lock-free linked-list as soon as the associated iterative scan is completed. Hence, at any point in time, the length of the list is at most the number of active NNSEARCH operations.

During an iterative scan, a subset of the dataset is pruned depending on whether the distance of the target point from a *bounding box* covering the subset is greater than that from the current best guess. Now, if the current best guess at a neighbour-collector is the outcome of already pruned many subsets, an NNSEARCH that starts its iterative scan at a later time-point, or is slow (or even delayed), will be able to complete much faster. Thus, the coordination among the concurrent NNS, via their iterative scans at the same neighbour-collector, speeds them up in aggregation.

The basic design of the LFkD-tree is based on the lock-free BST of Natarajan et al. [71]. To perform an iterative scan, we implement an efficient fully *non-recursive traversal* using *parent* links, which is not available in [71]. Thus, to manage an extra link in each node, our design requires extra effort for the lock-free synchronization. The modify operations use single-word-sized atomic `CAS` primitives. The *helping mechanism* is based on the *operation descriptors* at the child-links. Consequently, extra object allocations for synchronization is avoided. The linearizable implementation of NNSEARCH is not confined to the LFkD-tree, and it can be used in a similar concurrent implementation of any other multidimensional data structure available in [80].

6.2 LockFree-kD-tree: Basic Design

6.2.1 Design of the LFkD-tree

The LFkD-tree is a *point kD-tree* in which each node, that stores data, is assigned at most one data-point. Typically, to partition $\mathbb{R}^d$, we use *axis-orthogonal hyperplanes* that are given by $x_i{=}c,\ 1{\leq}i{\leq}d$. The structure and consequently the NNS performance of a kD-tree heavily depends on the *splitting rule* - the procedure to select the partitioning hyperplanes. Traditionally, in a sequential setting, to construct a kD-tree from static data, the partitioning hyperplanes are chosen to coincide with points that belong to the given dataset. In this approach, similar to an internal BST representation [24], each node is used for storing data. However, removing a node from an internal BST is costly, more so in a concurrent setting [53, 24]. With this in mind, we opt for an external BST representation [33, 71] to design the LFkD-tree. In this design, only *leaf-nodes* contain the data-points and *internal-nodes* route a traversal, see the Figure 6.1 (b). More importantly, it gives us the flexibility to compute c and $i : 1{\leq}i{\leq}d$ for a hyperplane $x_i{=}c$, which may not coincide with a data-point.

To compute the values of c and i, in the scenarios where the entire dataset is available beforehand, a number of splitting rules exist in the literature [37, 70]. These rules focus on the hierarchical partition of a *closed hyperrectangle* that covers the entire dataset and not only tries to balance a kD-tree but also optimize its depth. In a concurrent setting, where we do not have knowledge of the entire dataset in advance, the partitioning hyperplane needs to be computed dynamically and in a very localized fashion. For the LFkD-tree, we formulate a simple and practical splitting rule, the ***local-midpoint rule***, as given in the section 6.2.2. In this work, we do not delve in to an analytical discussion of the splitting rule.

A leaf-node of a LFkD-tree Υ, contains a unique data-point as its *key*, whereas, an internal-node corresponds to a partitioning hyperplane. Without ambiguity, we denote a leaf-node containing key $k{=}\{k_i\}_{i=1}^{d}{\in}\mathbb{R}^d$ by $\texttt{Nd}(k)$ (or $\texttt{Nd}(\{k_i\}_{i=1}^{d})$), and an internal node associated with a hyperplane $x_i{=}c$, by $\texttt{Nd}(i, c)$. An internal-node has three *links* connected to its *left-child*, *right-child* and *parent*. We indicate the link emanating from a node N and incoming to a node M by $\mathsf{N}{\rightsquigarrow}\mathsf{M}$. Access to Υ is given by the *address* of (pointer to) a unique node *root*. A node N is said to be *present* in Υ, denoted by $\mathsf{N}{\in}\Upsilon_t$, if it can be *reached* following the links starting from the root. For each internal-node $\texttt{Nd}(i, c)$, Υ maintains the following invariants: (i) a node $\texttt{Nd}(\{k_i\}_{i=1}^{d})$ belongs to the *left subtree*, if $k_i{<}c$, (ii) a node $\texttt{Nd}(\{k_i\}_{i=1}^{d})$ belongs to the *right subtree*, if $k_i{\geq}c$ and (iii) both subtrees are themselves LFkD-tree. (i) and (ii) together are

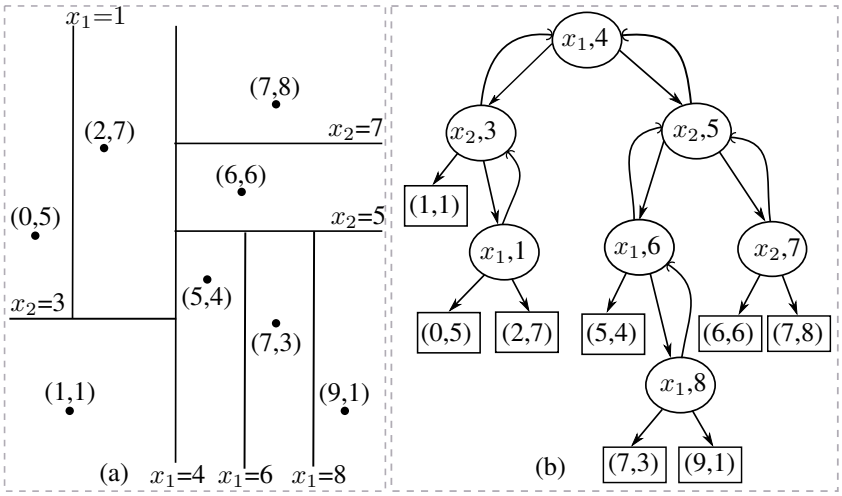


Fig. 6.1: LFkD-tree Structure

called the *symmetric order* of the LFkD-tree. Figure 6.1 illustrates the structure of a subtree of a LFkD-tree corresponding to a sample 2-dimensional dataset.

6.2.2 Sequential Behaviour of the ADT Operations

LFkD-tree implements an abstract data type that provides operations ADD, REMOVE, CONTAINS and NNSEARCH. For each of the operations, we start with a *query*: start from the root, traverse down Υ, at each internal node decide left / right child direction using the symmetric order until arrive at a leaf-node.

To perform ADD(a), $a \in \mathbb{R}^d$, if the query terminates at a leaf-node Nd(b), $b \in \mathbb{R}^d$, and $b = a$ (an element-wise comparison of keys), ADD(a) returns false. However, if $b \neq a$, we allocate a new internal-node Nd(i, c) with its child links connected to two leaf-nodes Nd(a) and Nd(b). If p(Nd(b)) was the parent of Nd(b) at the termination of query, we connect the parent link of Nd(i, c) to p(Nd(b)). We update the link p(Nd(b))$\rightsquigarrow$Nd(b) to point to Nd(i, c) and return true. To compute i and c, we employ the local-midpoint rule as given below.

Local-midpoint rule: *$1 \leq i \leq d$ is the index of the coordinate axis along which a and b have the maximum coordinate difference; if there are more than one such axis then select the one with the lowest index. Take the hyperplane as* $x_i = \frac{a[i]+b[i]}{2}$.

To perform REMOVE(a), if the leaf-node where the query terminates, has

the key a, i.e. $\texttt{Nd}(a) \in \Upsilon$, we modify the link from the *grandparent* of $\texttt{Nd}(a)$, denoted by $g(\texttt{Nd}(a))$, to its parent, to connect the *sibling* of $\texttt{Nd}(a)$, $s(\texttt{Nd}(a))$, to $g(\texttt{Nd}(a))$; and return true. If $\texttt{Nd}(a) \notin \Upsilon$, REMOVE$(a)$ returns false. To perform CONTAINS(a), using a similar query we check whether $\texttt{Nd}(a) \in \Upsilon$ and return true or false accordingly.

The operation NNSEARCH(a) is non-trivial. On termination of the initial query, if we reach at $\texttt{Nd}(b)$ and $b = a$, clearly the nearest neighbour of a, available in the dataset stored in Υ, is a itself. However, if $b \neq a$, we take b as our *current best guess* and check whether the *other subtree* of $p(\texttt{Nd}(b))$ (the current subtree consists the single node $\texttt{Nd}(b)$) stores a *better guess*. Suppose that $p(\texttt{Nd}(b)) = \texttt{Nd}(i, c)$. Now, any point on the *other side* of the hyperplane $x_i = c$ will be at least at a distance $|a_i - c|$ from the target point $\{a_i\}_{i=1}^{d}$. Therefore, if $|a_i - c| > ||a, b||_2$ (the Euclidean distance between a and b), we must prune the other subtree i.e. one rooted at $s(\texttt{Nd}(b))$, otherwise we visit it in the *next iteration*. A subtree once visited is not visited again and thus we traverse back to the root of Υ. At the termination of the iterative scan of Υ, the current best guess is returned as the nearest neighbour of a.

6.3 LockFree-kD-tree: Implementation

6.3.1 Lock-free Synchronization: Basics

In a sequential setting, when REMOVE(a) modifies $g(\texttt{Nd}(a)) \rightsquigarrow p(\texttt{Nd}(a))$, no operation is executed concurrently with a possibility to modify either of the links - $p(\texttt{Nd}(a)) \rightsquigarrow \texttt{Nd}(a)$ or $p(\texttt{Nd}(a)) \rightsquigarrow s(\texttt{Nd}(a))$. However, in a concurrent setting, where these pointers are shared by multiple operations, an ADD operation can concurrently modify any of these pointers. It may result into the newly added node not being a part of the LFkD-tree. Similarly, if $s(\texttt{Nd}(a))$ is an internal-node, a concurrent REMOVE operation trying to remove a child of $s(\texttt{Nd}(a))$ may end up connecting $p(\texttt{Nd}(a))$ to the sibling of the removed child which results into a wrong outcome. Essentially, for a correct concurrent implementation of modify operations in a LFkD-tree, we need to keep the pointers $p(\texttt{Nd}(a)) \rightsquigarrow \texttt{Nd}(a)$ and $p(\texttt{Nd}(a)) \rightsquigarrow s(\texttt{Nd}(a))$ *fixed* when $g(\texttt{Nd}(a)) \rightsquigarrow p(\texttt{Nd}(a))$ is updated to $g(\texttt{Nd}(a)) \rightsquigarrow s(\texttt{Nd}(a))$. Additionally, because we maintain parent pointers, we also need to keep the pointer $g(\texttt{Nd}(a)) \rightsquigarrow p(\texttt{Nd}(a))$ fixed when $s(\texttt{Nd}(a)) \rightsquigarrow p(\texttt{Nd}(a))$ is updated to $s(\texttt{Nd}(a)) \rightsquigarrow g(\texttt{Nd}(a))$, in case $s(\texttt{Nd}(a))$ is an internal node.

For a lock-free synchronization we can not use locks to keep these shared pointers fixed. Instead of locks, we design the *helping* protocol for operations. Basically, the idea is: whenever an operation encounters a shared pointer fixed

(although not by a lock) by a concurrent modify operation, i.e. obstructed, it takes necessary steps to complete the pending operation and thereby avoids the obstruction in its own progress. This ensures that no non-faulty thread is blocked due to a delayed or crashed thread and thereby provides progress guarantee.

Ellen et al. [33] suggested to put *operation descriptors*, using CAS, at the nodes $g(\texttt{Nd}(a))$ and $p(\texttt{Nd}(a))$ by a REMOVE operation and at $p(\texttt{Nd}(a))$ by an ADD operation, before updating the necessary pointers. An operation descriptor stores information about the changes that a modify operation needs to make. If CAS fails, appropriate helping is performed, using the information from the descriptor, before a reattempt.

Natarajan et al. [71] suggested that instead of putting the descriptors at the nodes $g(\texttt{Nd}(a))$ and $p(\texttt{Nd}(a))$, putting them at the links $p(\texttt{Nd}(a))\rightsquigarrow\texttt{Nd}(a)$ and $p(\texttt{Nd}(a))\rightsquigarrow s(\texttt{Nd}(a))$ improves performance. Both these designs use single-word-sized CAS to put descriptors and update the pointers.

As mentioned before, the basic structure of our LFkD-tree is based on an external BST. Therefore, for the lock-free synchronization in the LFkD-tree, we build upon the lock-free BST algorithm of [71]. The fundamental idea of the design is a *lazy remove* procedure that is essentially based on a protocol of atomically injecting *operation descriptors* on the links connected to the node to be removed, and then modifying those links to disconnect the node from the LFkD-tree. If multiple concurrent operations try to modify a link simultaneously, they synchronize by *helping* one of the pending operations that would have successfully injected its descriptor.

More specifically, to REMOVE the node $\texttt{Nd}(a)$, as shown in the Figure 6.2(b), we use a CAS to inject operation descriptors at the links $p(\texttt{Nd}(a))\rightsquigarrow\texttt{Nd}(a)$, $g(\texttt{Nd}(a))\rightsquigarrow p(\texttt{Nd}(a))$ and $p(\texttt{Nd}(a))\rightsquigarrow s(\texttt{Nd}(a))$, in this order. We call these descriptors `mark`, `tag` and `flag` respectively. An operation descriptor works as an *information source* about the steps already performed in $\text{REMOVE}(a)$ and thus a concurrent operation, if obstructed at a link with descriptor, *helps* by performing the remaining steps. In particular, a `mark` at a link indicates that the next step would be to inject a `tag` at the link $g(\texttt{Nd}(a))\rightsquigarrow p(\texttt{Nd}(a))$, whereas, a `tag` indicates that the next step is to inject the descriptor `flag` at the link $p(\texttt{Nd}(a))\rightsquigarrow s(\texttt{Nd}(a))$. Finally, a `flag` indicates the completion of steps of injecting operation descriptors and thereafter the required link updates are done. The *helping mechanism* ensures that the concurrent ADD and REMOVE operations do not violate any invariant maintained by the LFkD-tree. The steps of a REMOVE operation are shown in the Figure 6.2(c). An ADD operation uses a single CAS to update the target link only if it is free from any operation descriptor, otherwise it helps the concurrent pending REMOVE operation. A

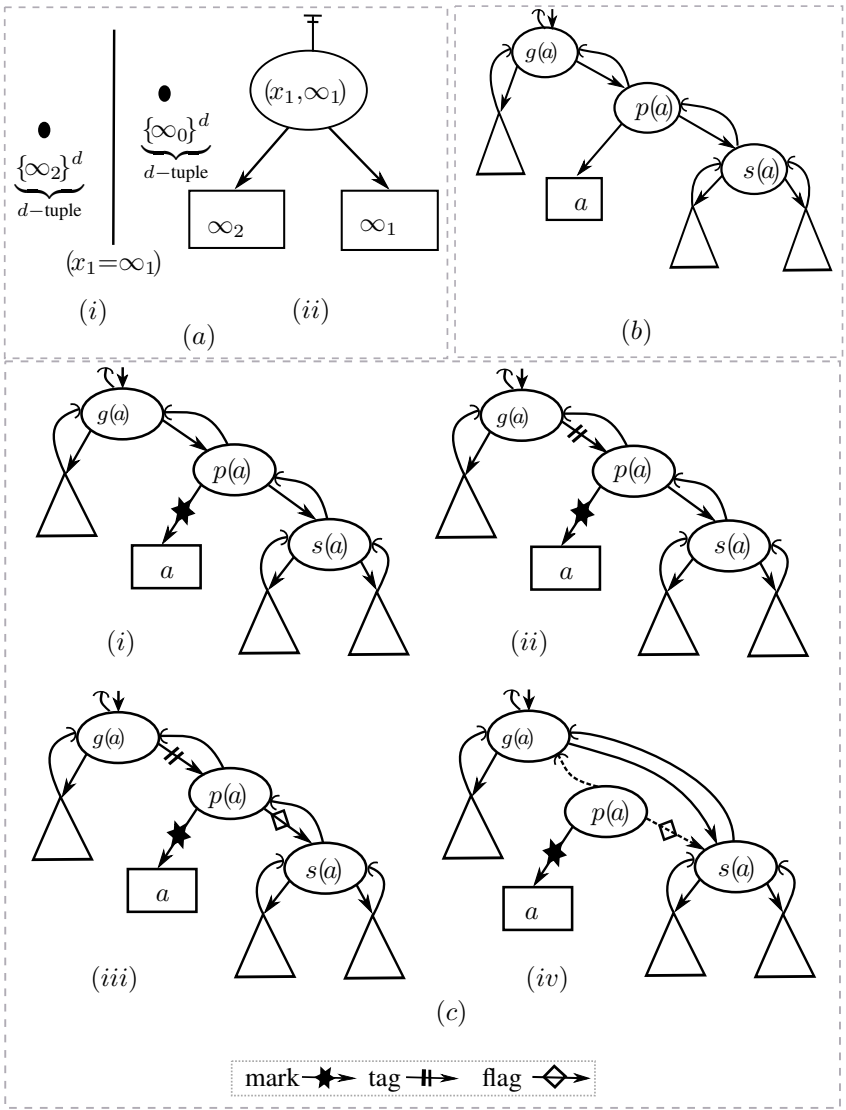


Fig. 6.2: ADD and REMOVE operations in LFkD-tree

CONTAINS or NNSEARCH operation does not perform help.

We call the CAS step, which injects a mark at $p(\mathtt{Nd}(a))\rightsquigarrow\mathtt{Nd}(a)$, the *logical remove* of a. After this step, a CONTAINS(a) that reads $p(\mathtt{Nd}(a))\rightsquigarrow\mathtt{Nd}(a)$ returns false. Accordingly, ADD(a) helps to complete the pending REMOVE(a), if it reads $p(\mathtt{Nd}(a))\rightsquigarrow\mathtt{Nd}(a)$ with a mark descriptor, and then reattempts its own steps. The helping mechanism guarantees that a logically removed node will be eventually detached from the LFkD-tree.

To realize the atomic step to inject an operation descriptor, we replace a link using a CAS with a single-word-sized packet of a link and a descriptor. Given a pointer delegates a link, a well-known method in C/C++ to pack extra information with a pointer in a single memory-word is *bit-stealing*. In a x86/64 machine, where memory allocation is aligned on a 64-bit boundary, three least significant bits in a pointer are unused. The three operation descriptors used in our algorithm fit over these bits.

For ease of exposition, we assume that a memory allocator always allocates a variable at a new address and thus an ABA (see section 1.2.2) problem does not occur. For lock-free memory reclamation in a C/C++ implementation of the algorithm, a method such as one based on reference counting [40] can be used. Whereas, traditionally a Java implementation uses the JVM garbage collector. Furthermore, to avoid null pointers at the beginning of an application, we use a subtree containing an internal-node and two leaf-nodes which work as *sentinel nodes*. See the Figure 6.2(a). The keys in the sentinel nodes maintain $\infty_0>\infty_1>\infty_2>k_i$, $1\leq i\leq d$, for any data point $\{k_i\}_{i=1}^{d}$ stored in the LFkD-tree. The sentinel internal-node $\mathtt{Nd}(1,\infty_1)$ works as the root of the LFkD-tree and the entire dataset is stored in its left subtree.

6.3.2 Linearizable ADD, REMOVE and CONTAINS operations

(A) Overview

Algorithm 6.1. The node structure in the LFkD-tree

```
1 struct INode {long i; double c; Node* lt, rt, pr;} ▷ A subclass of Node.
2 struct LNode {K k;} ▷ A subclass of Node.
3 root := INode*(1, ∞1, LNode*({∞2}^d), LNode*({∞0}^d), null);
```

First, we present the node-structures in the LFkD-tree, which will help in the subsequent discussion. The classes **INode** and **LNode**, which represent an internal- and a leaf- node respectively, are shown in lines 1 and 2 in Algorithm 6.1. Every **INode**, in addition to the fields i and c that represent the associated hyperplane, has three pointers lt, rt and pr that delegate the *left-child*, *right-child* and *parent* links, respectively. A **LNode** contains only an array k to represent a data-point $k=\{k_i\}_{i=1}^{d}\in\mathbb{R}^d$. The node-pointer root, line 3, delegates address of the sentinel node $\texttt{Nd}(1,\infty_1)$. As a convention, if x is a *field* of a class C, we use pc·x to indicate the field x of an instance of C pointed by pc; and, the type of a pointer to an instance of C is indicated by C$*$. Note that, **INode** and **LNode** inherit **Node**.

(B) The algorithm

Algorithm 6.2. LFkD-tree: `Search` method

```
   ▷Return a child-direction.
   Dir(Node* Nd(i,c)·ref, K k)
1  |  return k[i] < c ? L : R;
   |  ▷Directions - L: left, R: right; implemented as a boolean.

   ▷Return a child-pointer.
   Child(Node* pa, dir cD)
2  |  return cD = L ? pa·lt : pa·rt;

   Search(Node* pa, Node* a, K k)
3  |  while Ptr(a)·class ≠ LNode do
4  |  |pa := Ptr(a); a := Child(pa, Dir(pa, k));
5  |  return ⟨pa, a⟩;
```

We have already described in section 6.3.1 the operation descriptors and their denotation about the different steps of a REMOVE operation. In the following algorithms, we use the methods `IsMark`, `IsFlag` and `IsTag` to check whether a pointer has descriptor `mark`, `flag` and `tag` (actually `ltag` and `rtag`, see below), respectively. Further, to pack these descriptors, we use the methods `Mark`, `Flag` and `Tag`, respectively. To get the value of a pointer free from all descriptors, which gives a node-address, we use the method `Ptr`. The ADD, REMOVE and CONTAINS operations, along with the methods called by them, are described in a modular fashion in the Algorithm 6.2.

The basic methods `Dir` and `Child` are used in traversal. The method `Search`, line 3 to 5, which performs a query, returns the pointers to the leaf-node and its parent, where the query terminates.

Algorithm 6.2. LFkD-tree: The CONTAINS operation

```
   CONTAINS(K k)
6  | pa := root; a := pa·lt;
7  | ⟨pa, a⟩ := Search(pa, a, k);
8  | if !IsMark(a) then
9  | | Sync(Ptr(pa), Ptr(a));
10 | | return k = Ptr(a)·k ? true : false;
11 | else return false;
```

A CONTAINS, line 6 to 11, stats with calling `Search`, returns true only if the pointer a does not have `mark` and the query key matches at the leaf-node pointed by a at line 10; else it returns false, line 11. A CONTAINS calls `Sync`, line 9, before return to synchronize with concurrent NNSEARCH operations. We describe `Sync` in the section 6.3.3.

Algorithm 6.2. LFkD-tree: The REMOVE operation

```
   REMOVE(K k)
12 | pa := root; a := pa·lt;
13 | while true do
14 | | ⟨pa, a⟩ := Search(pa, a, k);
15 | | if !IsMark(a) then
16 | | | if k ≠ Ptr(a)·k then return false;
17 | | | if IsFlag(a) then pa := Help(pa, a);
18 | | | marker := Mark(a); cD := Dir(pa, k);
19 | | | else if ChCAS(pa, a, marker, cD) then
20 | | | | Help(pa, a); return true;
21 | | else return false;
22 | | a := Child(pa, Dir(pa, k));
```

The method `AddNode`, line 23 to 36, attempts to add a new node in the LFkD-tree. It starts with calling `Search`, line 25. If the returned leaf-node-pointer a is found containing `mark`, it indicates that the node containing the query key is logically removed, and therefore, the method `Help` is called to

help the concurrent pending REMOVE operation, line 35. Otherwise, the node pointed by a is checked whether it contains the query key, line 27, and if found, false is returned, line 28. `AddNode` also outputs the descriptor-free pointers to the leaf-node and its parent where the query terminated. However, if the leaf-node did not contain the query-key, it is checked whether a has the descriptor `flag`, which indicates a pending REMOVE of the *sibling* of the node pointed by a; and if `flag` is found, `Help` is called, line 29. We describe `Help` in the next subsection. Only in the case a is descriptor-free, the method `NewNode` (see lines 37 to 42) is called to allocate a new node, and a `CAS` executed in the method `ChCAS` (see lines 43 to 47), called at line 33, modifies a to add the new node. On that, return includes true.

The operation ADD, line 48 to 49, calls `AddNode` to get the pointer to the node and its parent, either added by itself or already present there, containing its query key, and the result of addition accordingly. Thereafter, ADD calls the method `Sync`, line 49, and outputs the result.

The REMOVE operation, line 12 to 22, performs query in a similar way calling `Search`, line 14. At the return of `Search`, if a is found to have `mark`, it indicates that even if the query key k was present in the LFkD-tree, has already been logically removed and therefore REMOVE returns false, line 21. If a is free of `mark`, we check if the node pointed by a contains the query key, and if not, REMOVE returns false, line 16. However, if the pointer a is found to have the descriptor `flag`, it indicates a pending REMOVE of the sibling of the node pointed by a, and therefore we call the method `Help` to perform helping steps. After return of `Help`, the steps are reattempted. Finally, if a was descriptor-free, `mark` is injected on it via the method `ChCAS`, line 19, and if it succeeds, the `Help` is called to take further steps and true is returned, line 20.

(C) The Helping steps

The method `Help` is described In the Algorithm 6.3, line 1 to 6. We call `Help` at a pointer to a leaf-node which had been injected with either the descriptor `mark` or `flag`. Therefore, it first decides the type of descriptor, and then accordingly calls either `HelpMrk`, line 5, or `HelpFlg`, line 6.

The method `HelpMrk`, line 7 to 10, first calls `ApndTag` to fix the $g(\mathtt{Nd}(a))$, pointed by ga. And then calls `HelpTag` to complete the remaining steps of REMOVE. To distinguish between the `tag` put by the REMOVE of left and right child of $p(\mathtt{Nd}(a))$, we use two types of `tag`: `ltag` and `rtag`. In the method `ApndTag`, line 13 to 26, if the link was found already tagged, the type of `tag` (`ltag` or `rtag`) is read using the method `TagDir`. And, if the link was found to be tagged by a REMOVE of the other child of $p(\mathtt{Nd}(a))$, first that REMOVE is

Algorithm 6.2. LFkD-tree: The ADD operation

```
   AddNodeK k
23 | pa := root; a := pa·lt;
24 | while true do
25 | | ⟨pa, a⟩ := Search(pa, a, k);
26 | | if !IsMark(a) then
27 | | | if k = Ptr(a)·k then
28 | | | | return ⟨Ptr(pa), Ptr(a), false⟩;
29 | | | if IsFlag(a) then pa := Help(pa, a);
30 | | | else
31 | | | | n := LNode(k); cD := Dir(pa, k);
32 | | | | newNd := NewNode(a, n·ref, pa);
33 | | | | if ChCAS(pa, a, newNd·ref, cD) then
34 | | | | | return ⟨newNd·ref, n·ref, true⟩;
35 | | | else pa := Help(pa, a);
36 | | | a := Child(pa, Dir(pa, k));
   ▷Crates a new internal-node.
   NewNode(Node* a, Node* b, Node* p)
37 | ka := a·k; kb := b·k;
38 | i := {i : 1≤i≤d and |ka[i]−kb[i]|≥{|ka[j]−kb[j]|}_{j=1}^{d}};
   | ▷Local-midpoint rule is applied.
39 | c := (ka[i]+kb[i])/2;
40 | left := (ka[m] < kb[m] ? a : b);
41 | right := (ka[m] > kb[m] ? a : b);
42 | return INode(m, c, left, right, p);
   ChCAS(Node* pa, Node* exp, Node* new, dir cD)
43 | if (cD = L) and pa·lt = exp then
44 | | return CAS(pa·lt·ref, exp, new);
45 | else if (cD = R) and pa·rt = exp then
46 | | return CAS(pa·rt·ref, exp, new);
47 | else return false;
   ADD(K k)
48 | ⟨pa, a, result⟩ := AddNodek;
49 | Sync(pa, a); return result;
```

Algorithm 6.3. LFkD-tree: `Help` method

```
Help(Node* pa, Node* a)
1  | cD := (a·k[pa·i] < pa·c) ? L : R;
2  | if IsFlag(a) then
3  |   | ga := Pr(pa); sa := Child(pa, !cD);
4  |   | pD := (a·k[ga·i] < ga·c) ? L : R;
5  |   | return HelpFlg(ga, pa, sa, pD);
6  | else return HelpMrk(pa, a, cD);
```

```
HelpMrk(Node* pa, Node* a, dir cD)
7  | ga := ApndTag(pa, a, cD);
8  | pD := Dir(ga, a·k); pl := Child(ga, pD);
9  | if Ptr(pl) = pa then HelpTag(ga, pl, pD);
10 | return ga;
```

```
HelpTag(Node* ga, Node* pl, bool pD)
11 | pa := Ptr(pl); sD := (TagDir(pl) = L ? R : L);
12 | HelpFlg(ga, pa, ApndFlg(pa, sD), sD);
```

```
ApndTag(Node* pa, Node* a, dir cD)
13 | while true do
14 |   | ga := Pr(pa); pD := Dir(ga, a·k);
15 |   | pl := Child(ga, pD);
16 |   | if Ptr(pl) = pa then
17 |   |   | if IsTag(pl) then
18 |   |   |   | if TagDir(pl) = cD then return ga;
19 |   |   |   | else HelpTag(ga, pl, pD);
20 |   |   | else if IsFlag(pl) then
21 |   |   |   | grGa := Pr(ga);
22 |   |   |   | HelpFlg(grGa, ga, pa, Dir(grGa, a·k));
23 |   |   | else if ChCAS(ga, pl, Tag(pl, cD), pD) then
24 |   |   |   | return ga;
25 |   | else if pl = a then pa := ga;
26 |   | else return ga;
```

```
ApndFlg(Node* pa, dir sD)
27 | while true do
28 |   | sa := Child(pa, sD);
29 |   | if IsMark(sa) then return sa;
30 |   | else if IsFlag(sa) then return Ptr(sa);
31 |   | else if IsTag(sa) then HelpTag(pa, sa, sD);
32 |   | else if ChCAS(pa, sa, Flag(sa), sD) then
33 |   |   | return sa;
```

Algorithm 6.3. LFkD-tree: `Help` method

```
   HelpFlg(Node* ga, Node* pa, Node* sa, dir pD)
34 if Ptr(pl := Child(ga, pD)) = pa then
35   if Pr(Ptr(sa)) = pa then
36     CAS(Pr(Ptr(sa))·ref, pa, ga);
37   ChCAS(ga, pl, sa, pD);
38 return ga;
```

helped and then we reattempt, line 19, otherwise we return ga, line 18. However, if the link $g(\texttt{Nd}(a))\rightsquigarrow p(\texttt{Nd}(a))$ is found `flagged`, line 22, it indicates a pending REMOVE of $s(p(a))$ and therefore we help it before reattempt. On successfully `tagging` the link $g(\texttt{Nd}(a))\rightsquigarrow p(\texttt{Nd}(a))$, we return the pointer ga, line 24. Also, if $g(\texttt{Nd}(a))$ is found not connected with $p(\texttt{Nd}(a))$, we return ga, line 26, and REMOVE operation terminates because it indicates the completion.

The method `HelpTag`, line 11 to 12, reads the direction of the child whose REMOVE had tagged the link $g(\texttt{Nd}(a))\rightsquigarrow p(\texttt{Nd}(a))$ (represented by *pl*), line 11, `flags` the (sibling) link calling `ApndFlg` and finally calls `HelpFlg` to perform the remaining steps, see line 12.

In `ApndFlg`, line 27 to 33, if the link $p(\texttt{Nd}(a))\rightsquigarrow s(\texttt{Nd}(a))$ (represented by sa) was found `marked`, line 29, we return this link as it is, because it is guaranteed that the REMOVE operation that `marked` this link, will perform helping before reattempting its `CAS` to put a `tag` in the method `ApndTag`. In that case, the `marked` link is further carried to the method `HelpFlg` and connected to $p(\texttt{Nd}(a))$. If $p(\texttt{Nd}(a))\rightsquigarrow s(\texttt{Nd}(a))$ is found flagged, we return $s(\texttt{Nd}(a))$, represented by the value of sa without any descriptor i.e. `Ptr`(sa), line 30. On a successful `CAS` to `flag` the link, we return address of $s(\texttt{Nd}(a))$ represented by sa, line 33.

Finally, the method `HelpFlg`, line 34 to 37, if required, connects the pr pointer of $s(\texttt{Nd}(a))$ to $g(\texttt{Nd}(a))$, see line 36. And lastly, node a is detached from the LFkD-tree by connecting $s(\texttt{Nd}(a))$, represented by *sa*, to $g(\texttt{Nd}(a))$ using a `CAS` at line 37.

6.3.3 Linearizable Nearest Neighbour Search

In this section, we begin with the algorithm that addresses the case where concurrent NNSEARCH operations have coinciding target points. We build on it to present the algorithm for general cases without any restriction. However,

before describing the NNSEARCH algorithms, we discuss the linearizability of the operations as its motivation.

(A) Linearization argument

Consider the concurrent modifications in the LFkD-tree, when an NNSEARCH operation, say *op*, performs its iterative scan. We can ensure, by checking whether `IsMark` returns true, that the key of a leaf-node which was logically removed, is never collected as a current guess for the nearest neighbour. Similar to a CONTAINS operation, we can place the linearization point of *op* at the point where it reads the pointer to the leaf-node, say `Nd`, whose key is returned as the nearest neighbour. Now, if `Nd` is logically removed after it was read by *op*, by a concurrent REMOVE operation, say op_1, which returns before the return of *op*, we still do not loose the linearizability argument, simply because linearization point of op_1 is ordered after that of *op*.

Algorithm 6.4. LFkD-tree: Structure of Neighbour-collector

```
1 struct Nebr {Node* a; double d;} ▷ Neighbour
2 struct NbrClctr {K tgt; bool isAct; Nebr* col, rep; NbrClctr* next;}
  ▷ Neighbour-collector
3 ncp := NbrClctr*(null, false, null, null, null);
4 tail := NbrClctr*(null, null, null, null, false);
5 head := NbrClctr*(null, null, null, tail·ref, false);
```

However, in case of a concurrent ADD operation, say op_2, we may be at the risk of returning *not the latest* nearest neighbour and thereby invalidating the linearizability, as explained in the section 6.1.1. Thus, op_2 essentially needs to *report* its modification to *op*, after completing its own steps. Now, suppose that op_2 got delayed after adding a new node `Nd` to the LFkD-tree and could not report it to *op*. If in the meantime a concurrent CONTAINS operation, say op_3, read `Nd` and returned as usual, we may again loose linearizability because to an outside observer the addition of a better guess is visible, possibly before the return of *op*, by way of op_3, although *op* did not return it. Therefore, op_3 also needs to report its output to *op*. Now, given that op_2 and op_3 are made to report their output to *op*, we need to change the linearization point of *op*. To maintain

the order, we put the linearization point of op just after that of op_2 or op_3, if op happens to return the nearest neighbour which was a report by one of them.

Note that, we need to be careful about unnecessary reporting, which may possibly be harmful as well, in the following sense. Suppose that op_2 and op_3 both got delayed after their linearization. Now, if invocation of op happened after that, op is guaranteed to read `Nd`, if `Nd` contained the nearest neighbour of the target point. But, if in between the linearization of op_3 and invocation of op, a concurrent REMOVE removed `Nd`, op will certainly not read it, and a reporting may render the linearization point of op to be shifted to even before its invocation, which is undesired. To avoid this situation, before every reporting, we first ascertain whether the node to be reported is logically removed by calling the method `IsMark`.

Having described the linearization argument, it seems tempting that we could have avoided the entire reporting method for synchronization between concurrent NNSEARCH and ADD/CONTAINS operations. For example, we could have reordered the linearization points of an NNSEARCH and an ADD operation in a way that if an NNSEARCH does not return the latest key, say x^*, added in the dataset, the NNSEARCH is linearized before the ADD, though it returned after the ADD. Furthermore, the concurrent CONTAINS operations that return x^*, are linearized after ADD, as expected. However, that does not seem in line with the idea of linearizability that proposes that the operations in a concurrent data structure must demonstrate their sequential behaviour.

Before describing the algorithm for NNSEARCH, we describe the classes to implement the *neighboour-collector*. See the Algorithm 6.4. The class **Nebr**, line 1, represents a packet of a data-point, as contained in a leaf-node pointed by the node-pointer a, and its distance, given as d, from the target point of an NNSEARCH. The class **NbrClctr**, line 2, represents a *neighbour-collector*: the platform for collecting and reporting the nearest neighbour. **NbrClctr** contains pointers to two **Nebr** instances: col points to one that contains collected data-point during iterative scan by an NNSEARCH operation and rep points to one that contains a data-point reported by a concurrent operation, in addition to the target point tgt. It also contains a boolean isAct, which if set true, implies an *active* neighbour-collector; and a neighbour-collector-pointer nxt to implement an augmented lock-free linked-list of neighbour-collectors. The LFkD-tree is augmented with a pointer ncp, line 3, initialized to point to an *inactive* neighbour-collector.

(B) Concurrent NNSEARCH *with coinciding target points*

When concurrent NNSEARCH operations have coinciding target points, they can output same result by adopting a single atomic step, which is performed during the lifetime of one of them, as the linearization point for each of them; the real-time order amongst them can be taken as the order of any fixed step for example their invocation step. Thus, essentially they require a single *iterative* scan. Principally, it is similar to the linearizable snapshot algorithm of [76]. The pseudo-code of the algorithm is given in the Algorithm 6.5.

The methods `Seek` and `NextGuess`, see lines 2 and 17, are used to perform a non-recursive traversal of the LFkD-tree. We describe these methods in the subsection (D). Here, we describe how the non-recursive traversal is used to perform co-ordinated iterative scan by concurrent NNSEARCH operations.

The operation NNSEARCH, line 1 to 5, starts with calling the method `Seek`, line 2, to perform the initial query to arrive at a leaf-node. If the pointer to leaf-node a is free of descriptor `mark`, which indicates that the node pointed by a is not logically removed, and if the query key k matches at the leaf-node, which is checked by the distance between k and the key at the leaf-node, k itself is the nearest neighbour available in the dataset and NNSEARCH returns, line 5. Otherwise, NNSEARCH calls the method `NNSync`, which performs further steps and returns the nearest neighbour, line 4. The arrays hi and lo are used to support non-recursive traversal, described in the subsection (D). `NNSync` and methods called subsequently are described here.

The method `NNSync`, line 6 to 15, starts with checking whether ncp points to an active neighbour-collector, and if it does not, it allocates a new active neighbour-collector and attempts a `CAS` to modify ncp to point to the new one, line 10. In case ncp was pointing to an active neighbour-collector, we attempt to update the current best guess of nearest-neighbour as the key in the leaf-node. On an active neighbour-collector, the method `Collect` is called to perform a *coordinated iterative scan*, line 14.

`Collect`, line 16 to 18, calls the method `NextGuess`, line 17, to perform next iteration that can better the current best guess of the nearest neighbour. Before attempting to add the new guess, contained in a leaf-node, to the neighbour-collector using the method `AdNebr`, it is always checked whether the leaf-node is logically removed by calling the method `ChkValid`. Please note that, given a (possibly stale) pointer to a leaf-node, we can not directly check whether it was logically removed. Therefore, we also supply the pointer to the parent and thus the method `ChkValid`, line 39 to line 43, gets the latest pointer to the leaf-node considering the fact that a new internal-node may get added between the parent of the leaf-node and the leaf-node to be reported.

Algorithm 6.5. LFkD-tree: NNSEARCH operations with coinciding target points

```
NNSEARCH(K k)
1  pa := root; a := pa·lt; hi := {∞0}^d; lo := {−∞0}^d;
2  ⟨pa, a⟩ := Seek(pa, a, k, hi, lo);
3  dst := IsMark(a) ? ∞ : ||k, a·k||2;
4  if dst ≠ 0 then return NNSync(pa, a, dst, k, hi, lo);
5  else {Sync(pa, a); return k;}

NNSync(Node* pa, Node* a, double dst, K k, K hi, K lo)
6  while true do
7    on := ncp;
8    if on·isAct = false then
9      cN := Nebr*(a, dst); nn := NbrClctr*(k, true, cN, cN, null);
10     if CAS(ncp·ref, on, nn) then break;
11   else
12     if ChkValid(pa, a) then dst := AdNebr(a, on, col);
13     nn := on; break;
14 nn := Collect(pa, a, dst, k, hi, lo, nn);
15 Deactivate(nn); return Process(nn);

Collect(Node* pa, Node* a, K k, K hi, K lo, double dst, NbrClctr* nn)
16 while pa ≠ Ptr(root) and dst ≠ 0 do
17   ⟨pa, a⟩ := NextGuess(pa, a, dst, k, hi, lo);
18   if ChkValid(pa, a) then dst := AdNebr(a, nn, col);
19 return nn;

AdNebr(Node* a, NbrClctr* nn, bool nt)▷ nt (Neighbour-type):
col or rep.
20 while true do
21   nbr := (nt == col) ? nn·col : report(nn);
22   if nn·isAct and !IsFinish(nbr) then
23     ⟨dst, nb⟩ := NearNbr(a, nn);
24     if nb == null then return dst;
25     if nt == col then res := CAS(nn·col·ref, nbr, nb);
26     else res := CAS(report(nn)·ref, nbr, nb);
27     if res then return dst;
28   else return 0;
```

Algorithm 6.5. LFkD-tree: NNSEARCH operations with coinciding target points

```
   NearNbr(Node* a, NbrClctr* nn)
29 | distTgt := ||a·k, nn·tgt||_2; col := nn·col; rep := report(nn);
30 | if distTgt < col·d and distTgt < rep·d then return
   | ⟨distTgt, Nebr*(a, distTgt)⟩;
31 | else return ⟨distTgt, null ⟩;
   ──────────────────────────────────────────────────────────────
   BlockNebr(NbrClctr* nn, bool nt)▷ nt (Neighbour-type): col or
   rep.
32 | nbr := (nt == col) ? nn·col : report(nn);
33 | while !IsFinish(nbr) do
34 | | if nt == col then CAS(nn·col·ref, nbr, Finish(nbr));
35 | | else CAS(report(nn)·ref, nbr, Finish(nbr));
36 | | nbr := nt == col ? nn·col : report(nn);
   ──────────────────────────────────────────────────────────────
   ChkValid(Node* pa, Node* a)
37 | k := a·k; ch := Child(pa, Dir(pa, k));
38 | while Ptr(ch)·class ≠ LNode do
39 | | ch := Ptr(Child(ch, Dir(ch, k)));
40 | if IsMark(ch) then return false;
41 | returnch == a ? true: false;
   ──────────────────────────────────────────────────────────────
   Deactivate(NbrClctr* nn)
42 | BlockNebr(nn, col); nn·isAct := false; BlockNebr(nn, rep);
   ──────────────────────────────────────────────────────────────
   Process(NbrClctr* nn)
43 | if report(nn)·d < nn·col·d then return report(nn)·a;
44 | else return nn·col·a;
   ──────────────────────────────────────────────────────────────
   Sync(Node* pa, Node* a)
45 | if ncp·isAct then
46 | | ⟨d, nb⟩ := NearNbr(a, ncp);
47 | | if nb ≠ null and ChkValid(pa, a) then Report(a, ncp);
   ──────────────────────────────────────────────────────────────
48 Report(Node* a, NbrClctr* nn) {AdNebr(a, nn, rep);}
```

AdNebr, line 21 to 29, is called to add a collected or reported neighbour to an active neighbour-collector. It calls the method NearerNbr, shown in line 30 to 31, which returns a new neighbour only if the distance of the new guess is less than the distance of the already collected or reported neighbours to the neighbour-collector.

After completion of the iterative scan, the method Deactivate is called by NNSync at line 15. Deactivate, line 44, other than setting the IsAct to false, also injects a descriptor finish at both the neighbour-pointers of the neighbour-collector using the method BlockNebr. BlockNebr, line 34 to line 38, performs a CAS to replace a neighbour-pointer with one that has the descriptor finish over it, see lines 36 and 37. It ensures that each of the concurrent NNSEARCH operations using same neighbour-collector have same view of it after linearization. The method IsFinish returns true when called on a neighbour-pointer with descriptor finish. Thus, AdNebr can not add a new neighbour in a neighbour-collector if the corresponding pointer is injected with finish, see line 23.

Finally, the method Process, line 45 to 46, is called by NNSync to select the better candidate between the reported and the collected neighbours of the target point, which is returned to the caller NNSEARCH to output. Note that, once a neighbour-collector is *deactivated* by an NNSEARCH, the method AdNebr returns 0, line 29. This in turn, immediately terminates the **While** loop in Collect at the line 16. Thus, as mentioned in section 6.1.2, we can observe that the *coordination* among the concurrent iterative scans at the same neighbour-collector helps a delayed NNSEARCH operation to complete faster.

The method Sync, line 47 to 49, is used by an ADD or a CONTAINS after their completion, see Algorithm 6.2 at lines 9 and 49. Sync is also used by NNSEARCH in the case a matching key is found, see line 5. It first checks the active status of the neighbour-collector and then calls the method NearerNbr to create a neighbour. If the point to be reported is not better than the current best guess available, NearerNbr returns null and in that case Sync returns without any change. Otherwise, it checks whether the leaf node with the point to be reported is logically removed by calling the method ChkValid, and then calls the method Report, which in turn calls AdNebr to add the reported neighbour, line 50.

(C) A general case of Concurrent NNSEARCH *with multiple target points*

To allow multiple concurrent NNSEARCH with non-coinciding target points to progress together, we need to have as many active neighbour-collectors as the

Algorithm 6.6. LFkD-tree: NNSEARCH operations with multiple distinct target points

```
1  tail := NbrClctr*(null, false, null, null, null);
2  head := NbrClctr*(null, false, null, null, tail);
   NNSync(Node* pa, Node* a, double dst, K k, K hi, K lo)
3    nn := null; mode := INIT;
4    retry:
5    while true do
6      p := null; c := head; n := c·nxt;
7      while Ptr(n) ≠ tail do
8        if n = nn and mode = CLEAN then
9          val := Clean(c, nn);
10         if val ≠ null then return val;
11         else goto retry;
12       else if k = n·tgt and n·isAct then
13         nn := n; mode := COLLECT; break;
14       else {p := c; c := n; n := n·nxt;}
15     if mode = INIT and IsMark(n) then
16       CAS(p·nxt·ref, c, Ptr(n)); goto retry;
17     if mode ≠ CLEAN then
18       ⟨val, mode⟩ := Finalize(pa, a, dst, k, hi, lo, p, c, mode);
19       if val ≠ null then return val;
20     else return Process(nn);
   Sync(Node* pa, Node* a)
21   n := head·nxt;
22   while n ≠ tail do
23     if n·isAct then
24       nb := NearNbr(a, n);
25       if nb ≠ null and ChkValid(pa, a) then Report(a, n) ;
26       else break;
27     else n := Ptr(n·nxt);
```

number of different target points. Essentially, we need to have a dynamic list of neighbour-collectors. In this list, before adding a new neighbour-collector, an NNSEARCH must scan through it so that if there was already an active neighbour-collector with a matching target point, coordination among the concurrent iterative scans with coinciding target points can be achieved. For each of the operations in the LFkD-tree to be lock-free, we ensure the lock-freedom of this list as well. Hence, we augment the LFkD-tree with a single-word `CAS` based lock-free list of neighbour-collectors.

Algorithm 6.6. LFkD-tree: NNSEARCH operations with multiple distinct target points

```
   Finalize(Node* pa, Node* a, double dst, K k, K hi, K lo, NbrClctr*
   p, NbrClctr* c, enum md)
28   if md = COLLECT then nn := c; pre := p;
29   else if md = INIT then
30     nn := Allocate(a, dst, k, c); pre := c;
31     if nn ≠ null then mode := COLLECT;
32   if md = COLLECT then
33     nn := Collect(pa, a, dst, k, hi, lo, nn);
34     Deactivate(nn); md := CLEAN;
35     if (val := Clean(pre, nn)) ≠ null then
36       return ⟨val, md⟩;

   Allocate(Node* a, double dst, K k, NbrClctr* c)
37   cNb := Nebr*(a, dst);
38   nn := NbrClctr*(k, true, cNb, cNb, tail);
39   if CAS(c·ref, on, nn) then return nn;
40   else return null;
```

The linearization points remain as before: the concurrent NNSEARCH with coinciding target points share an atomic step during the lifetime of one of them as their linearization point with some order among themselves.

The pseudo-code of the algorithm is given in the Algorithm 6.6, in which every method is absolutely same as those in the Algorithm 6.5, except `NNSync` and `Sync`. The list is initialized with two sentinel nodes pointed by tail and head, with head·nxt set as tail, as given in lines 1 and 2. A new neighbour-collector is added to this list at one of the ends only, which is just before the node pointed by tail. The method of maintaining this list is similar to the lock-

free linked-list of Harris et al. [46], except the fact that no addition happens anywhere in the middle of the list. Removal of a neighbour-collector, say one pointed by c, takes two successful CAS steps: first we inject a mark descriptor at the c·nxt using a CAS and then modify the pointer p·nxt to n with a CAS, if p and n happened to be the pointers to the predecessor and successor, respectively, of the neighbour-collector pointed by c.

We use the method Mark to get a word-sized packet of a neighbour-collector-pointer and the descriptor mark, whereas, the method Ptr masks the descriptor off such a packet and does not change a neighbour-collector-pointer. Please note that, earlier we used the same notation mark for an operation descriptor over a pointer to a LFkD-tree node. However, without any ambiguity, they indicate the descriptor for the type of pointer in the context. Similarly, the methods Mark and IsMark are used depending on the context. Adding a neighbour-collector takes a single successful CAS similar to [46].

The method NNSync, line 3 to 20, as called by NNSEARCH after the initial query in Algorithm 6.5, starts with traversing the list. We maintain an **enum** variable mode that indicates the stages of NNSync. Initially, the mode is INIT. During the traversal, if an active neighbour-collector with matching target point is found, the mode is changed to COLLECT and traversal terminates, line 13. Otherwise, the traversal terminates in the mode INIT itself. On the termination of the traversal in the mode INIT, it is checked whether the neighbour-collector, where traversal terminated (in this case c), is already logically removed, line 15, and if it is, a CAS is attempted to detach it from the list and the traversal is restarted, line 16.

After that, if the mode is INIT or COLLECT, the method Finalize is called. Finalize, line 28 to 36, if called in the mode INIT, allocates a new neighbour-collector by calling the method Allocate, line 37 to 40, otherwise uses the input neighbour-collector. If Allocate could not add a new neighbour-collector, it returns null and the entire process restarts from scratch with a fresh traversal. After successfully adding a new neighbour-collector to the list or asserting that it needs to use an existing one, Finalize calls the methods Collect and Deactivate similar to those in Algorithm 6.5. On deactivating the neighbour-collector, the method Clean is called to remove it from the list and return the value of the nearest neighbour.

Clean, line 41 to 45, performs the two CAS steps to remove the neighbour-collector and calls the method Process, line 44, to compute the nearest neighbour. However, if after injecting mark, it could not modify the nxt pointer of the predecessor, it returns null, which again causes a fresh traversal in the mode CLEAN in Finalize. A traversal in mode CLEAN, if finds the deactivated neighbour-collector, calls the method Clean, line 10, to redo the re-

maining steps and return the nearest neighbour. If the traversal terminates in the mode CLEAN, that implies that a concurrent NNSEARCH would have detached the deactivated neighbour-collector and therefore Process is called to finish, line 20.

Algorithm 6.6. LFkD-tree: NNSEARCH operations with multiple distinct target points

```
   Clean(NbrClctr* pre, NbrClctr* nn)
41 | nxt := nn·nxt;
42 | while !IsMark(nxt) do
43 | | CAS(nn·nxt·ref, nxt, Mark(nxt)); nxt := nn·nxt;
44 | if CAS(pre·nxt·ref, nn, Ptr(nxt)) then return Process(nn);
45 | else return null;
```

(D) The Non-recursive Traversal

The main tool of the non-recursive traversal for the iterative scan is to keep track of an (orthogonal) axis aligned bounding box (AABB) of the points in the subtrees, both visited and pruned. An AABB is described by its two corner points. We use the variables hi and lo throughout the algorithms to represent the two corner points. Initially, in order to begin the query in the operation NNSEARCH, the corner points are taken as $\{\infty_0\}^d$ and $\{-\infty_0\}^d$, see line 1 in the Algorithm 6.5, which cover the entire dataset.

The method Seek, line 1 to 7, which is called by NNSEARCH for the initial query at line 2 in the Algorithm 6.5, starts with the initial AABB as described by the two arrays hi and lo with their initial values, and performs a query absolutely similar to the method Search to arrive at a leaf-node. At the termination of Seek, the arrays AABB represent the bounding box that covers every data-point that can be in the sub-tree of the parent of the leaf-node, where it terminates, which has the same direction as the leaf-node with respect to its parent. We follow the convention that an array is always passed by reference and therefore any modification at any element in a method call persists even after the return of the method call. Thus, at the return of Seek, if the query point did not match at the key of the leaf-node, we go to perform further iterations using the method NextGuess with the current bounding box which represents the rectangular region of the Euclidean space that we have covered.

The method NextGuess, line 8 to 26, performs an iteration for a better guess of the nearest neighbour given the distance of the current guess from

Algorithm 6.7. Non-recursive traversal

```
   Seek(Node* pa, Node* a, K k, K hi, K lo)
1  | cD := (a·k[pa·i] < pa·c) ? L : R;
2  | while Ptr(a)·lt ≠ null do
3  | | pa := Ptr(a); cD := Dir(pa, k);
4  | | a := Child(pa, cD);
5  | | if cD = L then hi[pa·i] := pa·c;
6  | | else lo[pa·i] := pa·c;
7  | return ⟨pa, a⟩;
   NextGuess(Node* pa, Node* a, double dst, K k, K hi, K lo)
8  | cD := (a·k[pa·i] < pa·c) ? L : R;
9  | leafKey := a·k;
10 | while pa ≠ root do
11 | | if cD = L then ntVsted := (pa·c≥hi[pa·i]);
12 | | else ntVsted := (pa·c≤lo[pa·i]);
13 | | if |pa·c − k[pa·i]| < dst and ntVsted then
14 | | | cD := (cD = L ? R : L); a := Child(pa, cD);
15 | | | Seek(pa·ref, a·ref, cD·ref, k, hi)lo;
16 | | | leafKey := a·k;
17 | | | if (leafdst := ||k, leafKey||2) < dst then
18 | | | | if !IsMark(a) then
19 | | | | | dst := leafdst; break;
20 | | else
21 | | | a := pa; pa := Pr(pa); cD := Dir(pa, leafKey);
22 | | | if cD = L then
23 | | | | if pa·c > hi[pa·i] then hi[pa·i] := pa·c;
24 | | | else
25 | | | | if pa·c < lo[pa·i] then lo[pa·i] := pa·c;
26 | return ⟨pa, a⟩;
```

the target point. We input the pointers to the current leaf-node and its parent along with the AABB described by its two corners. The first step is to find the direction of the current sub-tree and then decide whether the other sub-tree of the parent is visited or not, see lines 8, 11 and 12.

In essence, we check whether the axis-orthogonal hyperplane associated with the parent node is beyond the AABB. Having done that, we check whether the unvisited AABB on the other side of the hyperplane should be visited by checking its distance from the target point and comparing it with the current distance as input, see line 13. Now, if we need to visit the other sub-tree, the method `Seek` is called to perform the query and update AABB, line 15, else we traverse back to root. When we traverse back to root, the AABB is widened to cover both sub-tree rooted at an internal node, see lines 23 and 25.

Thus, the method `Collect` repeatedly calls `NextGuess` to perform an iterative scan of the LFkD-tree, see line 17 in Algorithm 6.5.

(E) Relaxation in Consistency Requirements

Algorithm 6.8. Relaxed NNSEARCH operations in LFkD-tree

```
   NNSearchRelaxed(K k)
1  pa := root·ref; a := pa·lt; hi := {∞0}^d; lo := {−∞0}^d;
2  ⟨pa, a⟩ := Seek(pa, a, k, hi, lo);
3  dst := IsMark(a) ? ∞ : ||k, a·k||2;
4  if dst ≠ 0 then
5  |  while pa ≠ Ptr(root) do
6  |  |  ⟨pa, a⟩ := NextGuess(pa, a, dst, k, hi, lo);
7  |  return a;
```

Practitioners prefer better throughput at the cost of exact solution in various applications that require a nearest neighbour search, which is commonly known as approximate-NN (ANN). Generally, in a hierarchical multidimensional data structure like kD-tree, ANN algorithms relax the pruning criterion so that an NNSEARCH operation visits lesser number of subsets and thereby it speeds up the performance. Implementing ANN in a concurrent hierarchical multidimensional data structure may not directly impact the design-complexity as long as we follow the same consistency framework. However, in the spirit of getting better throughput at the cost of exact solution, we can certainly explore the relaxation in consistency requirements of an NNS operation.

Suppose that we do not make an ADD or a CONTAINS operation report its output to a concurrent NNSEARCH and each NNSEARCH progresses in oblivion to any concurrent NNSEARCH. Considering a point of reference *local to a thread*, an NNSEARCH outputs the nearest neighbour that it discovers with a certainty that there was no operation performed by the *thread itself* that can alter the result. Clearly, this relaxed arrangement satisfies the requirements of sequential consistency [31]. The operation NNSEARCHRELAXED, described in Algorithm 6.8, implements a sequentially consistent version of NNSEARCH. We shall utilize its experimental performance to empirically evaluate the overhead of linearizability with respect to NNSEARCH.

6.4 Correctness and Lock-freedom

In section (A), we discussed the arguments that determine linearization steps of NNSEARCH operations when target points are coincident. We also stated in section (C) that the linearization point of an NNSEARCH operation remains unchanged even if the target points of the concurrent NNSEARCH operations do not coincide. Here we list out the linearization points of the operations as the following:

Definition 6.1 (***Linearization points:***). *1. For a successful* ADD *operation, it is at line 44 or line 46 in the method* `ChCAS`*, which is called at line 33 in the method* `AddNode` *and which in turn was called by* ADD.

2. *For a successful* REMOVE *operation, it is at line 44 or line 46 in the method* `ChCAS`*, which is called at line 19 in* REMOVE.

3. *For an unsuccessful* ADD *and a successful* CONTAINS *operation it is at line 4 in the method* `Search` *called from these operations.*

4. *For an unsuccessful* CONTAINS *and* REMOVE *operation, it can be either just after the linearization point of a concurrent* REMOVE *operation or at the invocation point of these operations.*

5. *For a* NNSEARCH *operation, if it returns a data-point which was contained in a collected-neighbour, the linearization point is at line 3 in algorithm 6.7 in the method* `Seek` *called from the* NNSEARCH.

6. *For a* NNSEARCH *operation, if it returns a data-point which was contained in a reported-neighbour, the linearization point is just after the linearization point of either* CONTAINS *or* ADD *that reported the neighbour.*

It is easy to observe in the pseudo-codes presented in the chapter that these linearization points are in between $t^i(op)$ and $t^r(op)$ for an operation $op \in \mathcal{O}$, where $\mathcal{O}=\{\text{ADD}, \text{REMOVE}, \text{CONTAINS}, \text{NNSEARCH}\}$.

Now with that, given any concurrent execution history $\mathcal{H}$ of an implementation $\mathcal{I}_{\mathcal{O}}$, where $\mathcal{O} \subseteq \{\text{ADD}, \text{REMOVE}, \text{CONTAINS}, \text{NNSEARCH}\}$, we form an equivalent sequential history $\mathcal{S}$ by following the steps as described above. And thus it remains to be shown that such a sequential history will be consistent.

To do that, we essentially show that the invariants of the LFkD-tree, as stated in the section 6.2.1 are maintained, and the sequential specifications as described in the section 6.2.2, are satisfied by the consistent operations. Because the implementation of the lock-free list of neighbour-collectors is orthogonal to the implementation of the LFkD-tree, we also need to show that the invariants of the list, as stated in the section 6.3.3(C), are maintained by the NNSEARCH operations. Therefore, first we state the invariants and present some observations and lemmas which help us in that process.

Given a LFkD-tree Υ, let $\texttt{Nd}(i, c)$ be an internal-node and $\texttt{Nd}(\{k_i\}_{i=1}^{d})$ be a leaf node. Υ maintains the following invariants:

Invariant 6.1. *A node* $\texttt{Nd}(\{k_i\}_{i=1}^{d})$ *belongs to the left subtree, if* $k_i < c$.

Invariant 6.2. *A node* $\texttt{Nd}(\{k_i\}_{i=1}^{d})$ *belongs to the right subtree, if* $k_i \geq c$.

Invariant 6.3. *A node* $\texttt{Nd}(\{k_i\}_{i=1}^{d})$ *belongs to the right subtree, if* $k_i \geq c$.

A LFkD-tree state Υ_t that satisfies the invariants 6.1 to 6.3 is called a *valid state*. Now, for the list of the neighbour-collectors, we denote a neighbour-collector by $\texttt{NC}(\{k_i\}_{i=1}^{d})$ if the target point that it contains is $\{k_i\}_{i=1}^{d}$. A neighbour-collector list maintains following invariant:

Invariant 6.4. *In the list there can not be two neighbour-collectors* $\texttt{NC}(\{k_i\}_{i=1}^{d})$ *and* $\texttt{NC}(\{j_i\}_{i=1}^{d})$ *such that* $k_i = j_i \; \forall \; i \; : \; 1 \leq i \leq d$.

To prove that the above invariants are maintained throughout the algorithms, we present following observations and lemmas.

Observation 6.1. *The fields* k *and* i *are never changed in a* **Node**.

Observation 6.2. *Any link in a LFkD-tree is updated only using a* CAS.

Observation 6.3. *The sentinel nodes are never removed.*

Observation 6.4. *The* pr *pointer of the node* root *is never dereferenced.*

Going through the pseudo-code we can observe that once we allocate a node, we never call any store step on the fields k and i and any pointer update is done using a CAS. The choice of keys in the sentinel nodes verifies the third observation. The pr pointer of an internal node is dereferenced only if a REMOVE operation on any of its children is called. Thus the observation 6.3, implies the observation 6.4.

Lemma 6.1. *In each call of* Dir*, line 1, variable* Nd(*i*,*c*)·ref *represents a pointer which is* clean *and points to an internal-node and thus is not* null.

Lemma 6.2. *In each call of* Child*, line 2, pa is* clean *and points to an internal-node and thus is not* null.

Lemma 6.3. *In each call of* ChCAS*, line 43 to 47, pa is* clean *and points to an internal-node, whereas new is* clean *and points to a leaf-node; thus pa and a are both not* null.

Lemma 6.4. *In each call of* Search*, line 3, pa is* clean *and points to an internal-node, whereas a is* clean *and points to a node (internal or leaf); thus both are not* null.

Lemma 6.5. *In each call of* Search*, line 3, pa and a satisfy* $a = pa{\cdot}\mathsf{lt} \mid pa{\cdot}\mathsf{rt}$.

Lemma 6.6. *In each call of* HelpMrk*, line 7, pa is* clean *and points to an internal-node, whereas a is* clean *and points to a leaf-node; thus both are not* null.

Lemma 6.7. *In each call of* HelpFlg*, line 34, ga and pa are* clean *and point to two different internal-nodes, whereas sa is either points to a leaf-node and thus are not* null.

Lemma 6.8. *In each call of* HelpTag*, line 11, ga is* clean *and points to an internal-nodes, whereas pl is either* ltag *or* rtag *and points to an internal-node and thus are not* null.

Lemma 6.9. *In each call of* ApndTag*, line 13, pa and a are* clean*. pa points to an internal-nodes, whereas a points to a leaf-node and thus both are not* null.

Lemma 6.10. *In each call of* ApndTag*, line 27, pa is* clean *and points to an internal-node and thus is not* null.

Lemma 6.11. *A pointer once injected with a descriptor* mark*,* flag*,* ltag *or* rtag *is not injected with any descriptor ever after.*

The lemma 6.1 to 6.10 provide a base to prove that at no point an implementation of the presented algorithm faces a segmentation fault due to the dereferencing of a null pointer during the operations ADD, REMOVE and CONTAINS. To prove these lemmas we inspect the pseudo-code in the Algorithms 6.2 and 6.3. At each call of the utility methods we find that the inputs to the utility methods follow the requirements of these lemmas. A listing of the lines of the pseudo-code containing call of these methods verifies this claim. The statements of this set of lemmas is what we need to prove the next set of lemmas which provides the verified base for postconditions of the LFkD-tree operations.

Lemma 6.12. *At the termination of* `Search` *at line 5,*

(a) pa points to an internal-node and is `clean`.

(b) a points to a leaf-node and can be either `clean` *or* `mark` *or* `flag`.

(c) pa *and a satisfy* $a = pa{\cdot}\mathsf{lt} \mid pa{\cdot}\mathsf{rt}$.

(d) $a{\cdot}\mathsf{k}[pa{\cdot}\mathsf{i}] \geq pa{\cdot}\mathsf{c} \implies a = pa{\cdot}\mathsf{rt}$.

(e) $a{\cdot}\mathsf{k}[pa{\cdot}\mathsf{i}] < pa{\cdot}\mathsf{c} \implies a = pa{\cdot}\mathsf{lt}$.

Following from the lemmas 6.4 and 6.5, the **while** loop ensures that the variable *a* always points to one of the child-pointers of the node pointed by *pa*; this ensures the validity of the lemma 6.12 (a), (b) and (c).

Now, Following the lemma 6.11 shows that the `CAS` steps are performed orderly in a REMOVE operation. It is easy to verify that if the `CAS` steps are orderly in a REMOVE operation, it does not result into the malformation of the LFkD-tree. Also, for an ADD operation, because the single `CAS` that it requires can not happen over a link with descriptor.

Now, the keys in the sentinel nodes vacuously prove the following lemma 6.13, which provides base condition for an induction to prove the theorem 6.1.

Lemma 6.13. *Initially, the LFkD-tree consisting of the sentinel nodes satisfies the invariants as stated in section 6.2.1.*

Now we are prepared to prove theorem 6.1. We use induction to prove it. Using lemma 6.13, when no update has happened, the nodes in the LFkD-tree satisfy the invariants. It is straightforward to observe that no CONTAINS or NNSEARCH operation involves a write (`CAS`) step and therefore they do not change the state of the LFkD-tree. From lemma 6.12, at the end of every call to `Search`, which satisfies the symmetric order of the LFkD-tree, a `CAS` to ADD does not violate the invariant 6.1 to 6.3. For a REMOVE operation, after the

`CAS` to logically removing the node i.e. `mark CAS`, the order of `CAS` do not let any update operation let the node reappear in the LFkD-tree following the lemma 6.11.

Thus if the state of the LFkD-tree was consistent before the application of an update operation, it remains so after its linearization. Using induction the theorem 6.1 follows.

Theorem 6.1. *At any time $t \geq 0$ the LFkD-tree state Υ_t is a valid state.*

Now considering the neighbour-collector-list, its semantics are absolutely same as those of Harris's lock-free linked list [46] and which was further improved my Micheal [67]. A very sophisticated proof of the state change and thus validity of the list algorithm was provided by Micheal [67]. The invariant maintained our list, invariant 6.4, can be proved along the same lines and we skip the detail here. Now, we prove the linearizability of the implementation $\mathcal{I}_{\mathcal{M}}$ as given below.

Theorem 6.2. *(Correctness) The operations* ADD*,* REMOVE*,* CONTAINS *and* NNSEARCH *are linearizable.*

Proof. We show that a sequential history $\mathcal{S}$ obtained by following the steps: (a) in an arbitrary history $\mathcal{H}$ append appropriate response (in any arbitrary order) of all the operations which have performed their linearization steps as defined in definition 6.1 to obtain $ext(\mathcal{H})$, (b) drop the invocation steps without a matching response to obtain $complete(ext(\mathcal{H}))$, and (c) construct $\mathcal{S}$ by arranging the invocation-response pair of operations according to their linearization points, is consistent.

Let $\mathcal{S}_n$ be a sub-history of $\mathcal{S}$ that contains the *first* n complete operations. Let $\mathbb{A}_n$ be the dataset which was added to the LFkD-tree by the successful ADD operations in $\mathcal{S}_n$. Let $\mathbb{B}_n$ be the dataset which was removed from the LFkD-tree by the successful REMOVE operations in $\mathcal{S}_n$. Let $\mathbb{C}_n = \mathbb{A}_n/\mathbb{B}_n$. We use (strong) induction on n to show that $\mathcal{S}_n$ is consistent $\forall\ n{\geq}1$.

Suppose that $\mathcal{S}_n$ is consistent $\forall\ n\ :\ 1{\leq}n{\leq}i$. Let the $(i+1)^{th}$ operation in $\mathcal{S}_n$ be $op(k)$, where $k{\in}\mathbb{R}^d$. Then for $\mathcal{S}_{i+1}$ we prove the following:

1. Let $op(k)$ be an ADD operation.

   (a) Let $op(k)$ returns true. We show that if $op_1(k)$ is an ADD operation such that $op_1(k) \xrightarrow[\mathcal{S}_{i+1}]{} op(k)$ and $op_1(k)$ returns true then $\exists$ a REMOVE operation $op_2(k)$ such that $op_1(k) \xrightarrow[\mathcal{S}_{i+1}]{} op_2(k) \xrightarrow[\mathcal{S}_{i+1}]{} op(k)$ and $op_2(k)$ returns true.

Suppose there does not exist such a REMOVE operation. Now, following lemma 6.12, at the termination of `Search`, line 4 in the Algorithm 6.2, $pa \rightsquigarrow a$ is a leaf-node pointer. Now using the construction of $\mathcal{S}_i$ and definition 6.1-(1), at the linearization of op, it performed a successful `CAS` at the link $pa \rightsquigarrow a$ which must have been `clean`. Using the same argument op_1 also performed a successful `CAS` at the link $pa \rightsquigarrow a$ which must have been `clean`.

Now because op_1 linearized before op, the set of nodes that the `Search` called from op, terminates at, by the consistency of $\mathcal{S}_i$ op must find k being the key at that leaf-node. Now unless the link $pa \rightsquigarrow a$ was already injected with the descriptor `mark`, op would not have continued beyond the termination of `Search` and reading the descriptor at it and thereby returning false. Therefore, there must have been a REMOVE operation which marked the link $pa \rightsquigarrow a$ before op read and thus it had the linearization point before that of op. This is a contradiction.

(b) Let $op(k)$ returns false. We show that $\exists$ an ADD operation $op_1(k)$, which returns true, such that $op_1(k) \xrightarrow[\mathcal{S}_{i+1}]{} op(k)$ and $\nexists$ a REMOVE operation $op_2(k)$, which returns true, such that $op_1(k) \xrightarrow[\mathcal{S}_{i+1}]{} op_2(k) \xrightarrow[\mathcal{S}_{i+1}]{} op(k)$.

Suppose the contrary. Then at the termination of `Search`, line 4 in the Algorithm 6.2, by definition 6.1-(3) the link $pa \rightsquigarrow a$ is `clean` and $a{\cdot}\mathsf{k} = k$. But, following (a) as above and the consistency of $\mathcal{S}_i$, there must exist an $op_1(k)$ in $\mathcal{S}_i$ which returns true and that does not precede an $op_2(k)$ which returns true- which contradicts our assumption.

Now, it is easy to see that after the linearization of an ADD operation that returns true, the node added by it is reachable from root following the links and thus that node belongs to the LFkD-tree which in turn implies that $k \in \mathbb{C}_{i+1}$. Thus, combining this fact with (a) and (b) together, the mapping definition of ADD is satisfied. Thus, ADD is consistent in $\mathcal{S}_{i+1}$.

2. Let $op(k)$ be a REMOVE operation.

(a) Let $op(k)$ returns true. We show that if $op_1(k)$ is a REMOVE operation, which returns true, such that $op_1(k) \xrightarrow[\mathcal{S}_{i+1}]{} op(k)$ then $\exists$ an ADD operation $op_2(k)$, which returns true, such that $op_1(k) \xrightarrow[\mathcal{S}_{i+1}]{}$

$op_2(k) \xrightarrow[\mathcal{S}_{i+1}]{} op(k)$.

We use similar argument as given in (1) to prove it.

(b) Let $op(k)$ returns false. We show that one of the following is true:

i. If $op_1(k)$ is a REMOVE operation, which returns true, such that $op_1(k) \xrightarrow[\mathcal{S}_{i+1}]{} op(k)$ then $\nexists$ an ADD operation $op_2(k)$, which returns true, such that $op_1(k) \xrightarrow[\mathcal{S}_{i+1}]{} op_2(k) \xrightarrow[\mathcal{S}_{i+1}]{} op(k)$.

Suppose the contrary is true. Then, because $op_1(k)$ return true, by the construction of $\mathcal{S}_{i+1}$ and the definition of the linearization point definition 6.1-(2), either a leaf-node does not exist with key k or the link to it is injected with `mark`. Now if that is the case and op also returns true, then there must have been a link to a leaf-node with key k which was `clean`. But that was possible only if an ADD existed before op, which added a leaf-node with key k. This contradicts our claim.

ii. There $\nexists$ an ADD operation $op_1(k)$, which returns true, and $op_1(k) \xrightarrow[\mathcal{S}_{i+1}]{} op(k)$.

We can observe that at the linearization of $op(k)$, the link to the leaf-node with key k gets injected with `mark` and thus after that $k \notin \mathbb{C}_n$. Combining this fact with (a) and (b) satisfies the sequential specification of REMOVE. Thus, REMOVE is consistent in $\mathcal{S}_{i+1}$.

3. Let $op(k)$ be a CONTAINS operation.

(a) Let $op(k)$ returns true. We show that $\exists$ an ADD operation $op_1(k)$ such that $op_1(k) \xrightarrow[\mathcal{S}_{i+1}]{} op(k)$ and $\nexists$ a REMOVE operation $op_2(k)$ such that $op_1(k) \xrightarrow[\mathcal{S}_{i+1}]{} op_2(k) \xrightarrow[\mathcal{S}_{i+1}]{} op(k)$.

The arguments are similar to (1)(b) above.

(b) Let $op(k)$ returns false. We show that one of the following is true:

i. If $op_1(k)$ is a REMOVE operation, which returns true, such that $op_1(k) \xrightarrow[\mathcal{S}_{i+1}]{} op(k)$ then $\nexists$ an ADD operation $op_2(k)$, which returns true, such that $op_1(k) \xrightarrow[\mathcal{S}_{i+1}]{} op_2(k) \xrightarrow[\mathcal{S}_{i+1}]{} op(k)$.

ii. There $\nexists$ an ADD operation $op_1(k)$, which returns true, and $op_1(k) \xrightarrow[\mathcal{S}_{i+1}]{} op(k)$.

The arguments are similar to (2)(b) above. Combining (3)(a) and (3)(b), CONTAINS is consistent in $\mathcal{S}_{i+1}$.

4. Let $op(k)$ be a NNSEARCH operation that returns k^*. We show that (a) there $\exists$ $op_1(k^*)$ such that $op_1(k^*) \xrightarrow[\mathcal{S}_{i+1}]{} op(k)$ and (b) if there $\exists$ $op_1(k^{**})$, which returns true, where op_1 is either ADD or CONTAINS and $||k^{**},\ k||_2 < ||k^*,\ k||_2$ such that $op_1(k^{**}) \xrightarrow[\mathcal{S}_{i+1}]{} op(k)$ then there $\exists$ a REMOVE operation $op_2(k^{**})$, which returns true, such that $op_1(k^{**}) \xrightarrow[\mathcal{S}_{i+1}]{}$ $op_2(k^{**}) \xrightarrow[\mathcal{S}_{i+1}]{} op(k)$.

   To prove (a), it is easy to see that if such an ADD did not exist preceding op then at the linearization of op it can not read a leaf-node containing k^*. Therefore, (a) is true.

   Now, for (b), suppose the contrary is true. Thus, if there did not exist a REMOVE operation op_2 then at the linearization of op, which is either at the termination of the method `Seek` called by itself or at the termination of the method `Search` called by reporting CONTAINS or at the `CAS` step performed by a reporting ADD operation, the leaf-node containing k^{**} must have been connected by a `clean` link. But then either op would have read the `clean` link to the leaf-node with k^{**} or the operation reporting to it would have done the same. Thus the method `Process` that is called by NNSEARCH before its return, by virtue of $||k^{**},\ k||_2 < ||k^*,\ k||_2$, would have returned k^{**} which in turn would have been returned as the nearest neighbour of k by op. Which is a contradiction. Thus, NNSEARCH is consistent in $\mathcal{S}_{i+1}$.

By (1) to (4), $\mathcal{S}_{i+1}$ is consistent whenever $\mathcal{S}_n$ is consistent $\forall n : 1 \leq n \leq i$. Therefore, using (strong) induction, $\mathcal{S}_n$ is consistent for every positive integer n. □

Theorem 6.3. *(Lock-freedom) The LFkD-tree operations* ADD, REMOVE, CONTAINS *and* NNSEARCH *are lock-free and thus the presented algorithm implements a lock-free LFkD-tree.*

Proof. We take the NNSEARCH operation separately because it also involves the steps related to the lock-free list. By the description of the algorithm, a non-faulty thread performing a CONTAINS will always return unless its search path keeps on getting longer forever. If that happens, an infinite number of ADD operations would have successfully completed adding new nodes making the implementation lock-free. So, in the context of ADD, REMOVE and CONTAINS, it will suffice to prove that the modify operations are lock-free.

Suppose that a process $p \in \mathcal{P}$ performs a modify operation op on a valid state of LFkD-tree Υ_t and takes infinite steps and no other modify operation completes after that. Now, if no modify operation completes then Υ_t remains

unchanged forcing p to retract every time it wants to execute its own modification step on Υ_t. This is possible only if every time p finds the injection point of *op* with descriptor `mark`, `flag`, `ltag` or `rtag`. This implies that a REMOVE operation is pending. It is trivial to observe in the method ADD that if it gets obstructed by a concurrent REMOVE, then before retrying after recovery from failure, it helps the pending REMOVE by executing all the remaining steps of that. We can also observe that whenever two REMOVE operations obstruct each other, one finishes before the other. It implies that whenever two modify operations obstruct each other one finishes before the other and so Υ_t changes. It is contrary to our assumption. Hence, by contradiction we show that no non-faulty process shall remain taking infinite steps if no other non-faulty process makes progress where the executed operation is either ADD or REMOVE.

Now we consider a NNSEARCH with concurrent ADD, REMOVE or CONTAINS operations. We consider the case where concurrent NNSEARCH operations do not necessarily have coinciding target points; this case obviously covers the case when they do have coinciding target points. We can see that a REMOVE operation does not have to report to a concurrent NNSEARCH operation. Moreover, an ADD or a CONTAINS operation to perform a reporting, needs to first traverse through the unordered list and then possibly perform a `CAS` if required to report. Now, unless the number of NNSEARCH operations keep on increasing infinitely, the total length of the unordered list will be finite and thus the traversal path for an ADD or a CONTAINS operation to report will be finite. Now, at each neighbour-collector, where the reporting is required, if a `CAS` to report fails, that implies that a concurrent CONTAINS or ADD operation succeeds. Similarly, when a `CAS` by a NNSEARCH operation fails, it indicates that a `CAS` by a concurrent NNSEARCH operation succeeded. Finally, a `CAS` to add a new neighbour-collector only indicates that either a new neighbour-collector by a concurrent NNSEARCH has been successfully added or a NNSEARCH operation has terminated. In case of a `CAS` failure to add a new neighbour-collector, a NNSEARCH operation always helps a concurrent pending NNSEARCH operation before reattempting, in case it finds the link with descriptor `mark`. It shows that in all cases at least one non-faulty thread succeeds with respect to execute a NNSEARCH operation concurrent to any other LFkD-tree operation. Thus we arrive at the theorem 6.3. □

This concludes the proof of the presented algorithm.

6.5 A real-life application

Let us consider a web application that provides support for a real-time dynamic speed dating. The requirements of this application are as the following:

(a) Users join and leave dynamically.

(b) Users respond to a set of 5 multiple choice questions and based on the response their profile is created as a 5-tuple. A user is indexed by his/her profile.

(c) Users query for the most similar matching profile concurrently with profiles getting adding and removed.

(d) The application aims to utilize the multiple cores of a commonly available shared memory machine to get speed-up.

(e) In the fully asynchronous setting of the application, the concurrent operations must return consistent result. Additionally, progress guarantee is desired, that is, if multiple concurrent threads are assigned to the tasks of add, remove and similarity match queries by users, the application should tolerate any number of individual threads getting faulty.

We face many similar instances in our day-to-day experience with web based software. Given a 5-tuple $a{=}\{a_i\}_{i=1}^{5}$ representing the profile of a user querying similarity match, the problem here is to find the profile of a user, represented by $b{=}\{b_i\}_{i=1}^{5}$, such that $d(a,b){\leq}d(a,k)\ \forall\ k{=}\{k_i\}_{i=1}^{5}$, where $d()$ is a real-valued metric and k represents a 5-tuple corresponding to an *active* user. The problem becomes challenging for the dynamic nature of the application. Furthermore, desiring speed-up along with consistency and progress guarantee broadens the challenge.

Although the above problem statement is hypothetical but to our surprise we found that the sequential kD-tree used for throughput comparison in this work is perhaps being used in a similar web application as mentioned here `http://home.wlu.edu/~levys/software/kd/`. This clearly motivates our work which can most certainly speed up such an application with a provable progress guarantee.

6.6 Experimental Evaluation

6.6.1 Experimental Setup

We implemented the LFkD-tree algorithm in Java using RTTI. We used the library objects `AtomicReferenceFieldUpdater` to perform `CAS`. The test environment comprised a dual-socket server with a 2.0GHz Intel (R) Xeon (R) E5-2650 with 8 physical cores each (32 hardware threads in total with hyper-threading enabled). The server has 64 GB of RAM, runs Ubuntu 13.04 Linux (Kernel version: 3.8.0-35-generic x86_64) with Java HotSpot (TM) 64-Bit Server VM (build 25.60-b23), and we compiled all the implementations with `javac` version 1.8.0_60.

1. Levy-Kd: An implementation of kD-tree of [68] by Levy [63] that supports REMOVE operations (we could not find any other Java implementation of a kD-tree with REMOVE).

2. LFKD: Our implementation of the LFkD-tree with NNSEARCH.

3. LFKD(SC): Our implementation of the LFkD-tree with NNSEARCHRELAXED.

4. PH-tree: A multi-dimensional storage and indexing data structure by Zäschke *et al.* [85] that supports REMOVE operations. The implementation is single-threaded.

We run each test for 5 seconds and measured throughput as the total number of operations per microsecond executed by all threads in this time duration. We run each experiment in a separate instance of the JVM, starting off with a 2-second "warm-up" period to allow the Java HotSpot compiler to initialize and optimize the running code. During this warm-up phase, we performed random Add, Remove and Contains operations, and then flushed the tree at the end of the period. At the start of each execution, the data structure is pre-filled with a set of keys in the selected key-range.

To simulate the variation in contention and tree structure, we chose following combination of workload configurations: i) dataset space dimension $\in \{2, 3, 4, 5\}$, ii) number of key entries $\in \{\{0\text{-}10^6\}, \{0\text{-}10^7\}\}$, iii) distribution of (ADD-REMOVE-NNSEARCH) $\in \{(05, 05, 90), (25, 25, 50), (50, 50, 00)\}$, and iv) number of threads $\in \{1, 2, 4, 8, 16, 32\}$.

We did not include CONTAINS operations in experiment because essentially it would increase the proportion of exact-match NNSEARCH. All executions

use the same set of randomly generated points for the selected workload characteristics. The graphs present average of throughput over 6 runs of each experiment.

6.6.2 Datasets

We performed evaluation using a 2D real-world dataset and a set of synthetic benchmarks. For the real-world dataset, we used the United States Census Bureau 2010 TIGER/Line KML [21] dataset that consists of polylines describing map features of the United States of America. TIGER/Line is a standard dataset used for benchmarking spatial databases. For this evaluation, we extracted points representing the mainland, resulting in $18.4 * 10^6$ unique 2-dimensional points, with x-y coordinates that lie between -124.85 $\leqslant x \leqslant$ -66.89 and 24.40 $\leqslant y \leqslant$ 49.38 (ignoring the third dimension with all points 0.0).

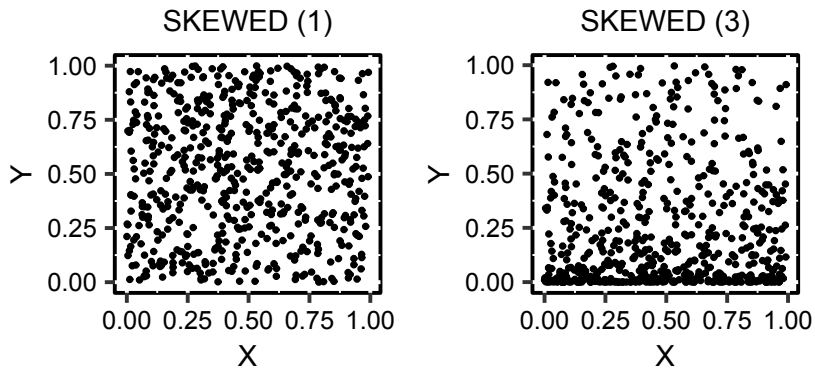


Fig. 6.3: Synthetic datasets: SKEWED(1) and SKEWED(3).

To investigate more extreme cases, two synthetic datasets were utilized. The SKEWED data simulates datasets in which different dimensions may have varying distributions. The SKEWED (c) dataset contains uniformly distributed points which fall within 0.0 and 1.0 in every dimension that have been skewed in the y-dimension. For each point in the dataset, the y value is replaced with the value y^c. In the Figure 6.3(a), we show examples for SKEWED(1) which is intuitively uniform distribution in all dimensions. SKEWED (3) and SKEWED (6) are shown in the Figure 6.3(b) and Figure 6.4(a), respectively .

The CLUSTER dataset [85] is an extension of a synthetic dataset previously described by Arge *et al.* [6]. In this evaluation we used clusters of 1000 points

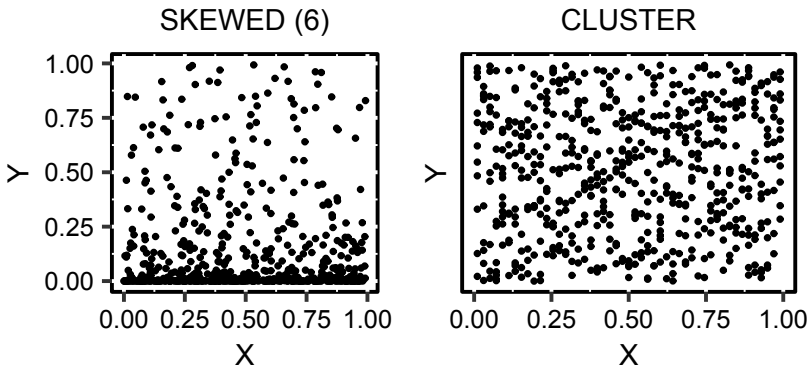


Fig. 6.4: Synthetic datasets: SKEWED(6) and CLUSTER.

evenly spaced on a horizontal line. Each of the clusters is filled with evenly distributed points and stretches 0.00001 in every dimension. Figure 6.4(b) depicts an example of the CLUSTER dataset with 49 points per cluster. The line of clusters falls within (0.0, 1.0) along the x-axis and is parallel to every other dimensional axis with a 0.5 offset. For this dataset, we generated up to 50,000,000 unique points.

6.6.3 Observations and Discussion

The Figures 6.5, 6.6 and 6.7 show the performance of the implementations for TIGER/Line, SKEWED and CLUSTER datasets respectively. In Figure 6.6 and 6.7, each row represents a combination of the range of key (k=N, N being the maximum) and the associated workload distribution while each column the dimensionality of key (d=dimension).

In all of them, LFKD and LFKD(SC) have higher performance compared to both the PH-tree and the Levy-Kd, even in single thread cases, for all workload distributions. The performance significantly scales up with increasing thread count. This shows that our implementation is both lightweight and scalable. As we increase the key dimension, the performance degrades for workloads dominated by the NNSEARCH. This degradation with increasing key dimensions is expected in kD-trees due to the *curse of dimensionality* [80]. This performance pattern is identical for different key ranges. However, the LFKD still achieve speed-up over the single threaded implementations.

We further observe that, as expected, LFKD(SC) outperforms LFKD in

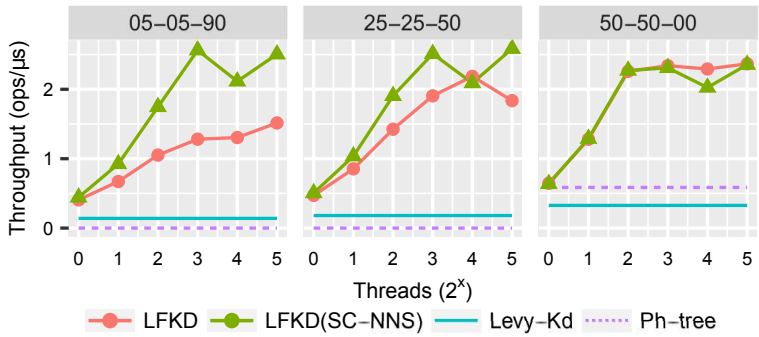


Fig. 6.5: Performance on the 2-D TIGER/Line dataset.

NNSEARCH dominated workload, however, the gap reduces with increasing dimensionality of the data set that brings the increased load of BFS traversal. This can be explained in terms of additional step complexity in account of reporting and maintaining the augmented lock-free list for linearizability. More importantly, it provides a significant exposition of NNSEARCH vis-a-vis consistency framework of a concurrent implementation: the overhead of linearizability, which is visible in a low dimension, gets subsumed by the cost of iterative scan, which visits almost every node of the kD-tree as the dimension increases.

For the TIGER/Line dataset, in a single thread case, both LFKD and LFKD(SC) perform at least $2.5\times$ better than Levy-Kd, and, it goes up to $19\times$ in the NNSEARCH dominated workload. Additionally, the PH-tree outperforms the Levy-Kd only for workloads that do not involve NNSEARCH (00% NNSEARCH, 50% ADD and 50% REMOVE).

We observe that for NNSEARCH dominated workload (90% NNSEARCH, 5% ADD and 5% REMOVE), the LFKD(SC) achieves speed-ups up to $66\times$ for SKEWED and up to $150\times$ for CLUSTER datasets over the sequential implementations. These observations can be partially attributed to the local-midpoint rule, which carries the essence of the sliding-midpoint-splitting rule of [70] that targets the extreme cases such as a CLUSTER dataset, to a concurrent setting.

For a mixed workload (50% NNSEARCH, 25% ADD and 25% REMOVE), the performance of LFkD-tree degrades by increasing key dimension. The absolute throughput figures are higher for the NNSEARCH dominated workload in lower dimensions than in mixed workloads. This is because the modify operations incur higher synchronization (conflicts, expensive atomic operations, and helping) overhead. However in higher dimensions, the throughput of the NNSEARCH is lower as the number of visited nodes increases tremendously

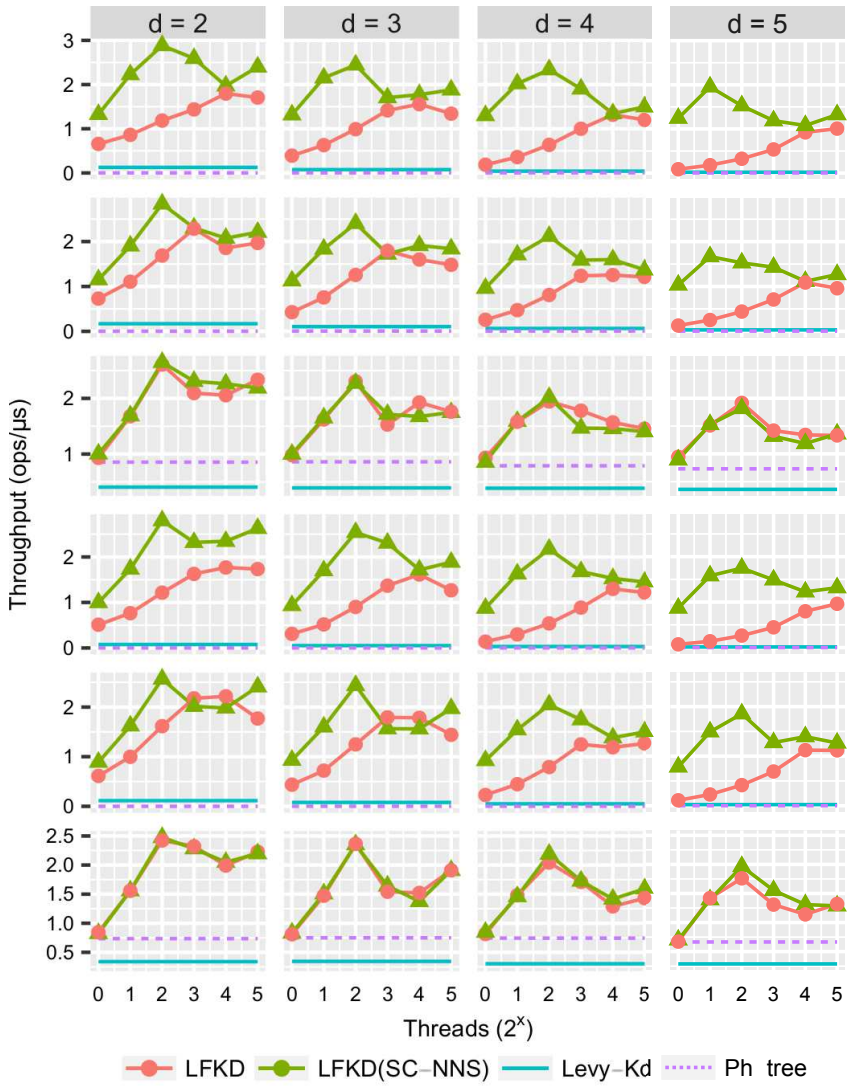


Fig. 6.6: Performance on the SKEWED(6) dataset.

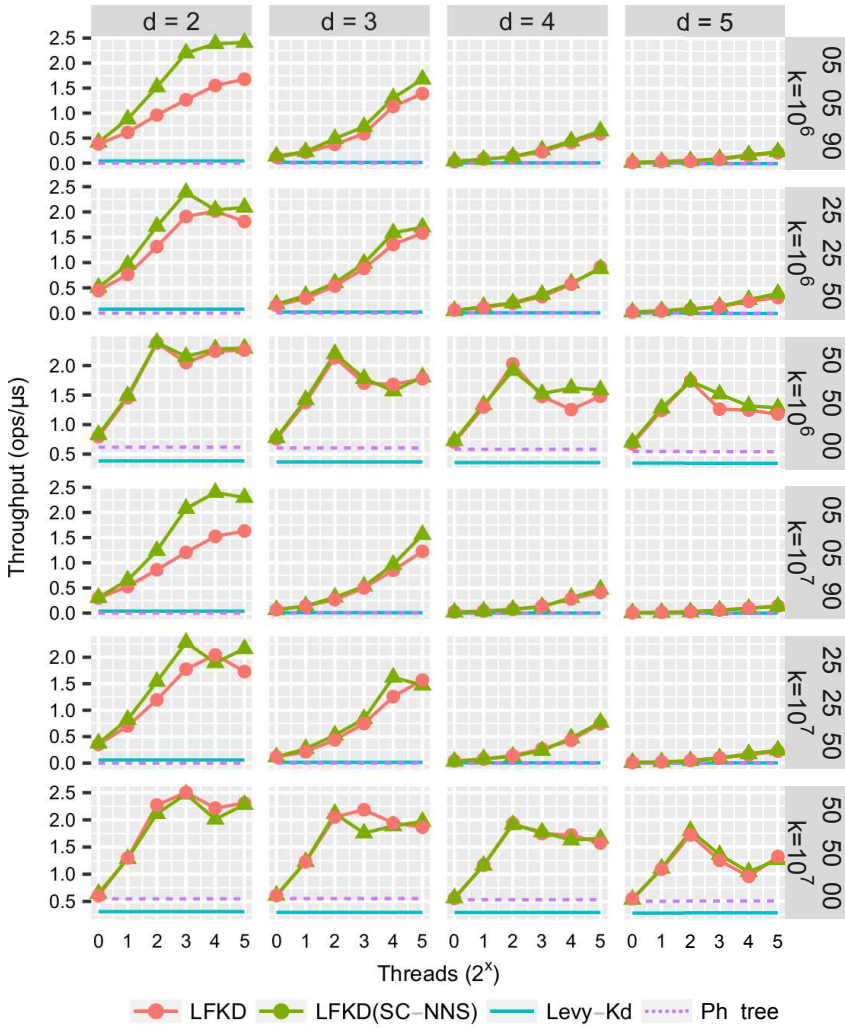


Fig. 6.7: Performance on the CLUSTER dataset.

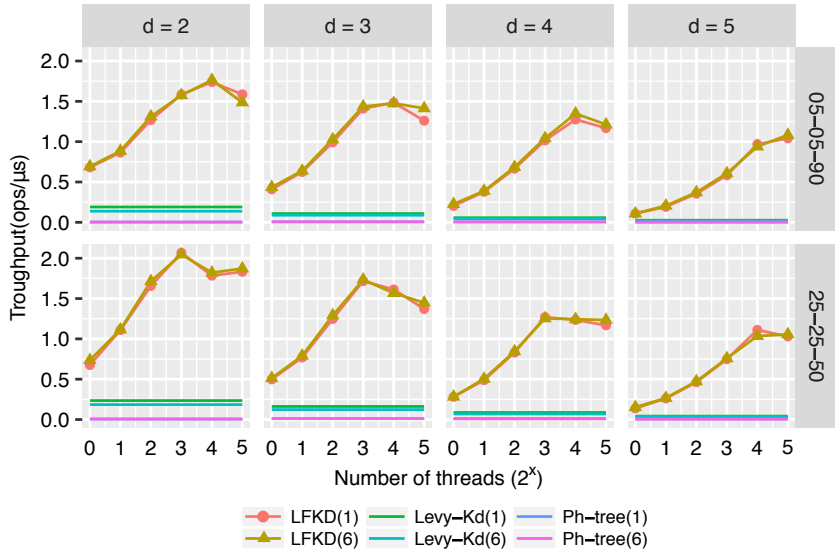


Fig. 6.8: System throughput for SKEWED(1) and SKEWED(6) datasets

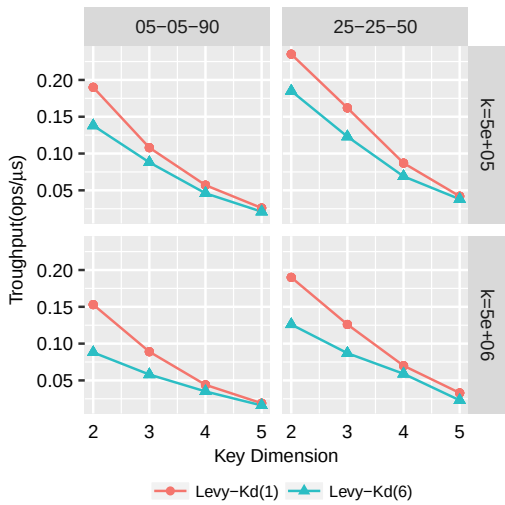


Fig. 6.9: System throughput for Levy-kd.

with dimension.

We also observed that the skewness of data does not affect the performance of the LFKD, see the Figure 6.8. On the contrary, as depicted in the Figure 6.9 the throughput performance for the Levy-Kd drops as we increase the skewness of the data. The observed different behaviour can be attributed mainly to the local-midpoint splitting rule in the concurrent setting.

Chapter Summary

For a large number of applications, which require a multidimensional data structure supporting dynamic modifications along with nearest neighbour search, research community has largely focused on improving the design of sequential data structures. Parallel implementations of the sequential designs focus on speeding up the loading of the data and then NNS on a fully loaded data structure. Thus, they do not address the issue of dynamic modifications in the datasets. On the other hand, the concurrent data structure research is primarily confined to one-dimensional problems.

Our work is the first to extend the concurrent data structures to problems covering multidimensional datasets. We introduced LFkD-tree, a lock-free design of kD-tree, which supports linearizable nearest neighbour search operations with concurrent dynamic addition and removal of data. We provided a sample implementation which shows that the LFkD-tree algorithm is highly scalable.

Our method to implement linearizable nearest neighbour search is generic and can be adapted to other multidimensional data structures. We plan to design lock-free data structures which are suitable for nearest neighbour search in high dimensions, for example, the ball-tree [64]. We also plan to extend our work to k-nearest neighbour (kNN) search.

Part V

CONCLUSION

7

GENERAL CONCLUSIONS AND DISCUSSION

Chapter Abstract

In this final chapter, we discuss the goals and the achievements of this Ph.D. work. Moreover, we present some possible future directions to which this work can be extended.

7.1 Goals and the main findings

The goal of this Ph.D. work was to explore the following research directions:

1. Design and analysis of efficient concurrent data structures, in particular, the non-blocking algorithms that provide progress guarantee.

2. Scalable implementation of the concurrent data structure algorithms.

3. Design optimization of the lock-free algorithms.

4. Exploring the adaptability of the concurrent data structures to heterogeneous computing platforms such as GPUs.

When we started, the state of the art in the non-blocking concurrent data structures were mainly lock-free queues, dequeue, linked-list, and skip-list. We realized that the efficient non-blocking data structure for search problems could be a significant contribution of our research. In pursuit of the same, this thesis contributes the following:

1. We presented efficient algorithms for lock-free linearizable point search in one-dimensional datasets [26]. We presented novel lock-free algorithms for linked-lists and BSTs [24] that competed well the existing state of the art.

2. We presented a novel lock-free linearizable range search algorithm for one-dimensional datasets [23]. The range search algorithm is generic, that is, any data structure that supports predecessor query can seamlessly adopt the presented methodology. The previously existing solutions for lock-free linearizable range search were tightly associated with the data structure, in this case, k-ary search trees.

3. Our algorithms for lock-free linked-lists and BSTs [26] improve on the existing similar algorithms in the following way:

   (a) Our algorithms are language portable, that is, they do not use language-specific constructs for an implementation. Specifically, this methodology demonstrates that a fully object-oriented design of a lock-free data structure that does not utilize any pointer manipulation, which is typically suitable for an implementation in Java, is equally competitive in C/C++.

   (b) The presented algorithms take the optimization of step complexity as a very important aspect of the design. In that direction, we present the notion of help-optimality that captures the wastage of costly `CAS` execution steps. Thereby, minimizing such wastage makes an algorithm more efficient.

4. We presented the first complete design for a lock-free kD-tree [25]. Our lock-free algorithm for a linearizable nearest neighbour search is the first in its category. Further, the nearest neighbour search algorithm for the multidimensional datasets is independent of the underlying data structure. To the best of our knowledge, ours is the first algorithm for a lock-free multi-dimensional data structure. Given that the application of multidimensional search is at the center of many contemporary applications, our work attempts to align scalable lock-free concurrent data structures with the contemporary computers.

In addition to the above contributions, we explored the adaptability of concurrent data structures to GPUs, which we did not include in this thesis to keep its theme focused on the lock-free search data structures. An interested reader may refer to the paper VI listed in the Contents and Publications [19]. This work was the first that implemented any concurrent data structure on the GPUs.

7.2 Further direction

The research work presented in this thesis can be carried forward in a number of interesting directions.

1. The lock-free linearizable range search algorithm for one-dimensional datasets can naturally be extended to the multidimensional datasets. Towards that, we aim to explore an efficient lock-free design of data structures like R-tree with range search. Another work in the same direction will be to design a lock-free version of the multidimensional hash table.

2. Traditionally, the trade-off between an approximate solution and the performance of a search problem has attracted intense attention of the research community. On that point, the trade-off between consistency frameworks (linearizability, sequential consistency, etc.) and approximate solution is an interesting issue to explore.

3. From an implementation point of view, implementing our language-portable lock-free design in a high-level programming language such as Go is an interesting future work. Go is a programming language that provides pointers without pointer arithmetic and has a restricted object inheritance support.

4. The complexity analysis of concurrent non-blocking data structures definitely entails efficient methodologies. Instead of translating the methods applied in sequential data structures, an entirely new and innovative approach, which keeps the main focus on concurrency, can be an important future work. The literature of the lower bound analysis of sequential search data structures is vast. At the same time, in wait-free data structures, where every operation finishes in a finite number of steps, there are works in literature to compute lower bounds. Mostly the lower bound computation in such wait-free data structures assumes that the dataset is finite. For example, see [59]. With that assumption, even in lock-free data structures, computation of lower bound can be largely simplified. We plan to work on lower bound of the lock-free search algorithms.

7.3 Some reflections

Today's computing systems are very different from the ones that existed just a decade ago. Multi-core CPUs are commonplace and heterogeneous systems have replaced usual workstations. Processing is fast migrating to big distributed

systems that are the constellations of commodity multi-core computers. Simultaneously, data is getting generated by an ever-increasing number of interconnected devices at a pace that was previously unimagined. Applications are constantly in a rush to get information out of the data, which obviously happens by way of a variety of real-time search queries. Even in big distributed systems, with "big data" platforms running on them, the real-time information processing takes place in main-memory data structures.

When we started this Ph.D. work, the above-mentioned scenario was fast evolving. Unlike today, when every smart-phone is equipped with a GPU, high-end GPUs were costly and mainly found in the workstations. At such a time my contribution to the research project "Software Abstractions for Heterogeneous Computers (SCHEME)" was envisaged as a library of concurrent data structures for multi-core CPUs as well as many-core GPUs. We started with implementing existing concurrent queue algorithms on GPUs and had a quick publication [19] too. However, soon we realized that designing new efficient concurrent data structures entailed a much greater significance and implementation on GPUs could follow. Thus, given the demand for efficient real-time search problems we soon converged at a collective and focused study of lock-free search data structures.

While writing this thesis after five long years, I am definitely satisfied with the detailed study presented in this work. The new lock-free data structures for point search and range search in one-dimensional data and nearest neighbour search in multi-dimensional data take steps in the pursuit of our envisioned research goals. I wish I could have contributed more to this exciting area of research. I am surely interested to continue this work in the areas mentioned as the future directions above and in many more upcoming computer science research domains in the time to come.

7

BIBLIOGRAPHY

[1] M AdelsonVelskii and E. M. Landis. An algorithm for the organization of information. Technical report, DTIC Document, 1963.

[2] Y. Afek, H. Attiya, D. Dolev, E. Gafni, M. Merritt, and N. Shavit. Atomic snapshots of shared memory. *Journal of the ACM (JACM)*, 40(4):873–890, 1993.

[3] Y. Afek, G. Stupp, and D. Touitou. Long-lived adaptive collect with applications. In *40th Annual Symposium on Foundations of Computer Science*, pages 262–272. IEEE, 1999.

[4] A. V. Aho and J. E. Hopcroft. *The design and analysis of computer algorithms*. Pearson Education India, 1974.

[5] R. Alur and G. Taubenfeld. Results about fast mutual exclusion. In *Real-Time Systems Symposium, 1992*, pages 12–21. IEEE, 1992.

[6] L. Arge, M. D. Berg, H. Haverkort, and K. Yi. The priority r-tree: a practically efficient and worst-case optimal r-tree. *ACM Trans. Algorithms*, 4(1):9:1–9:30, 2008. ISSN: 1549-6325.

[7] S. Arya and H.-Y. A. Fu. Expected-case complexity of approximate nearest neighbor searching. *SIAM Journal on Computing*, 32(3):793–815, 2003.

[8] H. Attiya and A. Fouren. Algorithms adapting to point contention. *Journal of the ACM*, 50(4):444–468, 2003.

[9] H. Attiya, R. Guerraoui, and E. Ruppert. Partial snapshot objects. In *Proceedings of the 20thannual symposium on Parallelism in algorithms and architectures*, pages 336–343. ACM, 2008.

[10] H. Avni, N. Shavit, and A. Suissa. Leaplist: lessons learned in designing tm-supported range queries. In *Proceedings of the 32nd ACM symposium on Principles of distributed computing*, pages 299–308. ACM, 2013.

[11] G. Barnes. A method for implementing lock-free shared-data structures. In *Proceedings of the fifth annual ACM symposium on Parallel algorithms and architectures*, pages 261–270. ACM, 1993.

[12] D. Basin, E. Bortnikov, A. Braginsky, G. Golan-Gueta, E. Hillel, I. Keidar, and M. Sulamy. Kiwi: a key-value map for scalable real-time analytics. In *Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 357–369. ACM, 2017.

[13] R. Bayer and E. McCreight. Organization and maintenance of large ordered indexes. In *Software pioneers*, pages 245–262. Springer, 2002.

[14] J. L. Bentley. Multidimensional binary search trees used for associative searching. *CACM*, 18(9):509–517, 1975.

[15] N. G. Bronson, J. Casper, H. Chafi, and K. Olukotun. A practical concurrent binary search tree. In *ACM Sigplan Notices*, volume 45 of number 5, pages 257–268. ACM, 2010.

[16] T. Brown and H. Avni. Range queries in non-blocking k-ary search trees. In *16th International Conference On Principles Of Distributed Systems*, pages 31–45. Springer, 2012.

[17] T. Brown, F. Ellen, and E. Ruppert. A general technique for non-blocking trees. In *ACM SIGPLAN Notices*, volume 49 of number 8, pages 329–342. ACM, 2014.

[18] D. Cederman, B. Chatterjee, N. Nguyen, Y. Nikolakopoulos, M. Papatriantafilou, and P. Tsigas. A study of the behavior of synchronization methods in commonly used languages and systems. In *27th IEEE International Parallel & Distributed Processing Symposium*, pages 1309–1320, 2013.

[19] D. Cederman, B. Chatterjee, and P. Tsigas. Understanding the performance of concurrent data structures on graphics processors. *Euro-Par 2012 Parallel Processing*:883–894, 2012.

[20] K. Censor-Hillel, E. Petrank, and S. Timnat. Help! In *Proceedings of the 2015 ACM Symposium on Principles of Distributed Computing*, pages 241–250. ACM, 2015.

[21] U. Census. Tiger/line dataset. In `https://www.census.gov/geo/maps-data/data/tiger.html`, 2017.

[22] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: a distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)*, 26(2):4, 2008.

[23] B. Chatterjee. Lock-free linearizable 1-dimensional range queries. In *Proceedings of the 18th International Conference on Distributed Computing and Networking*, page 9. ACM, 2017.

[24] B. Chatterjee, N. Nguyen, and P. Tsigas. Efficient lock-free binary search trees. In *Proceedings of the 2014 ACM Symposium on Principles of Distributed Computing*, PODC '14, pages 322–331. ACM, 2014.

[25] B. Chatterjee, I. Walulya, and P. Tsigas. Concurrent linearizable nearest neighbour search in lockfree-kd-tree. In *Proceedings of the 19th International Conference on Distributed Computing and Networking*. ACM, 2018.

[26] B. Chatterjee, I. Walulya, and P. Tsigas. Help-optimal and language-portable lock-free concurrent data structures. In *Parallel Processing (ICPP), 2016 45th International Conference on*, pages 360–369. IEEE, 2016.

[27] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to algorithms*. MIT press Cambridge, 2001.

[28] T. Crain, V. Gramoli, and M. Raynal. A speculation- friendly binary search tree. In *Proceedings of the 17th ACM PPoPP*, pages 161–170, 2012.

[29] T. David, R. Guerraoui, and V. Trigonakis. Asynchronized concurrency: the secret to scaling concurrent search data structures. In *ACM SIGARCH Computer Architecture News*, volume 43 of number 1, pages 631–644. ACM, 2015.

[30] D. Drachsler, M. Vechev, and E. Yahav. Practical concurrent binary search trees via logical ordering. In volume 49 of number 8, pages 343–356. ACM, 2014.

[31] M. Dubois and C. Scheurich. Memory access dependencies in shared-memory multiprocessors. *Software Engineering, IEEE Transactions on*, 16(6):660–673, 1990.

[32] F. Ellen, P. Fatourou, J. Helga, and E. Rupert. The amortized complexity of non-blocking binary search trees. In *33rd ACM Symposium on Principles of Distributed Computing*, pages 332–341, 2014.

[33] F. Ellen, P. Fatourou, E. Ruppert, and F. v. Breugel. Non-blocking binary search trees. In *29th ACM Symposium on Principles of Distributed Computing*, pages 131–140, 2010.

[34] M. Er. Efficient generation of k-ary trees in natural order. *The Computer Journal*, 35(3):306–308, 1992.

[35] M. Fomitchev and E. Ruppert. Lock-free linked lists and skip lists. In *Proceedings of the twenty-third annual ACM symposium on Principles of distributed computing*, pages 50–59. ACM, 2004.

[36] K. Fraser. *Practical lock-freedom*. PhD thesis, Cambridge University, Computer Laboratory, 2004.

[37] J. H. Friedman, J. L. Bentley, and R. A. Finkel. An algorithm for finding best matches in logarithmic expected time. *Transactions on Mathematical Software (TOMS)*, 3(3):209–226, 1977.

[38] S. Ghemawat and P. Menage. Tcmalloc : thread-caching malloc. `http://goog-perftools.sourceforge.net/doc/tcmalloc.html`, 2017.

[39] J. Gibson and V. Gramoli. Why non-blocking operations should be selfish. In *Distributed Computing*, pages 200–214. Springer, 2015.

[40] A. Gidenstam, M. Papatriantafilou, H. Sundell, and P. Tsigas. Efficient and reliable lock-free memory reclamation based on reference counting. *IEEE Transactions on Parallel and Distributed Systems*, 20(8):1173–1187, 2009.

[41] M. T. Goodrich and R. Tamassia. *Data structures and algorithms in Java*. John Wiley & Sons, 2008.

[42] V. Gramoli. More than you ever wanted to know about synchronization: synchrobench, measuring the impact of the synchronization on concurrent algorithms. 50(8):1–10, 2015.

[43] L. J. Guibas and R. Sedgewick. A dichromatic framework for balanced trees. In *Foundations of Computer Science, 1978., 19th Annual Symposium on*, pages 8–21. IEEE, 1978.

[44] A. Guttman. *R-trees: a dynamic index structure for spatial searching*, volume 14 of number 2. ACM, 1984.

[45] B. Haeupler, S. Sen, and R. E. Tarjan. Rank-balanced trees. In *WADS*, pages 351–362. Springer, 2009.

[46] T. L. Harris. A pragmatic implementation of non-blocking linked-lists. In *Distributed Computing*, pages 300–314. Springer, 2001.

[47] HBase. A distributed database for large datasets. *The Apache Software Foundation*, 4(4.2), 2017.

[48] S. Heller, M. Herlihy, V. Luchangco, M. Moir, W. N. S. Iii, and N. Shavit. A Lazy Concurrent List-Based Set Algorithm. In *9th International Conference On Principles Of Distributed Systems*, pages 3–16. 2005.

[49] J. Hennessy, N. Jouppi, S. Przybylski, C. Rowen, T. Gross, F. Baskett, and J. Gill. Mips: a microprocessor architecture. In *ACM SIGMICRO Newsletter*, volume 13 of number 4, pages 17–22. IEEE Press, 1982.

[50] M. P. Herlihy and J. M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(3):463–492, 1990.

[51] M. Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13(1):124–149, 1991.

[52] M. Herlihy, V. Luchangco, and M. Moir. Space-and time-adaptive nonblocking algorithms. *Electronic Notes in Theoretical Computer Science*, 78:260–280, 2003.

[53] S. V. Howley and J. Jones. A non-blocking internal binary search tree. In *24th ACM Symposium on Parallelism in Algorithms and Architectures*, pages 161–171, 2012.

[54] J. Ichnowski and R. Alterovitz. Scalable multicore motion planning using lock-free concurrency. *Robotics, IEEE Transactions on*, 30(5):1123–1136, 2014.

[55] D. Imbs and M. Raynal. Help when needed, but no more: efficient read/write partial snapshot. *Journal of Parallel and Distributed Computing*, 72(1):1–12, 2012.

[56] P. Indyk, R. Motwani, P. Raghavan, and S. Vempala. Locality-preserving hashing in multidimensional spaces. In *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*, pages 618–625. ACM, 1997.

[57] A. Israeli and L. Rappoport. Disjoint-access-parallel implementations of strong shared memory primitives. In *13th ACM Symposium on Principles of Distributed Computing*, pages 151–160, 1994.

[58] P. Jayanti. An optimal multi-writer snapshot algorithm. In *37th Annual ACM Symposium on Theory of Computing*, pages 723–732, 2005.

[59] S. V. Jayanti and R. E. Tarjan. A randomized concurrent algorithm for disjoint set union. In *Proceedings of the 2016 ACM Symposium on Principles of Distributed Computing*, pages 75–82. ACM, 2016.

[60] D. E. Knuth. *The art of computer programming: sorting and searching*, volume 3. Pearson Education, 1998.

[61] L. Lamport. Proving the correctness of multiprocess programs. *IEEE transactions on software engineering*, (2):125–143, 1977.

[62] D. Lea. `ConcurrentSkipListMap`. In *`java.util.concurrent`*, 2017.

[63] S. D. Levy. `KDTree`. In *`edu.wlu.cs.levy.CG.KDTree`*, 2017.

[64] T. Liu, A. W. Moore, and A. Gray. New algorithms for efficient high-dimensional nonparametric classification. *Journal of Machine Learning Research*, 7(Jun):1135–1158, 2006.

[65] J. W. Mauchly. Sorting and collating. *Theory and Techniques for Design of Electronic Digital Computers*:271–287, 1946.

[66] M. M. Michael and M. L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *15th ACM Symposium on Principles of Distributed Computing*, pages 267–275, 1996.

[67] M. M. Michael. High performance dynamic lock-free hash tables and list-based sets. In *Proceedings of the fourteenth annual ACM symposium on Parallel algorithms and architectures*, pages 73–82. ACM, 2002.

[68] A. W. Moore. Efficient memory-based learning for robot control. Technical report 209, University of Cambridge, 1991.

[69] G. M. Morton. *A computer oriented geodetic data base and a new technique in file sequencing*. International Business Machines Company New York, 1966.

[70] D. M. Mount and S. Arya. Ann: a library for approximate nearest neighbor searching. *`http://www.cs.umd.edu/~mount/ANN/`*, 2017.

[71] A. Natarajan and N. Mittal. Fast concurrent lock-free binary search trees. In *ACM SIGPLAN Notices*, volume 49 of number 8, pages 317–328. ACM, 2014.

[72] N. Nguyen, P. Tsigas, and H. Sundell. Parmarksplit: a parallel mark-split garbage collector based on a lock-free skip-list. In *International Conference on Principles of Distributed Systems*, pages 372–387. Springer, 2014.

[73] Oracle. Java.util.concurrent. In *`https://docs.oracle.com/javase/8/docs/api/`*, 2017.

[74] R. Oshman and N. Shavit. The skiptrie: low-depth concurrent search without rebalancing. In *Proceedings of the 2013 ACM Symposium on Principles of Distributed Computing*, PODC '13, pages 23–32. ACM, 2013.

[75] A. J. Perlis and C. Thornton. Symbol manipulation by threaded lists. *Communications of the ACM*, 3(4):195–204, 1960.

[76] E. Petrank and S. Timnat. Lock-free data-structure iterators. In *Distributed Computing*, pages 224–238. Springer, 2013.

[77] A. Prokopec, N. G. Bronson, P. Bagwell, and M. Odersky. Concurrent tries with efficient non-blocking snapshots. In *Acm Sigplan Notices*, volume 47 of number 8, pages 151–160. ACM, 2012.

[78] A. Ramachandran and N. Mittal. A fast lock-free internal binary search tree. In *Proceedings of the 2015 International Conference on Distributed Computing and Networking*, page 37. ACM, 2015.

[79] K. F. Sagonas and K. Winblad. Efficient support for range queries and range updates using contention adapting search trees. In *Languages and Compilers for Parallel Computing - 28th International Workshop, LCPC 2015, Raleigh, NC, USA, September 9-11, 2015, Revised Selected Papers*, pages 37–53, 2015.

[80] H. Samet. *Foundations of multidimensional and metric data structures*. Morgan Kaufmann, 2006.

[81] H. Sundell and P. Tsigas. Fast and lock-free concurrent priority queues for multi-thread systems. *Journal of Parallel and Distributed Computing*, 65(5):609–627, 2005.

[82] H. Sundell and P. Tsigas. Lock-free and practical doubly linked list-based deques using single-word compare-and-swap. In *9th International Conference On Principles Of Distributed Systems*, pages 240–255. Springer, 2005.

[83] S. Timnat, A. Braginsky, A. Kogan, and E. Petrank. Wait-free linked-lists. In R. Baldoni, P. Flocchini, and R. Binoy, editors, *Principles of Distributed Systems*. Volume 7702, LNCS. Springer Berlin Heidelberg, 2012.

[84] J. D. Valois. Lock-free linked lists using compare-and-swap. In *Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing*, pages 214–222. ACM, 1995.

[85] T. Zäschke, C. Zimmerli, and M. C. Norrie. The ph-tree: a space-efficient storage structure and multi-dimensional index. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pages 397–408. ACM, 2014.

[86] K. Zhou, Q. Hou, R. Wang, and B. Guo. Real-time kd-tree construction on graphics hardware. *ACM Transactions on Graphics (TOG)*, 27(5):126, 2008.