Technical Report no. 2015-08

Concurrent Linearizable Nearest Neighbour Search in LockFree-kD-tree

Bapi Chatterjee *Ivan Walulya* *Philippas Tsigas*

CHALMERS | GÖTEBORG UNIVERSITY

Department of Computer Science and Engineering
Chalmers University of Technology and Göteborg University
SE-412 96 Göteborg, Sweden

Göteborg, 2015

Technical Report in Computer Science and Engineering at
Chalmers University of Technology and Göteborg University

Technical Report no. 2015-08
ISSN: 1652-926X

Department of Computer Science and Engineering
Chalmers University of Technology and Göteborg University
SE-412 96 Göteborg, Sweden

Göteborg, Sweden, 2016

Abstract

The Nearest neighbour search (NNS) is an important problem in a large number of application domains dealing with multidimensional data. In concurrent settings, where dynamic modifications are allowed, a linearizable implementation of NNS is highly desirable to discover the latest nearest neighbour of a given target data-point.

In this paper, we introduce 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 NNS. Our implementation is linearizable. The operations in the LFkD-tree use single-word read and compare-and-swap (`CAS`) atomic primitives, which are readily supported on commonly available multi-core processors. 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.

Keywords: concurrent data structure, kD-tree, nearest neighbor search, similarity search, lock-free, linearizability

1 Introduction

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 numerous application domains including data mining, information retrieval, machine learning, robotics and image search. A variety of data structures, which store multidimensional points, are available in the literature to solve the NNS in a sequential setting. An excellent collection can be found in Samet's book [1]. Several of them have been adapted to perform parallel NNS over a static data structure, e.g. [2]. However, both sequential and parallel designs primarily consider NNS queries without accommodating dynamic addition or removal (together called modification) in the data structure. Chan *et al.* [3] presented a randomized data structure that provides worst-case $O(\log^2 n)$ NNS for two-dimensional data, subjected to dynamic modifications. Nevertheless, efficient multidimensional data structures for NNS admitting dynamic modifications have attracted limited attention as yet.

The growing availability of multi-core processors has brought about an emphasis on designing scalable concurrent data structures. Furthermore, the surge in the popularity of in-memory databases has led to significant interest in the index structures that can support NNS with dynamic concurrent add and remove operations. In Appendix A, we narrate an interesting real life application which requires a concurrent dynamic data structure for real-time NNS queries in multidimensional datasets.

Considering operations in a concurrent data structure, linearizability [4] is the hallmark for consistency. It relates the *externally observable* behaviour to the sequential specification of an operation. Intuitively, a concurrent data structure is linearizable if every execution provides time-points, called *linearization points*, between the invocation and response of each operation, where it seems to take effect instantaneously. Thus, forming a sequence of concurrent operations, described by the *real-time order* of the linearization points, we observe that concurrent operations meet their sequential specifications. A linearizable NNS is highly desirable to discover the latest point available in the dataset in a global real-time order which is the nearest neighbour of the target point.

In real life applications that employ NNS, most often the notion of distance is a metric which satisfies triangle inequality, for example, Hamming distance for string data, Hausdorff distance for image data and Euclidean distance for real number data. In one-dimensional metric spaces, the triangle inequality $(d(x, y) \leq d(z, x) + d(z, y))$ holds as an equality. Consequently, arrangements of one-dimensional data-points in the sorted order of their (signed) distance from different arbitrary reference points are same. Given an ordered data structure (e.g. sorted list, search tree, etc.) of one-dimensional points, on termination of a search query, we can always find two consecutive nodes which store the two nearest neighbours of the target point. In linked-lists and skip-lists it is trivial and in BSTs requires maintaining a successor link. Thus, performing an NNS in a one-dimensional dataset is analogous to a general exact match point query.

However, in multidimensional metric spaces, the triangle inequality is strict for non-collinear points. Hence, the arrangements of data-points in sorted order of distance from different arbitrary reference points are not same. Consequently, we do not have a multidimensional "ordered" data structures. 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 the dataset which do not contain the 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 latest 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 in the data structure faces an inescapable challenge. Consider the case of an operation *op* performing an NNS query in a 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 data-points 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 concurrently, and possibly before *op* read node containing k^*, added to the data structure in a subset which may already have been visited or got pruned by the completion of the latest iteration step. Thus, *op* would not visit that subset again. Now, if k^{**} was closer to a compared to k^*, and *op* returns k^*, which was possibly not the nearest neighbour of a when read by *op*, we can not determine a linearization point between the invocation and return of *op*. Clearly, the iterative scan method used in a sequential setting, which usually is implemented using a recursive tree traversal, is not directly adaptable to a concurrent implementation with linearizability.

Now considering the modify operations in a concurrent data structure, the traditional approach to accommodate them is by way of mutual exclusion using locks. But, in an asynchronous shared-memory system, where an infinite delay or crash failure of a thread 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, threads do not hold locks and at least one non-faulty thread is guaranteed to finish its operation in a finite number of steps. Therefore, lock-free data structures foster both scalability and fault tolerance.

In recent years, a number of practical lock-free search data structures have been designed: skip-lists [5,6], binary search trees (BSTs) [7–10], k-ary search tree [11], B+tree [12], 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.

One of the most commonly used multidimensional data structures for NNS is the kD-tree, introduced by Bentley [13]. In principle, a kD-tree is a generalization of the BST to store multidimensional data. Friedmann et al. [14] proved that a kD-tree can process an NNS in expected logarithmic time assuming uniformly distributed data points. Successively, many efforts, including the approximate solutions, have contributed to improving the performance of NNS in kD-trees [15–18]. Furthermore, various parallel kD-tree implementations have been presented, specifically in the computer graphics community, where they focus on accelerating the applications such as ray tracing in single-instruction-multiple-data (SIMD) programming model [19–22]. Nonetheless, these designs do not fit in a concurrent setting where we desire linearizable NNS with concurrent addition and removal of data. For robotic motion planning, Ichnowski *et al.* [23] 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 NNS is not linearizable. We note that the list of lock-free one-dimensional search data structures includes a number of BSTs [7–10]. We can use a lock-free BST as a foundation to design a lock-free kD-tree.

Contributions: We describe a linearizable implementation of an abstract data type (ADT) that provides ADD, REMOVE, CONTAINS and NNS operations for a multidimensional dataset. 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. 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 [24].

1.2 A high-level summary of the work

The main challenge in implementing a linearizable NNS is to ensure that it does not remain oblivious to concurrent modifications in the data structure. Please note that; an iterative scan of a data structure, because of the involved (often aggressive*) pruning, may not necessarily be obtaining a *snapshot*: an atomic view of all the nodes currently present, but still involves "scan". In the literature, there exist algorithms to implement shared wait-free multi-writer multi-word *snapshot objects* which support concurrent `scan` and `update` operations [25–29]. An `update` writes a new value at a word in shared memory, and `scan` returns an atomic view of all the words. These algorithms provide a synchronization method, which ensures that at the linearization a `scan` does not "miss" any linearized concurrent `update` in the multi-word object to reflect in its output. However, these algorithms do not offer themselves to be directly emulated in lock-free data structures. The main reason is that they do not provide a direct "`read`" operation which could be emulated to implement linearizable CONTAINS operations, without using the costly `scan`.

In general, the existing concurrent data structures do not support atomic snapshots. Exceptions are - lock-based BST by Bronson et al. [30] and lock-free Trie by Prokopec et al. [31]. Avani et al. [32] presented a linearizable range search algorithm for one-dimensional datasets using a lock-free k-ary search tree, which can be used to obtain a snapshot by allowing a range to cover the entire dataset. Building on the snapshot object of Jayanti [26], Petrank et al. presented a method to support atomic snapshots in lock-free ordered data structures [33], and illustrated it in linked-list and skip-list. Their approach enables a scan operation to avoid *restart* due to a concurrent modification, as is the case in [32], which would otherwise make it poorly scalable. Essentially, they augment a data structure with a pointer to a special object called *snap-*

* The hierarchical space-partitioning schemes mainly focus on improving on the sizes of the pruned subsets to speed-up the NNS queries. See [1] for various space-partitioning methods, specifically for moderately high-dimensional datasets.

collector that provides a platform for a modify or a CONTAINS operation to *report* a possible modification to a concurrent scan operation. Concurrent scanners use a single snap-collector to return the same snapshot. Nevertheless, directly using a scan of a multidimensional data structure for an NNS will be too naive an idea as it completely discards the advantage of an efficient hierarchical space partitioning structure.

Our work proposes a solution based on augmenting a concurrent multidimensional data structure with a pointer to a special object called *neighbour-collector* that provides a platform for reporting concurrent modifications that can otherwise *invalidate* the output of an NNS if it were to satisfy linearizability. In effect, an operation NNS(a) first searches for an exact match of a in the data structure, and if succeeded returns the same data. However, if an exact match is not found, before starting the iterative scan, NNS(a) *announces* the target point i.e. a and the current best guess for the nearest neighbour using a pointer to a new *active* neighbour-collector. On completing the iterative scan, it *deactivates* the neighbour-collector. A concurrent operation, after completing its own necessary steps, checks for any active neighbour-collector, and if found, reports its output if it was a better guess than the current best guess available at there. Finally, NNS(a) outputs the nearest neighbour as the better guess between the collected and the reported neighbour.

However, unlike the snapshot algorithm in [33], here we can not use a single neighbour-collector for concurrent NNS operations with non-coinciding target points. At the same time, we must allow every NNS operation 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 multidimensional data structure stores a pointer to one of the ends of this list, say the *head*. A new neighbour-collector is allowed to be added only at the other end, say the *tail*. Thus, before announcing a new iterative scan, an NNS operation goes through the list and checks whether there is an active neighbour-collector with same target point. If in the list such an active neighbour-collector is found, it is used for the concurrent *coordinated* iterative scans. A neighbour-collector is removed from the lock-free linked-list as soon as the iterative scan at it gets over. Thus, at any point of time, the length of the list is equal to the number of active NNS operations.

Although it is not necessary for different concurrent NNS operations with coinciding target points to use a single neighbour-collector and perform coordinated iterative scan, but we can observe that this approach helps to speed up the operations in quite an interesting way. Typically, a subset of the dataset is pruned during the iterative scan 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 NNS that starts its iterative scan at a later point, or is slow (or even delayed), will be able to complete much faster.

The design of the LFkD-tree is based on the lock-free BST of Natarajan et al. [9]. To perform an iterative scan, we implement an efficient fully *non-recursive traversal* using *parent* links, which is not available in [9]. 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. Thus, the concurrent modify operations get better progress conditions. Additionally, extra object allocations for synchronization is avoided. The CONTAINS do not perform helping and are wait-free for a finite dataset. The linearizable implementation of NNS 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 [1]. Consequently, we describe the NNS operations independently and comprehensively. Other operations in the LFkD-tree, which are very similar to the same in lock-free BST of [9], are described at a high level in the main paper and their detail discussion is included in Appendix C.

We implemented the LFkD-tree algorithm in Java. The implementation code is available at [34].

In section 2, we present the design of the LockFree-kD-tree. In section 3, we describe the implementation of linearizable NNS. In section 4, we detail the experimental evaluation. Section 5 concludes the paper.

2 LockFree-kD-tree Implementation

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 is given by $x_i{=}c,\ 1{\le}i{\le}d$. 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. This arrangement allows each node, as in an internal BST representation [10], to be used for storing data. However, removing a node from an internal BST is costly, more so in a concurrent setting [8,10]. With this in mind, we opt for an external BST representation [7,9] to design the LFkD-tree. In this design, only *leaf-nodes* contain the data-points and *internal-nodes* route a traversal, see fig. 1 (b). More importantly, it gives us the flexibility to compute c and $i : 1{\le}i{\le}d$ 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 [14,15]. 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 2.2.

Every 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 $\mathtt{N}(k)$ (or $\mathtt{N}(\{k_i\}_{i=1}^{d})$)„ and an internal node associated with a hyperplane $x_i{=}c$, by $\mathtt{N}(i,c)$. Every 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 root. For every internal-node $\mathtt{N}(i,c)$, Υ maintains the following invariants: (i) a node $\mathtt{N}(\{k_i\}_{i=1}^{d})$ belongs to the *left subtree*, if $k_i{<}c$, (ii) a node $\mathtt{N}(\{k_i\}_{i=1}^{d})$ belongs to the *right subtree*, if $k_i{\ge}c$ and (iii) both subtrees are themselves LFkD-tree. (i) and (ii) together are called the *symmetric order* of the LFkD-tree. Figure 1 illustrates the structure of a subtree of a LFkD-tree corresponding to a sample 2-dimensional dataset.

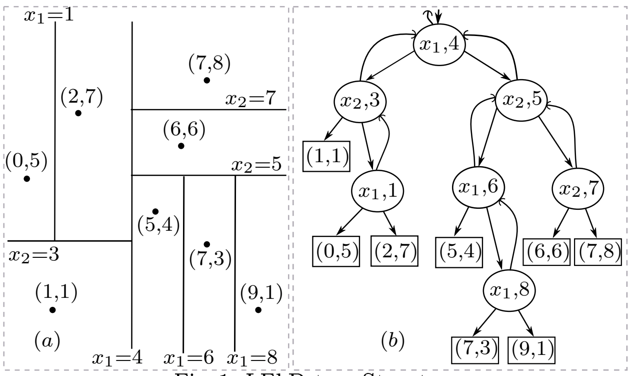

Fig. 1: LFkD-tree Structure

2.2 Sequential Specification of the ADT Operations

LFkD-tree implements an abstract data type kDSet that provides operations ADD, REMOVE, CONTAINS and NNS. 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 and thus arrive at a leaf-node.

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

Local-midpoint rule: *$1{\le}i{\le}d$ is the index of 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 at, has the key a, i.e. $\mathtt{N}(a){\in}\Upsilon$, we modify the link from the *grandparent* of $\mathtt{N}(a)$, denoted by $g(\mathtt{N}(a))$, to its parent, to connect the *sibling* of $\mathtt{N}(a)$, $s(\mathtt{N}(a))$, to $g(\mathtt{N}(a))$; and return true. If $\mathtt{N}(a){\notin}\Upsilon$, REMOVE($a$) returns false. To perform CONTAINS(a), using a similar query we check whether $\mathtt{N}(a){\in}\Upsilon$ and return true or false accordingly.

The operation NNS(a) is non-trivial. On termination of the initial query, if we reach at $\mathtt{N}(b)$ 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 $p(\mathtt{N}(b))$ (the current subtree consists the single node $\mathtt{N}(b)$) stores a *better guess*. Suppose that $p(\mathtt{N}(b)){=}\mathtt{N}(i,c)$. 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 $|a_i{-}c|{>}||a,b||_2$, we must prune the other subtree i.e. one rooted at $s(\mathtt{N}(b))$, 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.

2.3 Lock-free Synchronization

As the basic structure of our LFkD-tree is based on an external BST, for the lock-free synchronization in the LFkD-tree, we build upon the lock-free BST algorithm of [9]. 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 $\mathtt{N}(a)$, as shown in the fig. 2(b), we use a `CAS` to inject operation descriptors at the links $p(\mathtt{N}(a))\rightsquigarrow\mathtt{N}(a)$, $g(\mathtt{N}(a))\rightsquigarrow p(\mathtt{N}(a))$ and $p(\mathtt{N}(a))\rightsquigarrow s(\mathtt{N}(a))$, 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 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 $g(\mathtt{N}(a))\rightsquigarrow p(\mathtt{N}(a))$, whereas, a `tag` indicates that the next step is to inject the descriptor `flag` at the link $p(\mathtt{N}(a))\rightsquigarrow s(\mathtt{N}(a))$. 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 are shown in the fig. 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 CONTAINS or NNS operation does not perform help; thus is wait-free for a finite dataset.

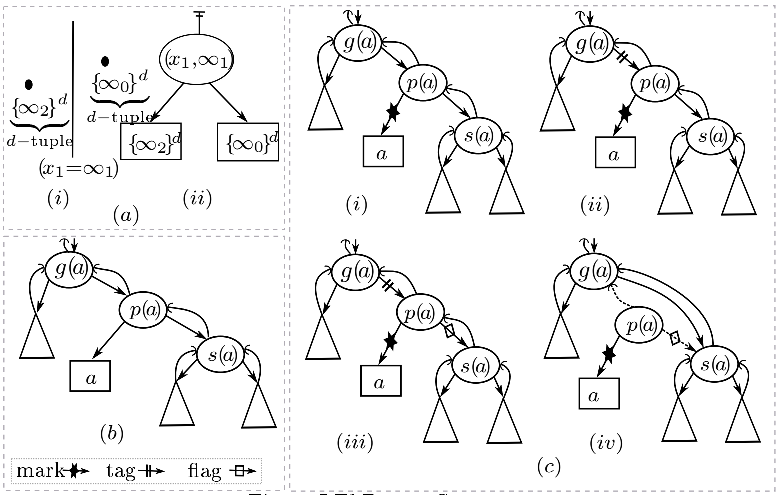

Fig. 2: LFkD-tree Structure

We call the `CAS` step that injects a `mark` at $p(\mathtt{N}(a))\rightsquigarrow\mathtt{N}(a)$ the *logical remove* of a. After this step, a CONTAINS(a) that reads $p(\mathtt{N}(a))\rightsquigarrow\mathtt{N}(a)$ returns false. Accordingly, ADD(a) performs helping to complete the pending REMOVE(a), if it reads $p(\mathtt{N}(a))\rightsquigarrow\mathtt{N}(a)$ 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[**] problem does not occur. Additionally, we assume the availability of a lock-free 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 fig. 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 $\mathtt{N}(1,\infty_1)$ works as the root of the LFkD-tree and the entire dataset is stored in its left subtree.

The focus of the paper is on the linearizable NNS. Therefore, we skip the detail description of the operations ADD, REMOVE and CONTAINS to Appendix C. The full correctness proof is given in Appendix E, but here as a precursor we state the linearization points of the operations ADD, REMOVE and CONTAINS.

2.4 Linearization points of the ADD, REMOVE and CONTAINS operations

For a successful ADD operation, execution of the `CAS`, where a new internal-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 leaf-node with matching key. For a successful REMOVE operation, the `CAS` that replaces a descriptor-free pointer with `marker` i.e. 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 `N` containing the query key in the LFkD-tree at the invocation but was logically or completely removed by a concurrent REMOVE operation before the return of `Search()`, 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 LFkD-tree at the invocation of the REMOVE or CONTAINS, the invocation point itself is taken as the linearization point.

[**] ABA is a shortcut for stating that a value at a shared variable can change from A to B and then back to A, which is a fundamental problem to a `CAS`-based lock-free algorithm, and if not remedied, it can corrupt the semantics of the algorithm.

3 Linearizable Nearest Neighbour Search

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

But, 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 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 $\mathtt{N}(1, \infty_1)$. 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. Note that, **INode**$*$ and **LNode**$*$ inherit **Node**$*$.

```
1 class INode {▷ A subclass of Node.
    long i; double c; Node* lt, rt, pr;
  }▷ Node*: A Node-pointer.
2 class LNode {▷ A subclass of Node.
    double[] k; ▷ k is an array.
  }
3 root := INode*(1, $\infty_1$, null, null, null);
  root·lt := LNode*($\{\infty_2\}^d$);
  root·rt := LNode*($\{\infty_0\}^d$);
```

Algorithm 1. The node structure in the LFkD-tree

Now, before describing the NNS algorithms, we discuss the linearizability of the operations.

3.1 Linearization argument

Consider the concurrent modifications in the LFkD-tree, when an NNS 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 `N`, whose key is returned as the nearest neighbour. Now, if `N` 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*.

```
1 class Nebr {▷Neighbour
    Node* a; double d;
  }▷ Nebr*: Nebr-pointer.
2 class NbrClctr{▷Neighbour-collector
    double[] tgt; bool isAct;
    Nebr* col, rep; NbrClctr* next;
  }▷ NbrClctr*: NbrClctr-pointer.
3 ncp := NbrClctr*(null, false, null, null, null);

NNS(double[] k)
4   pa := root; a := pa·lt; cD := L;
5   hi := $\{\infty_0\}^d$; lo := $\{-\infty_0\}^d$;
6   ⟨pa, a⟩ := Seek(pa, a, k, hi, lo);
7   dst := IsMark(a) ? $\infty$ : $||k, a·k||_2$;
8   if dst $\neq$ 0 then
9       return NNSync(pa, a, dst, k, hi, lo);
10  else {Sync(pa, a); return k;}

Seek(Node* pa, Node* a, double[] k, double[] hi, double[] lo)
11  ···; return ⟨pa, a⟩;▷See appendix D.

Collect(Node* pa, Node* a, double[] k, double[] hi, double[] lo, double dst, NbrClctr* nn)
12  while pa $\neq$ Ptr(root) and dst $\neq$ 0 do
13      ⟨pa, a⟩ := NextGuess(pa, a, dst, k, hi, lo);
14      if ChkValid(pa, a) then
15          dst := AdNebr(a, nn, col);
16  return nn;

NextGuess(Node* pa, Node* a, double dst, double[] k, double[] hi, double[] lo)
17  ···; return ⟨pa, a⟩;▷See appendix D.

NNSync(Node* pa, Node* a, double dst, double[] k, double[] hi, double[] lo)
18  while true do
19      on := ncp;
20      if on·isAct = false then
21          cN := Nebr*(a, dst);
22          nn := NbrClctr*(k, true, cN, cN, null);
23          if CAS(ncp·ref, on, nn) then break;
24      else
25          if ChkValid(pa, a) then
26              dst := AdNebr(a, on, col);
27          nn := on; break;
28  nn := Collect(pa, a, dst, k, hi, lo, nn);
29  Deactivate(nn); return Process(nn);

AdNebr(Node* a, NbrClctr* nn, bool nt)
    ▷nt (Neighbour-type): col or rep.
30  while true do
31      nbr := (nt = col) ? nn·col : nn·rep;
32      if nn·isAct and !IsFinish(nbr) then
33          ⟨dst, nb⟩ := NearNbr(a, nn);
34          if nb = null then return dst;
35          if nt = col then
36              res := CAS(nn·col·ref, nbr, nb);
37          else res := CAS(nn·rep·ref, nbr, nb);
38          if res then return dst;
39      else return 0;

Sync(Node* pa, Node* a)
40  if ncp·isAct then
41      ⟨d, nb⟩ := NearNbr(a, ncp);
42      if nb $\neq$ null and ChkValid(pa, a) then
43          Report(a, ncp);

NearNbr(Node* a, NbrClctr* nn)
44  distTgt := $||a·k, nn·tgt||_2$;
45  col := nn·col; rep := nn·rep;
46  if distTgt<col·d and distTgt<rep·d then
47      return ⟨distTgt, Nebr*(a, distTgt)⟩;
48  else return ⟨distTgt, null ⟩;

Process(NbrClctr* nn)
49  if nn·rep·d < nn·col·d then
50      return nn·rep·a;
51  else return nn·col·a;

Deactivate(NbrClctr* nn)
52  BlockNebr(nn, col);
53  nn·isAct := false;
54  BlockNebr(nn, rep);

BlockNebr(NbrClctr* nn, bool nt)
    ▷nt (Neighbour-type): col or rep.
55  nbr := (nt = col) ? nn·col : nn·rep;
56  while !IsFinish(nbr) do
57      if nt = col then
58          CAS(nn·col·ref, nbr, Finish(nbr));
59      else CAS(nn·rep·ref, nbr, Finish(nbr));
60      nbr := nt = col ? nn·col : nn·rep;

ChkValid(Node* pa, Node* a)
61  k := a·k; ch := Child(pa, Dir(pa, k));
62  while Ptr(ch)·class $\neq$ LNode do
63      ch := Ptr(Child(ch, Dir(ch, k)));
64  if IsMark(ch) then return false;
65  return ch = a ? true: false;

Report(Node* a, NbrClctr* nn)
66  AdNebr(a, nn, rep);
```

Algorithm 2. Linearizable NNS operations with single target point in LFkD-tree

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 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 `N` to the LFkD-tree and could not report it to *op*. If in the meantime a concurrent Contains operation, say op_3, read `N` 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.

Please 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 `N`, if `N` contained the nearest neighbour of the target point. But, if in between the linearization of op_3 and invocation of op, a concurrent REMOVE operation removed `N`, 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`().

3.2 Concurrent NNS with coinciding target point

3.2.1 Overview:

When concurrent NNS 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 [33]. The pseudo-code of the algorithm is given in algorithm 2.

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 NNS. 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 NNS 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`, which is used in algorithm 3. The LFkD-tree is augmented with a pointer `ncp`, line 3, initialized to point to an *inactive* neighbour-collector.

3.2.2 The coordinated iterative scan:

The operation NNS, line 4 to 10, starts with calling the method `Seek`(), line 6, to perform the initial query to arrive at a leaf-node. Now, if the pointer to leaf-node `a` is free of descriptor `mark`, which indicates the 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 NNS returns, line 10. Otherwise, NNS calls the method `NNSync`(), which performs further steps and returns the nearest neighbour, line 9. The arrays `hi` and `lo` are used to support non-recursive traversal, which we describe in the Appendix D. `NNSync`() and the methods called subsequently are described here.

The method `NNSync`(), line 18 to 29, 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 23. In case `ncp` was pointing to an active neighbour-collector, the current best guess of nearest-neighbour, as contained in the leaf-node, is attempted to be added to that. On an active neighbour-collector, the method `Collect`() is called to perform a *coordinated iterative scan*, line 28.

`Collect`(), line 12 to 15, calls the method `NextGuess`(), line 13, to perform next iteration that can better the current best guess of the nearest neighbour. We describe `NextGuess`() in the Appendix D. 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 61 to line 65, performs a query to get 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.

`AdNebr`(), line 30 to 39, is called to add a collected or reported neighbour to an active neighbour-collector. It calls the method `NearNbr`(), shown in line 44 to 47, 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 29. `Deactivate`(), line 52 to 54, 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 55 to line 60, performs a `CAS` to replace a neighbour-pointer with one that has the descriptor `finish` over it, see lines 58 and 59. It ensures that each of the concurrent NNS 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 32.

Finally, the method Process(), line 49 to 51, 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 NNS to output.

Note that, once a neighbour-collector is *deactivated* by an NNS, the method AdNebr() returns 0, line 39. This in turn, immediately terminates the **While** loop in Collect() at the line 12. Thus, as mentioned in section 1.2, we can observe that the *coordination* among the concurrent iterative scans at the same neighbour-collector helps a delayed NNS operation to complete faster.

3.2.3 The reporting methods:

The method Sync(), line 40 to 43, is used by an Add or a Contains operation after their completion, see algorithm 4 at lines 33 and 48. It first checks the active status of the neighbour-collector and then calls the method NearNbr() to create a neighbour. If the point to be reported is not better than the current best guess available, NearNbr() 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 66.

3.3 A general case of Concurrent NNS with multiple target points

3.3.1 Overview:

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

The linearization points remain unchanged as before: the concurrent NNS with coinciding target points share an atomic step during the lifetime of one of them as their linearization point with some order among themselves; other operations linearize as described in the section 2.4.

```
1  tail := NbrClctr∗(null, false, null, null, null);
2  head := NbrClctr∗(null, false, null, null, tail);

   Sync(Node∗ pa, Node∗ a)
3      n := head·nxt;
4      while n ≠ tail do
5          if n·isAct then
6              nb := NearNbr(a, n);
7              if nb ≠ null and ChkValid(pa, a) then
8                  Report(a, n);
9              else break;
10         else n := Ptr(n·nxt);

   Clean(NbrClctr∗ pre, NbrClctr∗ nn)
11     nxt := nn·nxt;
12     while !IsMark(nxt) do
13         CAS(nn·nxt·ref, nxt, Mark(nxt));
14         nxt := nn·nxt;
15     if CAS(pre·nxt·ref, nn, Ptr(nxt)) then
16         return Process(nn);
17     else return null;

   Finalize(Node∗ pa, Node∗ a, double dst, double[] k, double[] hi, double[] lo,
   NbrClctr∗ p, NbrClctr∗ c, enum md)
18     if md = COLLECT then nn := c; pre := p;
19     else if md = INIT then
20         nn := Allocate(a, dst, k, c); pre := c;
21         if nn ≠ null then mode := COLLECT;
22     if md = COLLECT then
23         nn := Collect(pa, a, dst, k, hi, lo, nn);
24         Deactivate(nn); md := CLEAN;
25         if (val := Clean(pre, nn)) ≠ null then
26             return ⟨val, md⟩;

   Allocate(Node∗ a, double dst, double[] k, NbrClctr∗ c)
27     cNb := Nebr∗(a, dst);
28     nn := NbrClctr∗(k, true, cNb, cNb, tail);
29     if CAS(c·ref, on, nn) then return nn;
30     else return null;

   NNSync(Node∗ pa, Node∗ a, double dst, double[] k, double[] hi, double[] lo)
31     nn := null; mode := INIT;
32     retry:
33     while true do
34         p := null; c := head; n := c·nxt;
35         while Ptr(n) ≠ tail do
36             if n = nn and mode = CLEAN then
37                 if (val := Clean(c, nn)) ≠ null then
38                     return val;
39                 else goto retry;
40             else if k = n·tgt and n·isAct then
41                 nn := n; mode := COLLECT; break;
42             else {p := c; c := n; n := n·nxt;}
43         if mode = INIT and IsMark(n) then
44             CAS(p·nxt·ref, c, Ptr(n)); goto retry;
45         if mode ≠ CLEAN then
46             ⟨val, mode⟩ := Finalize(pa, a, dst, k,
               hi, lo, p, c, mode);
47             if val ≠ null then return val;
48         else return Process(nn);
```

Algorithm 3. Linearizable NNS operations with multiple distinct target points in LFkD-tree

3.3.2 Algorithm:

The pseudo-code of the algorithm is given in algorithm 3, in which every method is absolutely same as that in algorithm 2, 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. [35], 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. Adding a neighbour-collector takes a single successful CAS similar to [35].

The method NNSync(), line 31 to 48, as called by NNS after the initial query in algorithm 2, 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 41. 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 43, and if it is, a CAS is attempted to detach it from the list and the traversal is restarted, line 44.

After that, if the mode is INIT or COLLECT, the method Finalize() is called. Finalize(), line 18 to 26, if called in the mode INIT, allocates a new neighbour-collector by calling the method Allocate(), 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 2. 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 11 to 17, performs the two CAS steps to remove the neighbour-collector and calls the method Process(), line 16, 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 38, to redo the remaining steps and return the nearest neighbour. If the traversal terminates in the mode CLEAN, that implies that a concurrent NNS would have detached the deactivated neighbour-collector and therefore Process() is called to finish, line 48.

4 Experimental Evaluation

4.1 Experimental Setup

We implemented the LFkD-tree algorithm in Java using RTTI. We used AtomicReferenceFieldUpdater for 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: A Java implementation of sequential kD-tree of [36] by Levy [37] that supports Remove operation (mostly the available kD-tree implementations do not support Remove).
2. LFKD: Our implementation of the LFkD-tree with NNS.
3. PH-tree: A multi-dimensional storage and indexing data structure by Zäschke *et al.* [24] that supports Remove operations. The implementation is single-threaded.

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 [38] 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. Two synthetic datasets represent more extreme cases. The SKEWED dataset described by Arge *et al.* [39] in which different dimensions have different distributions. The CLUSTER dataset [24] is an extension of a synthetic dataset previously described by Arge *et al.* [39]. It consists of 10000 clusters of points evenly spaced on a horizontal line. A detailed description of the datasets can found in Appendix B.

Workload and Methodology: 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 $\in \big\{\{0\text{-}10^6\},\{0\text{-}10^7\}\big\}$, iii) distribution of (Add-Remove-NNS) $\in \{(05,05,90),(25,25,50)\}$, and iV) number of threads $\in \{1,2,4,8,16,32\}$. We have not included CONTAINS operations in experiment because essentially it would increase the proportion of exact-match NNS. All executions use the same set of randomly generated points for the selected workload characteristics. The graphs present statistical averages of throughput over 6 runs of each experiment.

4.2 Observations and Discussion

For the TIGER/Line dataset, LFKD has significantly higher performance than both the PH-tree and the Levy-Kd for all workload distributions (2.5× for the single thread case), see Figure 4. This performance scales up with increasing thread count (with the LFKD up to 19× in the NNS dominated workload). Additionally, the graphs show that the PH-tree outperforms the Levy-Kd only for workloads that do not involve NNS. Figure 3(a) and 3(b) depict the throughput of different implementations for the SKEWED and CLUSTER datasets respectively. 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). We observe that our LFKD outperforms other designs and scales well up to 32 threads (when CPU saturates with threads) for lower dimensions of the key. As we increase the key dimension, the performance degrades for workloads dominated by the NNS. This degradation with increasing key dimensions is expected in kD-trees due to the *curse of dimensionality* [1]. This performance pattern is identical for different key ranges. However, the LFKD still achieve speedup over the single threaded implementations.

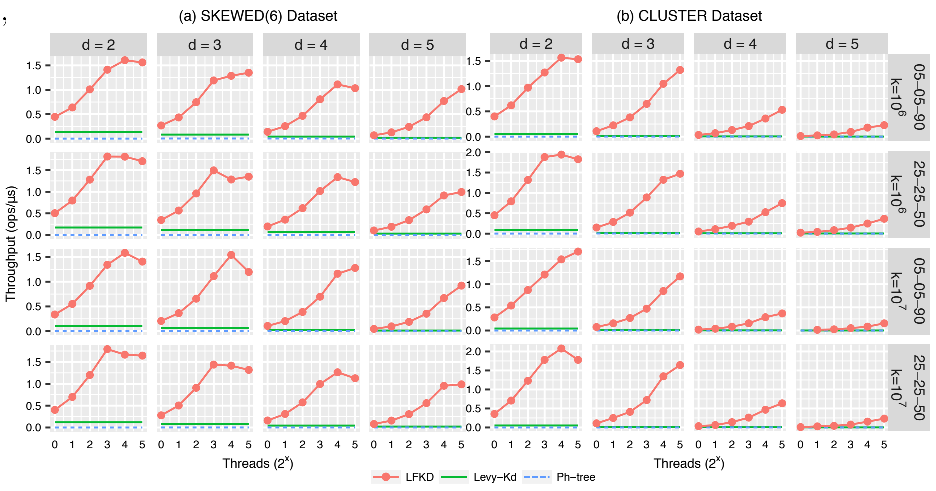

Fig. 3: Performance on the SKEWED(6) and CLUSTER datasets. A column corresponds to a dimension of the data.

We observe that for NNS dominated workload(90% NNS, 5% ADD and 5% REMOVE), the LFKD achieves speedups up to 66× for SKEWED and up to 150× 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 [15] that was designed for the purpose of handling the extreme case such as a CLUSTER distribution, to a concurrent setting.

For a workload with increased modify operations (50% NNS, 25% ADD and 25% REMOVE), the performance of LFkD-tree is degraded by increasing key dimension. The absolute throughput figures are higher for the NNS 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 compared to search operations. However in higher dimensions, the throughput of the NNS is lower as the number of visited nodes increases tremendously with dimension.

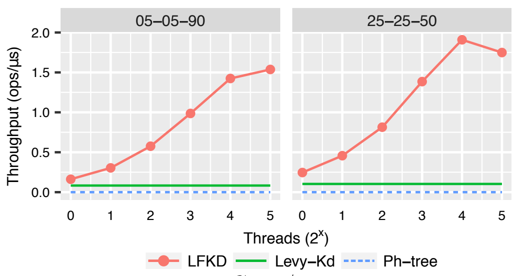

Fig. 4: 2-D TIGER/Line dataset.

5 Conclusion and Future Work

For a large number of applications, which require data-structures supporting dynamic modifications along with nearest neighbour search, research community has largely focused on improving the design of sequential data-structures. The approximate solutions of this problem, which are popular, do not address the issue of providing concurrency support. We introduce LFkD-tree, a lock-free design of kD-tree, which supports linearizable nearest neighbour search operations with concurrent dynamic add and remove of data. We provide 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 multi-dimensional 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. We also plan to extend our work to k-nearest neighbour (kNN) search.

References

1. H. Samet, *Foundations of multidimensional and metric data structures.* Morgan Kaufmann, 2006.
2. S. Berchtold, C. Böhm, B. Braunmüller, D. A. Keim, and H.-P. Kriegel, *Fast parallel similarity search in multimedia databases.* ACM, 1997, vol. 26, no. 2.
3. T. M. Chan, "A dynamic data structure for 3-d convex hulls and 2-d nearest neighbor queries," in *17th SODA*, 2006, pp. 1196–1202.
4. M. P. Herlihy and J. M. Wing, "Linearizability: A correctness condition for concurrent objects," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 12, no. 3, pp. 463–492, 1990.
5. H. Sundell and P. Tsigas, "Fast and lock-free concurrent priority queues for multi-thread systems," *J. Parallel Distrib. Comput.*, vol. 65, no. 5, pp. 609–627, 2005.
6. D. Lea, "`ConcurrentSkipListMap`," in *`java.util.concurrent`*.
7. F. Ellen, P. Fatourou, E. Ruppert, and F. van Breugel, "Non-blocking binary search trees," in *29th PODC*, 2010, pp. 131–140.
8. S. V. Howley and J. Jones, "A non-blocking internal binary search tree," in *24th SPAA*, 2012, pp. 161–171.
9. A. Natarajan and N. Mittal, "Fast concurrent lock-free binary search trees," in *19th PPoPP*, 2014, pp. 317–328.
10. B. Chatterjee, N. Nguyen, and P. Tsigas, "Efficient lock-free binary search trees," in *33rd PODC*, 2014, pp. 322–331.
11. T. Brown and J. Helga, "Non-blocking k-ary search trees," in *15th OPODIS*. Springer, 2011, pp. 207–221.
12. A. Braginsky and E. Petrank, "A lock-free b+tree," in *Proceedings of the 24th SPAA*, 2012, pp. 58–67.
13. J. L. Bentley, "Multidimensional binary search trees used for associative searching," *CACM*, vol. 18, no. 9, pp. 509–517, 1975.
14. 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)*, vol. 3, no. 3, pp. 209–226, 1977.
15. D. M. Mount and S. Arya, "Ann: a library for approximate nearest neighbor searching," *http://www.cs.umd.edu/~mount/ANN/*, 1998.
16. M. Dickerson, C. A. Duncan, and M. T. Goodrich, "Kd trees are better when cut on the longest side," in *Algorithms-ESA 2000.* Springer, 2000, pp. 179–190.
17. S. Arya and H.-Y. A. Fu, "Expected-case complexity of approximate nearest neighbor searching," *SIAM Journal on Computing*, vol. 32, no. 3, pp. 793–815, 2003.
18. R. Panigrahy, "An improved algorithm finding nearest neighbor using kd-trees," in *LATIN 2008*, 2008, pp. 387–398.
19. D. R. Horn, J. Sugerman, M. Houston, and P. Hanrahan, "Interactive kd tree gpu raytracing," in *Proceedings of the 2007 symposium on Interactive 3D graphics and games.* ACM, 2007, pp. 167–174.
20. M. Shevtsov, A. Soupikov, and A. Kapustin, "Highly parallel fast kd-tree construction for interactive ray tracing of dynamic scenes," in *Computer Graphics Forum*, vol. 26, no. 3, 2007, pp. 395–404.
21. K. Zhou, Q. Hou, R. Wang, and B. Guo, "Real-time kd-tree construction on graphics hardware," *ACM Transactions on Graphics (TOG)*, vol. 27, no. 5, p. 126, 2008.
22. B. Choi, R. Komuravelli, V. Lu, H. Sung, R. L. Bocchino, S. V. Adve, and J. C. Hart, "Parallel sah kd tree construction," in *Proceedings of the Conference on High Performance Graphics.* Eurographics Association, 2010, pp. 77–86.
23. J. Ichnowski and R. Alterovitz, "Scalable multicore motion planning using lock-free concurrency," *Robotics, IEEE Transactions on*, vol. 30, no. 5, pp. 1123–1136, 2014.
24. 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.* ACM, 2014, pp. 397–408.
25. Y. Afek, H. Attiya, D. Dolev, E. Gafni, M. Merritt, and N. Shavit, "Atomic snapshots of shared memory," *Journal of the ACM (JACM)*, vol. 40, no. 4, pp. 873–890, 1993.
26. P. Jayanti, "An optimal multi-writer snapshot algorithm," in *37th STOC*, 2005, pp. 723–732.
27. P. Fatourou and N. D. Kallimanis, "Time-optimal, space-efficient single-scanner snapshots & multi-scanner snapshots using cas," in *Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing.* ACM, 2007, pp. 33–42.
28. Y. Afek, N. Shavit, and M. Tzafrir, "Interrupting snapshots and the javatm size method," *Journal of Parallel and Distributed Computing*, vol. 72, no. 7, pp. 880–888, 2012.
29. Y. Nikolakopoulos, A. Gidenstam, M. Papatriantafilou, and P. Tsigas, "A consistency framework for iteration operations in concurrent data structures," in *Parallel and Distributed Processing Symposium (IPDPS), 2015 IEEE International.* IEEE, 2015, pp. 239–248.
30. N. G. Bronson, J. Casper, H. Chafi, and K. Olukotun, "A practical concurrent binary search tree," *SIGPLAN Not.*, vol. 45, no. 5, pp. 257–268, 2010.
31. A. Prokopec, N. G. Bronson, P. Bagwell, and M. Odersky, "Concurrent tries with efficient non-blocking snapshots," in *Acm Sigplan Notices*, vol. 47, no. 8. ACM, 2012, pp. 151–160.
32. H. Avni, N. Shavit, and A. Suissa, "Leaplist: lessons learned in designing tm-supported range queries," in *32nd PODC.* ACM, 2013, pp. 299–308.
33. E. Petrank and S. Timnat, "Lock-free data-structure iterators," in *Distributed Computing.* Springer, 2013, pp. 224–238.
34. B. Chatterjee, "`ConcurrentKDTree`," in *https://github.com/bapi/ConcurrentKDTree.*
35. T. L. Harris, "A pragmatic implementation of non-blocking linked-lists," in *DISC*, vol. 1. Springer, 2001, pp. 300–314.

36. A. W. Moore, "Efficient memory-based learning for robot control," University of Cambridge, Tech. Rep. 209, 1991.
37. S. D. Levy, "`KDTree`," in *edu.wlu.cs.levy.CG.KDTree*.
38. "https://www.census.gov/geo/maps-data/data/tiger.html."
39. 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*, vol. 4, no. 1, pp. 9:1–9:30, 2008.
40. M. Herlihy, "Wait-free synchronization," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 13, no. 1, pp. 124–149, 1991.
41. 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.* ACM, 2002, pp. 73–82.

A 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 following:

(a) Users can join and leave dynamically in real time.
(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 in real time.
(d) The application aims to utilize the multiple cores of a commonly available shared memory machine to get speedup.
(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 crash failure of individual threads.

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 because of the dynamic nature of the application. Furthermore, desiring speedup 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 throughout 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 speedup such an application with a provable progress guarantee.

B Datasets

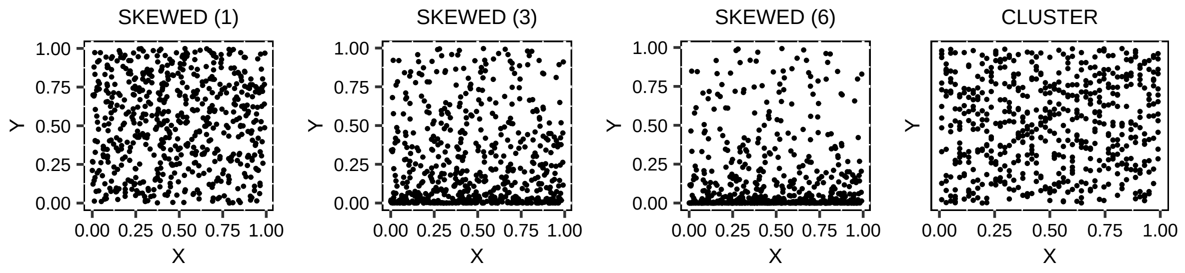

Fig. 5: Synthetic dataset.

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 [38] 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).

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 as depicted in fig. 5. For each point in the dataset, the y value is replaced with the value y^c. In the fig. 5, we show examples for SKEWED(1) which is intuitively uniform distribution in all dimensions, SKEWED (3) and SKEWED (6).

The CLUSTER dataset [24] is an extension of a synthetic dataset previously described by Arge *et al.* [39]. In this evaluation we used clusters of 1000 points evenly spaced on a horizontal line. Each of the clusters is filled with evenly distributed points and stretches 0.00001 in every dimension. Figure 5 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.

During our experimental evaluation, we observed that skewness of the data does not affect the performance of the LFKD fig. 6. On the contrary, as depicted in fig. 7 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.

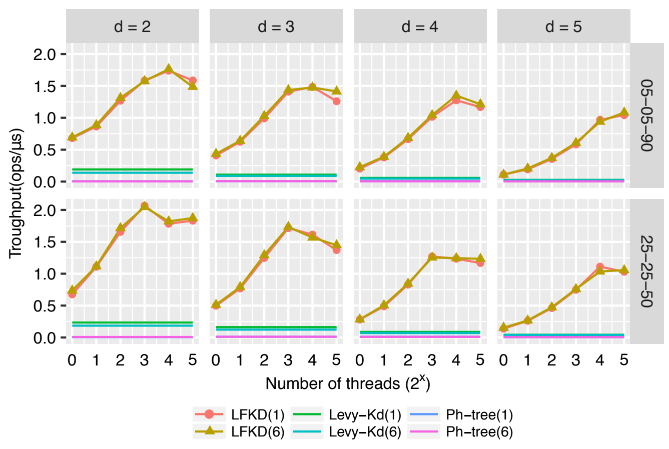

Fig. 6: SKEWED(1) and SKEWED(6) datasets: System throughput

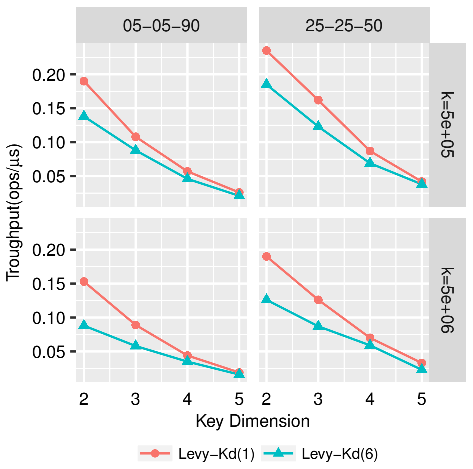

Fig. 7: System throughput for Levy-kd as we vary the dimension for skewness value 1 and 6. Each row representing a key range and each column associated workload distribution.

C Linearizable Add, Remove and Contains operations

C.1 Overview

In a sequential setting, when Remove(a) modifies the link $g(\mathtt{N}(a)) \rightsquigarrow p(\mathtt{N}(a))$, no operation is executed concurrently with a possibility to modify either $p(\mathtt{N}(a)) \rightsquigarrow \mathtt{N}(a)$ or $p(\mathtt{N}(a)) \rightsquigarrow s(\mathtt{N}(a))$. 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 $s(\mathtt{N}(a))$ is an internal-node, a concurrent Remove operation trying to remove a child of $s(\mathtt{N}(a))$ may end up connecting $p(\mathtt{N}(a))$ 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 $p(\mathtt{N}(a)) \rightsquigarrow \mathtt{N}(a)$ and $p(\mathtt{N}(a)) \rightsquigarrow s(\mathtt{N}(a))$ *fixed* when $g(\mathtt{N}(a)) \rightsquigarrow p(\mathtt{N}(a))$ is updated to $g(\mathtt{N}(a)) \rightsquigarrow s(\mathtt{N}(a))$. Additionally, because we maintain parent pointers, we also need to keep the pointer $g(\mathtt{N}(a)) \rightsquigarrow p(\mathtt{N}(a))$ fixed when $s(\mathtt{N}(a)) \rightsquigarrow p(\mathtt{N}(a))$ is updated to $s(\mathtt{N}(a)) \rightsquigarrow g(\mathtt{N}(a))$, in case $s(\mathtt{N}(a))$ 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. [7] suggested to put *operation descriptors*, using `CAS`, at the nodes $g(\mathtt{N}(a))$ and $p(\mathtt{N}(a))$ by a Remove operation and at $p(\mathtt{N}(a))$ 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. [9] suggested that instead of putting the descriptors at the nodes $g(\mathtt{N}(a))$ and $p(\mathtt{N}(a))$, putting them at the links $p(\mathtt{N}(a)) \rightsquigarrow \mathtt{N}(a)$ and $p(\mathtt{N}(a)) \rightsquigarrow s(\mathtt{N}(a))$ improves performance. Both these designs use single-word-sized `CAS` to put descriptors and update the pointers.

Our design is based on [9]. In section 2.3, we described the basics of a lock-free implementation of Add, Remove and Contains operations. Here we describe the algorithm with a pseudo-code.

C.2 Linearizable Add, Remove and Contains operations

We have already described in section 2.3 the operation descriptors and their denotation about the different steps of a Remove operation. In algorithm 4, we use the methods `IsMark`() and `IsFlag`() to check whether a pointer has descriptor `mark` and `flag`, respectively. And, to pack these descriptors, we use the methods `Mark`() and `Flag`(), respectively. To get the value of a pointer free from all descriptors, which gives a node-address,

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

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

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

    Search(Node* pa, Node* a, double[] k)
9       while Ptr(a)·class ≠ LNode do
10          pa := Ptr(a); a := Child(pa, Dir(pa, k));
11      return ⟨pa, a⟩;

    ▷ Crates a new internal-node.
    NewNode(Node* a, Node* b, Node* p)
12      ka := a·k; kb := b·k;
13      i := {i : 1≤i≤d and |ka[i]−kb[i]|≥{|ka[j]−kb[j]|}_{j=1}^{d}};
        ▷ Local-midpoint rule is applied.
14      c := (ka[i]+kb[i])/2;
15      left := (ka[m] < kb[m] ? a : b);
16      right := (ka[m] > kb[m] ? a : b);
17      return INode(m, c, left, right, p);

    AddNode(double[] k)
18      pa := root; a := pa·lt;
19      while true do
20          ⟨pa, a⟩ := Search(pa, a, k);
21          if !IsMark(a) then
22              if k = Ptr(a)·k then
23                  return ⟨Ptr(pa), Ptr(a), false⟩;
24              if IsFlag(a) then pa := Help(pa, a);
25              else
26                  n := LNode(k); cD := Dir(pa, k);
27                  newNd := NewNode(a, n·ref, pa);
28                  if ChCAS(pa, a, newNd·ref, cD) then
29                      return ⟨newNd·ref, n·ref, true⟩;
30          else pa := Help(pa, a);
31          a := Child(pa, Dir(pa, k));

    Add(double[] k)
32      ⟨pa, a, result⟩ := AddNode(k);
33      Sync(pa, a); return result;

    Remove(double[] k)
34      pa := root; a := pa·lt;
35      while true do
36          ⟨pa, a⟩ := Search(pa, a, k);
37          if !IsMark(a) then
38              if k ≠ Ptr(a)·k then return false;
39              if IsFlag(a) then pa := Help(pa, a);
40              marker := Mark(a); cD := Dir(pa, k);
41              else if ChCAS(pa, a, marker, cD) then
42                  Help(pa, a); return true;
43          else return false;
44          a := Child(pa, Dir(pa, k));

    Contains(double[] k)
45      pa := root; a := pa·lt;
46      ⟨pa, a⟩ := Search(pa, a, k);
47      if !IsMark(a) then
48          Sync(Ptr(pa), Ptr(a));
49          return k = Ptr(a)·k ? true : false;
50      else return false;
```

Algorithm 4. The Add, Remove and Contains operations in LFkD-tree

we use the method `Ptr()`. The steps in helping by an operation at a leaf-node-pointer, which has been injected with the descriptor `mark` or `flag`, are performed in the method `Help()`. The steps of the method `Help()` are presented in algorithm 5 in the next section.

The method `Search()`, line 9 to 11, which performs a query, returns the pointers to the leaf-node and its parent, where the query terminates. The method `AddNode()`, line 18 to 31, attempts to add a new node in the LFkD-tree. It starts with calling `Search()`, line 20. 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 30. Otherwise, the node pointed by `a` is checked whether it contains the query key, line 22, and if found, `false` is returned, line 23. `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 24. Only in the case `a` is descriptor-free, the method `NewNode()` is called to allocate a new node, and a `CAS` executed in the method `ChCAS()`, called at line 28, modifies `a` to add the new node. On that, return includes `true`.

The operation Add, line 32 to 33, 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 33, and outputs the result. We describe `Sync()` in the section 3.

The Remove operation, line 34 to 44, performs query in a similar way calling `Search()`, line 36. 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 43. 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 38. 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 41, and if it succeeds, the `Help()` is called to take further steps and `true` is returned, line 42.

A Contains, line 45 to 50, by 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 49; else it returns `false`, line 50. Similar to Add, Contains also calls `Sync()`, which will be explained in the section 3.

C.3 The Helping steps

In algorithm 5, the method `Help()`, line 1 to 6, is called 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 $g(\mathtt{N}(a))$, 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 $p(\mathtt{N}(a))$, we use two types of `tag`: `ltag` and `rtag`. In the method `ApndTag()`, line 13

```
    Help(Node* pa, Node* a)
1   cD := (a·k[pa·i] < pa·c) ? L : R;
2   if IsFlag(a) then
3       ga := pa·pr; 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 := pa·pr; 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 := ga·pr;
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;

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

Algorithm 5. `Help()` Method of Algorithm 4

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 $p(\mathtt{N}(a))$, first that REMOVE is helped and then we reattempt, line 19, otherwise we return ga, line 18. However, if the link $g(\mathtt{N}(a)) \rightsquigarrow p(\mathtt{N}(a))$ 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 $g(\mathtt{N}(a)) \rightsquigarrow p(\mathtt{N}(a))$, we return the pointer ga, line 24. Also, if $g(\mathtt{N}(a))$ is found not connected with $p(\mathtt{N}(a))$, 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 $g(\mathtt{N}(a)) \rightsquigarrow p(\mathtt{N}(a))$ (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 $p(\mathtt{N}(a)) \rightsquigarrow s(\mathtt{N}(a))$ (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 $p(\mathtt{N}(a))$. If $p(\mathtt{N}(a)) \rightsquigarrow s(\mathtt{N}(a))$ is found flagged, we return $s(\mathtt{N}(a))$, 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 $s(\mathtt{N}(a))$ represented by sa, line 33.

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

D The Non-recursive Traversal

```
    Seek(Node* pa, Node* a, double[] k, double[] hi, double[] 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, double[] k,
    double[] hi, double[] 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 {dst := leafdst; break;}
19      else
20          a := pa; pa := pa·pr; cD := Dir(pa, leafKey);
21          if cD = L then
22              if pa·c > hi[pa·i] then hi[pa·i] := pa·c;
23          else
24              if pa·c < lo[pa·i] then lo[pa·i] := pa·c;
25  return ⟨pa, a⟩;
```

Algorithm 6. 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 to begin the query in the operation NNS, the corner points are taken as $\{\infty_0\}^d$ and $\{-\infty_0\}^d$, see line 5 in algorithm 2, which cover the entire dataset.

The method `Seek()`, line 2 to 7, which is called by NNS for the initial query at line 6 in algorithm 2, 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 25, performs an iteration for a better guess of the nearest neighbour given the distance of the current guess from 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. Basically 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 22 and 24.

Thus, the method `Collect()` repeatedly calls `NextGuess()` to perform an iterative scan of the LFkD-tree, see line 13 in algorithm 2.

E Correctness and Lock-freedom

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 processes $\mathcal{P}$ 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 [40]. 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 $p \in \mathcal{P}$.

Abstract Data Type: In this paper, by multidimensional data we mean a point from the real space $\mathbb{R}^d$ and by distance we mean Euclidean distance $||a,b||_2 \; \forall \; a,b \in \mathbb{R}^d$. 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 $\mathcal{B}$:={true, false}. An *abstract data type* (ADT) kDSet is specified as a set of mappings $\mathcal{M} = \{\textsc{Add}, \textsc{Remove}, \textsc{Contains}, \text{NNS}\}$ as defined below:

Definition 1 (***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. $\text{NNS} : \mathbb{R}^d \times F^{\mathbb{R}^d} \mapsto \mathbb{R}^d \times F^{\mathbb{R}^d}$ *s.t.* $\text{NNS}(k, \mathbb{A}) = (a^*, \mathbb{A})$ *where* $a^* \in \mathbb{A} \; \wedge \; ||a^*, k||_2 \leq ||a, k||_2 \forall a \in \mathbb{A}$.

Data Structure: A *LFkD-tree 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 $v \in \mathcal{V}$. In section 2.1, we described the structure of the LFkD-tree in detail. The access of Υ is availed by root - the address of a fixed *sentinel* node. The *left (right) -subtree* of a node N, denoted by N·L (N·R), is the set of nodes comprising of the left (right) -child and all its descendants.

Operation Descriptors: An *operation descriptor* is a boolean variable. A link represented by a pointer, which occupies a single object $v \in \mathcal{V}$, is called to be *injected with a descriptor* des if a *test* for des on the link returns true. A descriptor des can be `mark`, `flag`, `ltag` or `rtag`. We call a link `clean` if it is not injected with any descriptor. 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 can be *reached* following links starting from `root` and the link that connects its parent to itself is not injected with the descriptor `mark`. If Υ_t stores $\mathbb{A}_t$ then $\mathsf{N}{\in}\Upsilon_t \implies \mathsf{N}{\cdot}\mathsf{ref}{\cdot}\mathsf{ky}{\in}\mathbb{A}_t$.

Implementation: An *implementation* $\mathcal{I}_\mathcal{O}$ of kDSet 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 NNS$(k, \mathbb{A})$ is implemented by NNS(k), which outputs a point $a^*{\in}\mathbb{R}^d$ according to the mapping definition.

Operation Steps: A process $p{\in}\mathcal{P}$ 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 process-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 processes $p{\in}\mathcal{P}$, 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 *process 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 process. 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 $op_1\underset{\mathcal{H}}{\rightarrow}op_2$, 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 from $\mathcal{H}$.

Consistent Sequential History: A *sequential specification* of $\mathcal{I}_\mathcal{O}$ is a set of sequential histories. 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 kDSet in $\mathcal{S}$ if the output arguments at the response and $A_{t^r(op)}$ satisfy the corresponding mapping definition of kDSet. The sequential history $\mathcal{S}$ is *consistent* if each operation in it is consistent.

Definition 2 (***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.

In section 2.4, we already mentioned the linearization points of the operations in the implementation $\mathcal{I}_\mathcal{O}$, where $\mathcal{O}{=}\{$Add, Remove, Contains$\}$, of the kDSet. In section 3.1, we discussed the arguments that determine linearization steps of NNS operations when target points are coincident. We also stated in section 3.3 that the linearization point of an NNS operation remains unchanged even if the target points of the concurrent NNS operations do not coincide. Here we list out the linearization points of the operations as the following:

Definition 3 (***Linearization points:***)**.**

1. *For a successful* ADD *operation, it is at line 5 or line 7 in the method* `ChCAS()`*, which is called at line 28 in the method* `AddNode()` *and which in turn was called by* ADD.
2. *For a successful* REMOVE *operation, it is at line 5 or line 7 in the method* `ChCAS()`*, which is called at line 41 in* REMOVE.
3. *For an unsuccessful* ADD *and a successful* CONTAINS *operation it is at line 10 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, as stated in section 2.4.*
5. *For a* NNS *operation, if it returns a data-point which was contained in a collected-neighbour, the linearization point is at line 3 in algorithm 6 in the method* `Seek()` *called from the* NNS.
6. *For a* NNS *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 algorithm 4 that these linearization points are in between $t^i(op)$ and $t^r(op)$ for respective $op{\in}\mathcal{O}{=}\{$ADD, REMOVE, CONTAINS, NNS$\}$.

Now with that, given any concurrent execution history $\mathcal{H}$ of an implementation $\mathcal{I}_{\mathcal{O}}$, where $\mathcal{O}{\subseteq}\{$ADD, REMOVE, CONTAINS, NNS$\}$, 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 section 2.1 are maintained, and the sequential specifications as described in section 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 section 3.3, are maintained by the NNS operations. Therefore, here we state the invariants and present some observations and lemmas which help us to show that the invariants are maintained.

Given a LFkD-tree Υ, for every internal-node $\mathtt{N}(i,c)$ and a leaf node $\mathtt{N}(\{k_i\}_{i=1}^{d})$, Υ maintains the following invariants:

INV 1 *A node* $\mathtt{N}(\{k_i\}_{i=1}^{d})$ *belongs to the left subtree, if* $k_i{<}c$.

INV 2 *A node* $\mathtt{N}(\{k_i\}_{i=1}^{d})$ *belongs to the right subtree, if* $k_i{\geq}c$.

INV 3 *A node* $\mathtt{N}(\{k_i\}_{i=1}^{d})$ *belongs to the right subtree, if* $k_i{\geq}c$.

A LFkD-tree state Υ_t that satisfies the invariants 1 to 3 is called a *valid state*. Now, for the list of the neighbour-collectors, we denote a neighbour-collector by $\mathtt{NC}(\{k_i\}_{i=1}^{d})$ if the target point that it contains is $\{k_i\}_{i=1}^{d}$. A neighbour-collector list maintains following invariant:

INV 4 *In the list there can not be two neighbour-collectors* $\mathtt{NC}(\{k_i\}_{i=1}^{d})$ *and* $\mathtt{NC}(\{j_i\}_{i=1}^{d})$ *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 1 *The fields* k *and* i *are never changed in a* **Node**.

Observation 2 *Any link in a LFkD-tree is updated only using a* `CAS`.

Observation 3 *The sentinel nodes are never removed.*

Observation 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 3, implies the observation 4.

Lemma 1. *In each call of* `Dir()`*, line 2, variable* $\mathtt{N}(i,c){\cdot}$ref *represents a pointer which is* `clean` *and points to an internal-node and thus is not* null.

Lemma 2. *In each call of* `Child()`*, line 3, pa is* `clean` *and points to an internal-node and thus is not* null*.*

Lemma 3. *In each call of* `ChCAS()`*, line 4 to 8, 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 4. *In each call of* `Search()`*, line 9, 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 5. *In each call of* `Search()`*, line 9, pa and a satisfy* $a = pa{\cdot}\mathsf{lt} \mid pa{\cdot}\mathsf{rt}$*.*

Lemma 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 7. *In each call of* `HelpFlg()`*, line 34, ga and pa are* `clean` *and point to two different internal-nodes, whereas sa is eitherpoints to a leaf-node and thus are not* null*.*

Lemma 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 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 10. *In each call of* `ApndFlg()`*, line 27, pa is* `clean` *and points to an internal-node and thus is not* null*.*

Lemma 11. *A pointer once injected with a descriptor* `mark`*,* `flag`*,* `ltag` *or* `rtag` *is not injected with any descriptor ever after.*

The lemma 1 to 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 algorithms 4 and 5. 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 12. *At the termination of* `Search` *at line 11,*

(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 4 and 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 12 (a), (b) and (c).

Now, Following the lemma 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 13, which provides base condition for an induction to prove the theorem 1.

Lemma 13. *Initially, the LFkD-tree consisting of the sentinel nodes satisfies the invariants as stated in section 2.1.*

Now we are prepared to prove theorem 1. We use induction to prove it. Using lemma 13, when no update has happened, the nodes in the LFkD-tree satisfy the invariants. It is straightforward to observe that no CONTAINS or NNS operation involves a write (`CAS`) step and therefore they do not change the state of the LFkD-tree. From lemma 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 1 to 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 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 1 follows.

Theorem 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 [35] and which was further improved my Micheal [41]. A very sophisticated proof of the state change and thus validity of the list algorithm was provided by Micheal [41]. The invariant maintained our list, invariant 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 2. *(Correctness) The operations* ADD, REMOVE, CONTAINS *and* NNS *are linearizable with respect to the abstract data type kDSet.*

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 3 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 12, at the termination of `Search`(), line 10 in algorithm 4, $pa \rightsquigarrow a$ is a leaf-node pointer. Now using the construction of $\mathcal{S}_i$ and definition 3-(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 10 in algorithm 4, by definition 3-(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, definition 1-(1), 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 3-(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- definition 1-(2). 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 NNS 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 NNS 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, NNS 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 3. *(Lock-freedom) The LFkD-tree operations* ADD*,* REMOVE*,* CONTAINS *and* NNS *are lock-free and thus the presented algorithm implements a lock-free LFkD-tree.*

Proof. We take the NNS 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 NNS with concurrent ADD, REMOVE or CONTAINS operations. We consider the case where concurrent NNS 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 NNS 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 NNS 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 NNS operation fails, it indicates that a `CAS` by a concurrent NNS operation succeeded. Finally, a `CAS` to add a new neighbour-collector only indicates that either a new neighbour-collector by a concurrent NNS has been successfully added or a NNS operation has terminated. In case of a `CAS` failure to add a new neighbour-collector, a NNS operation always helps a concurrent pending NNS 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 NNS operation concurrent to any other LFkD-tree operation. Thus we arrive at the theorem 3.

This concludes the proof of the presented algorithm.