Ministry of Science and Higher Education of the Russian Federation

ITMO UNIVERSITY

GRADUATION THESIS

SELF-ADJUSTING CONCURRENT DATA STRUCTURES

Author: Drozdova Aleksandra Alekseevna_______________

Subject area: 01.03.02 Applied mathematics and informatics

Degree level: Bachelor

Thesis supervisor: Aksenov V.E., PhD _______________

Saint Petersburg, 2021

Student Drozdova Aleksandra Alekseevna
Group M3439 Faculty of IT&P

Subject area, program/major
Mathematical models and algorithms in software engineering

Consultant(s):

a) Alistarh D., PhD, assistant professor, IST Austria ______________

Thesis received "____" ___________ 20__

Originality of thesis ____%

Thesis completed with grade ____

Date of defense "____" ___________ 20__

Secretary of State Exam Commission Pavlova O.N. ______________

Number of pages ____

Number of supplementary materials/Blueprints ___________________________

CONTENTS

INTRODUCTION . . . 6
1. The introduction into the Self-Adjusting Ordered Sets field . . . 8
1.1. The Ordered Set . . . 8
1.2. The Static Optimality . . . 8
1.3. The CBTree . . . 9
1.4. The Skip-List . . . 10
1.5. The Interpolation Search Tree . . . 11
Conclusions on Chapter 1 . . . 11
2. The Sequential Distribution-Adaptive Data Structures . . . 12
2.1. The Sequential Splay-List Design . . . 12
2.1.1. The Contains Operation . . . 13
2.1.2. Insert and Delete Operations . . . 18
2.1.3. The Sequential Splay-List Analysis . . . 19
2.2. The Sequential Distribution-Adaptive Interpolation Search Tree . . . 24
2.2.1. Ideal IST and its properties . . . 24
2.2.2. Insert, Delete and Rebuild operations . . . 27
2.2.3. Searching in the distribution-adaptive IST . . . 28
Conclusions on Chapter 2 . . . 32
3. The Concurrent Distribution-Adaptive Data Structures . . . 33
3.1. The Concurrent Splay-List . . . 33
3.1.1. Overview . . . 33
3.1.2. The Relaxed Rebalancing Analysis . . . 33
3.1.3. Relaxed and Forward Rebalancing . . . 36
3.1.4. Lazy Expansion . . . 37
3.2. The Concurrent Distribution-Adaptive Interpolation Search Tree . . . 38
Conclusions on Chapter 3 . . . 38
4. Experiments and results . . . 39
4.1. The Splay-List Implementation . . . 39
4.2. The Splay-List Experiments . . . 43
4.2.1. Read-Only Workloads . . . 43
4.2.2. General workloads . . . 50
4.2.3. The Correlation between Key Popularity and Height . . . 52
Conclusions on Chapter 4 . . . 53

CONCLUSION 54
REFERENCES 55
APPENDIX A. Pseudo code of Rebalancing operation 57

INTRODUCTION

The past decades have seen significant effort on designing efficient concurrent data structures, leading to fast variants being known for many classic data structures, such as hash tables [5, 9, 16], skip lists [1, 7], or search trees [10, 11]. Most of this work has focused on efficient concurrent variants of data structures with optimal worst-case guarantees. However, in many real workloads, the access rates for individual objects are not uniform. This fact is well-known, and is modelled in several industrial benchmarks, such as YCSB [2], or TPC-C [12], where the generated access distributions are heavy-tailed, e.g., following a Zipf distribution [2]. While in the sequential case the question of designing data structures which adapt to the access distribution is well-studied, see [6] and references therein, in the concurrent case significantly less is known. The intuitive reason for this difficulty is that self-adjusting data structures require non-trivial and frequent pointer manipulations, such as node rotations in a balanced search tree, which can be complex to implement concurrently.

Our goal is to create distribution-adaptive concurrent data structures, which would provide significant performance benefits over a classic non-adaptive concurrent designs and existing concurrent self-adjusting designs. In this work, we propose distribution-adaptive versions of the well-known data structures: skip-list and interpolation search tree (later referred to as IST) [8].

In order to achieve our goal we defined several tasks. The tasks are:

— To create self-adjusting modifications of sequential skip-list and IST. This task includes theoretical analysis of memory and time complexities.
— To create the concurrent versions of the designed self-adjusting data structures. This task includes an theoretical and experimental analysis.
— To compare the existing solutions and non-adaptive solutions with the created ones. This task includes implementation of tool, which would execute the experiments, visualize the results. Also, this task includes analysis of the results.

To date, the CBTree [3] is the only concurrent data structure which leverages the skew in the access distribution for faster access. We propose modifications of the existing data structures, which shows better scalability and which are easier to implement.

The thesis is structured as follows:

- In Chapter 1, we give an introduction to the field of self-adjusting data structures. We define an ordered set, briefly describe data structures on top of which, we build our solutions and describe static optimality theorem, which we later proof for the splay-list. We finish the chapter with the description of related work and analysis of existing solutions.
- In Chapter 2, we develop the sequential self-adjusting modifications of skip-list and IST. Then, we analyse time and memory complexities. We finish the second chapter with the comparison of theoretical results with the existing solutions.
- In Chapter 3, we present concurrent modifications of the algorithms in Chapter 2.
- In the last Chapter, we present the results of experiments. In these experiments we compare the existing solution with our for different concurrent settings on different workloads and briefly describe them.

CHAPTER 1. THE INTRODUCTION INTO THE SELF-ADJUSTING ORDERED SETS FIELD

In this chapter, we provide the generalized information about the original versions of data structures, their common interfaces and properties that we desire to obtain.

1.1. The Ordered Set

The ordered set is a data structure that provides the following operations:

a) `contains` for the key, returns information if the key exists in the data structure or not.
b) `find` for the key, returns the value of an element with such a key, if it exists in the data structure, `null` – otherwise.
c) `insert` for the pair of a key and a value, adds pair to the data structure if the key is not presented.
d) `delete` for the key, removes a pair with that key from the data structure if it exists.
e) `nextKey` for the key, returns a minimal key in the data structure, which is strictly greater than the given one.

For all data structures presented in this work, we only describe how to do `insert`, `delete` and `contains` operations, because for them `nextKey` and `find` operations are obvious modifications of `contains` operation.

We say that a `contains` operation is *successful* (returns *true*) if the requested key is found in the data structure and was not marked as deleted; otherwise, the operation is *unsuccessful*. An `insert` operation is *successful* (returns *true*) if the requested key was not present upon insertion; otherwise, it is *unsuccessful*. A `delete` operation is *successful* (returns *true*) if the requested key is found and was not marked as deleted, otherwise, the operation is *unsuccessful*. As suggested, in our implementations the `delete` implementation does not always physically delete the object from the lists–instead, it may just mark it as deleted.

1.2. The Static Optimality

One of the important properties of self-adjusting data structures is static optimality. The splay-tree [17] is one of the first data structures that satisfies this property.

For the splay-tree the following theorem was proven in [17]:

Theorem 1. If every element is accessed at least once, than the total access time is $O\left(m + \sum\limits_{i=1}^{n} q(i) \times \log\left(\frac{m}{q(i)}\right)\right)$, where n is the number of elements in the data structure, $q(i)$ is the number of accesses made to the i-th element and m is $\sum\limits_{m=1}^{n} q(i)$.

Note, that if we are given the requests in advance, the best static data structure has exactly this complexity due to the information theory results. That is why this property is named static optimality.

We use this definition of static optimality for the created data structures. Formally, our task is to create data structures that:

a) can execute ordered set operations;
b) hold static optimality property for a wide class of access distributions;
c) show good worst-case time and memory performance.

1.3. The CBTree

As mentioned above, the CBTree [3] is the only existing concurrent self-adjusting ordered set. It uses technique of semi-splaying to balance itself. Unlike the splay-tree it performs rotations during lookup, on the path from top to bottom, and only if the special function, i.e., a potential, on the tree is decreased by at least some preselected constant. There are two types of rotations, that can be performed depending on the situation (see Figure 1).

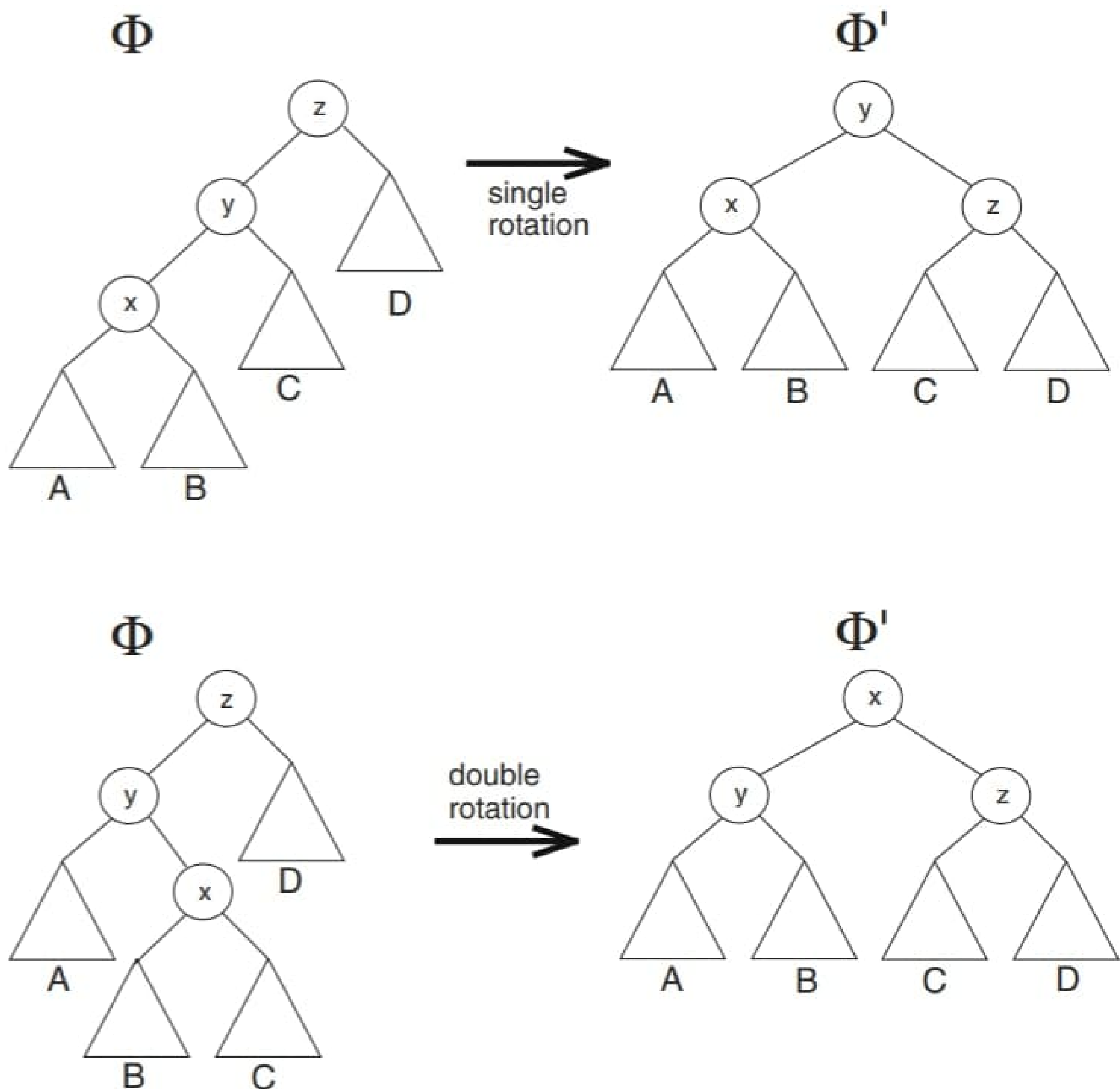


Figure 1 – Rotations of CBTree from [3]

Now let's discuss the main issue of the CBTree. Every contains operation in CBTree makes the second update traversal. During this traversal, threads are conflicting on top of it, which makes the data structure less scalable. So the authors propose a lazy variant in which update traversals are made only by one thread. Though, the authors do not provide the theoretical analysis of the modified version. In this work, we also propose the relaxed version of our data structure, but unlike CBTree authors we prove its theoretical bounds (See Theorem 19).

1.4. The Skip-List

The skip-list was proposed by Pugh in [15]. Generally, the skip-list is linked-list like structure, which allows fast search. It is useful to view these lists as stacked

on top of each other; a list's index (starting from the bottom one, indexed at 0) is also called its *height*. The lists are also ordered by containment, as a higher-index list contains a subset of the objects present in a lower-index list. The higher-index lists are also called *sub-lists*. The bottom list, indexed at 0, contains all the objects present in the data structure at a given point in time. The `contains` operation searches for the key by walking from left to right, from the topmost list to the lowest, skipping elements in a list, which are lower or equal to the key. The `insert` operation first searches for the place to add new element. Then, it adds it to the base list. With probability $\frac{1}{2}$ the node is added to the second list, with probability $\frac{1}{4}$ to the third list, and so on. The `delete` operation finds a node with the provided key and unlinks it from all the levels by changing the links from predecessors to the successors.

The described data structure has lock-free version and is easy to made concurrent in comparison with the binary search trees, for example. That is the reason why we chose this data structure to make it distribution-adaptive (see Section 2.1).

1.5. The Interpolation Search Tree

The interpolation search tree was proposed in [8]. Its worst-case amortized bounds for all operations are $O(\log^2 n)$, but for a wide class of distributions called `smooth`, its expected time of all operations is $O(\log\log n)$. The definition of smoothness is given in [8]. Here, in our work, we slightly change the definition of smoothness. The newly presented class still remains wide that we prove in Lemma 15. The non-blocking algorithm for the IST was described in [14]. That algorithm shows great performance. So, we suggest the self-adjusting modification of it in Section 2.2, and slightly modify collaborative algorithm proposed in [14] to fit our data structure in Section 3.2.

Conclusions on Chapter 1

There are lots of data structures, which can be used as an ordered set. But all of them have some issues. Some are not static optimal, others do not scale well.

In this Chapter, we gave an overview of several data structures that we are going to modify to make them self-adjusting. Moreover, we briefly described the only existing solution concurrent self-adjusting data structure and its flaw.

CHAPTER 2. THE SEQUENTIAL DISTRIBUTION-ADAPTIVE DATA STRUCTURES

In this chapter we introduce two distribution-adaptive data structures: the splay-list (the self-adjusting skip-list) and the self-adjusting interpolation search tree, that build on the well-known designs [8, 16]. Also, we prove their theoretical bounds.

2.1. The Sequential Splay-List Design

The splay-list design builds on the classic skip-list by Pugh [16]. In the following, we will only briefly overview the skip-list structure, and focus on the main technical differences. We refer the reader to [5] for a more in-depth treatment of concurrent skip-lists.

Similar to skip-lists, the splay-list maintains a set of sorted lists, starting from the bottom list, which contains all the objects present in the data structure. Without loss of generality, we assume that each object consists of a key-value pair. We thus use the terms *object* and *key* interchangeably.

Unlike skip-lists, where the choice of which objects should be present in each sub-list is random, a splay-list's structure is adjusted according to the access distribution across keys/objects.

The following definitions make it easier to understand how the operations are handled in splay-lists. The *height of the splay-list* is the number of its sub-lists. The *height of an object* is the height of the highest sub-list containing it. Typically, we do not distinguish between the object and its key.

The height of a key u is the height of a corresponding object h_u. Key u is the *parent of key v at height h* if u is the largest key whose value is smaller than or equal to v, and whose height is at least h. That is, u is the last key at height h in the traversal path to reach v. Critically, note that, if the height of a key v is at least h, then v is its own parent at height h; otherwise, its parent is some node $v \neq u$. In addition, we call the set of objects for which u is the parent at height h, its *h-children* or the *subtree of u at height h, denoted by C_u^h*.

For every key u, we maintain a counter $hits_u$, which counts the number of `contains(`u`)`, `insert(`u`)`, and `delete(`u`)` operations which *visit the object*. In particular, *successful* `contains(`u`)`, `insert(`u`)`, and `delete(`u`)` operations increment $hits_u$ Moreover, unsuccessful operations can also increment $hits_u$ if the element is physically present in the data structure, even though logically deleted,

upon the operation. In this case, the marked element is still visited by the corresponding operation. (We will re-discuss this notion in the later sections, but the simple intuition here is that we cannot store access counts for elements which are not physically present in the data structure, and therefore ignore their access counts.) We will refer to operations that visits an object with the corresponding key simply as *hit-operations*.

For any set of keys S, we define a function $hits(S)$ to be the sum of the number of hits-operations performed to the keys in S. As usual, sentinel *head* and *tail* nodes are added to all sub-lists. The height of a sentinel node height is equal to the height of the splay-list itself, and exceeds the height of all other nodes by at least 1. By convention, $hits_{head} = hits_{tail} = 1$.

2.1.1. The Contains Operation

The contains operation consists of two phases: the search phase and the balancing phase. The search phase is exactly as in skip-list: starting from the head of the top-most list, we traverse the current list until we find the last object with key lower than or equal to the search key. If this object's key is not equal to the search key, the search continues from the same object in the lower list. Otherwise, the search operation completes. The process is repeated until either the key is found or the algorithm attempts to descend from the bottom list, in which case the key is not present.

If the operation finds its target object, its *hits* counter is incremented and the balancing phase starts: its goal is to update the splay-list's structure to better fit the access distribution, by traversing the search path backwards and checking two conditions, which we call the *ascent* and *descent* conditions.

We now overview these conditions. For the descent condition, consider two neighbouring nodes at height h, corresponding to two keys $v < u$. Assume that both v and u are on level h, and consider their respective subtrees C_v^h and C_u^h. Assume further that the number of hits to objects in their subtrees ($hits(C_v^h \cup C_u^h)$) became smaller than a given threshold, which we deem appropriate for the nodes to be at height h. (This threshold is updated as more and more operations are performed.) To fix this imbalance, we can "merge" these two subtrees, by descending the right neighbour, u, below v, thus creating a new subtree of higher overall hit count. Similarly, for the ascent condition, we check whether an object's subtree has *higher* hit count than a threshold, in which case we increase its height by one.

Now, we describe the conditions more formally. Assume that the total number of hit-operations to all objects, including those marked for deletion, appearing in splay-list is m, and that the current height of the splay-list is equal to $k + 1$. Thus, there are k sub-lists, and the sentinel sub-list containing exclusively $head$ and $tail$. Excluding the head, for each object u on a backward path, the following conditions are checked in order.

More formally, the contains operation consists of two phases: the search phase and the balancing phase. The search phase is exactly as in skip-list: starting from the head of the top-most list, we traverse the current list until we find the last object with key lower than or equal to the search key. If this object's key is not equal to the search key, the search continues from the same object in the lower list. Otherwise, the search operation completes. The process is repeated until either the key is found or the algorithm attempts to descend from the bottom list, in which case the key is not present.

Now we describe **the descent condition** formally. Since u is not the head, there must exist an object v which precedes it in the forward traversal order, such that v has height $\geqslant h_u$. If

$$hits(C_u^{h_u}) + hits(C_v^{h_u}) \leqslant \frac{m}{2^{k-h_u}},$$

then the object u is demoted from height h_u, by simply being removed from the sub-list at height h_u. The object stays a member of the sub-list at height $h_u - 1$ and h_u is decremented. The backward traversal is then continued at v.

Let's describe **the ascent condition** formally. Let w be the first successor of u in the list at height h_u, such that w has height *strictly greater than* h_u. Denote the set of objects with keys in the interval $[u, w)$ with height equal to h_u by S_u. If the number of hits m is greater than zero and the following inequality holds:

$$\sum_{x \in S_u} hits(C_x^{h_u}) > \frac{m}{2^{k-h_u-1}},$$

then u is promoted and inserted into the sub-list at height $h_u + 1$. The backward traversal is then continued from u, which is now in the higher-index sub-list. The rest of the path at height h_u is skipped. Note that the object u is again checked against the ascent condition at height $h_u + 1$, so it may be promoted again. Also note that

the calculated sum is just an interval sum, which can be maintained efficiently, as we show later.

For the better understanding of these conditions it is important to talk about **splay-list initialization and expansion**. Initially, the splay-list is empty and has only one level with two nodes, head and tail. Suppose that the total number of hits to objects in splay-list is m. The lowest level on which the object can be depends on how low the element can be demoted. Suppose that the current height of the list is $k+1$. Consider any object at the lowest level 0: in the descent condition we compare $hits(C_u^0) + hits(C_v^0)$ against $\frac{m}{2^k}$. While m is less than 2^{k+1}, the object cannot satisfy this condition since $C_v^{h_u} \geqslant hits_v \geqslant 1$, but when m becomes larger than this threshold, it could. Thus, we have to increase the height of splay-list and add a new list to allow such an object to be demoted. By that, the height of the splay-list is always $\log m$. This process is referred to as *splay-list expansion*. Notice that this procedure could eventually lead to a skip-list of unbounded height. However, this height does not exceed 64, since this would mean that we performed at least 2^{64} successful operations which is unrealistic. We discuss ways to make this procedure more practical, i.e., lazily increase the height of an object only on its traversal, in Section 3.1.1.

Now, we return to the description of the `contains` function. The first phase is the forward pass, which is simply the standard `contains` algorithm which stores the traversal path. If the key is not found, then we stop. Otherwise, suppose that we found an object t. We have to restructure the splay-list by applying ascent and descent conditions. Note, that the only objects that are affected and can change their height lie on the stored path. For that, in each object u we store the total hits to the object itself, $hits_u$, as well as the total number of hits into the "subtree" of each height excluding u, i.e., for all h we maintain $hits_u^h = hits(C_u^h \setminus \{u\})$. We denote the hits to the object u as sh_u.

Thus, when **traversing the path backwards** we check the following:

a) If the object $u \neq t$ is a parent of t on some level h, we then increase its $hits_u^h$ counter. Note that $h \leqslant h_u$.

b) Check the descent condition for v and u as

$$sh_v + hits_v^{h_u} + sh_u + hits_u^{h_u} \leqslant \frac{m}{2^{k-h_u}}.$$

If this is satisfied, demote u and increment $hits_v^{h_u}$ by $sh_u + hits_u^{h_u}$. Continue on the path.

c) Check the ascent condition for u by comparing $\sum_{w \in S_u} sh_w + hits_w^{h_u}$ with $\frac{m}{2^{k-h_u-1}}$. If this is satisfied, add u to the sub-list $h_u + 1$, set $hits_u^{h_u+1}$ to the calculated sum minus sh_u and decrease $hits_v^{h_u+1}$ by the calculated sum, where h is a parent of u at height $h_u + 1$. We then continue with the sub-list on level $h_u + 1$. Below, we describe how to maintain this sum in constant time.

For this we will use **the partial sums trick** [13]. Suppose that $p(u)$ is the parent of u on level $h_u + 1$. During the forward pass, we compute the sum of

$$hits(C_x^{h_u}) = sh_x + hits_x^{h_u}$$

over all objects x which lie on the traversal path between $p(u)$ (including it) and u (not including it). Denote this sum by P_u. Thus, to check the ascent condition on the backward pass, we simply have to compare

$$\sum_{x \in S_u} sh_u + hits(C_x^{h_u}) = sh_{p(u)} + hits_{p(u)}^{h_u+1} - P_u$$

against $\frac{m}{2^{k-h_u-1}}$. Observe that the partial sums $hits(S_u)$ can be increased only by one after each operation. Thus, the only object on level h that can be promoted is the leftmost object on this level. For the first object u, S_u can be calculated as $hits_{p(u)}^{h_u+1} - hits_{p(u)}^{h_u}$. In addition, after the promotion of u, only u and $p(u)$ have their $hits^{h_u+1}$ counters changed. Moreover, there is no need to skip the objects to the left of the promoted object, as suggested by the ascent condition, since there cannot be any such objects.

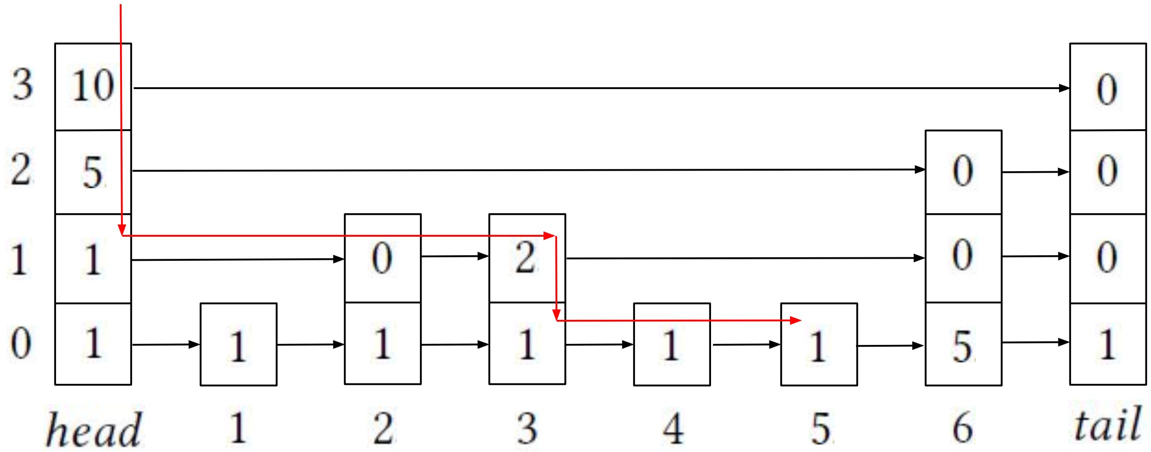


Figure 2 – The state of the splay-list before **contains(5)**

Let's look at an **example**. To illustrate, consider the splay-list provided on Figure 2. It contains keys $1, \dots, 6$ with values $m = 10$ and $k = \lfloor \log m \rfloor = 3$. We can instantiate the sets described above as follows: $C_3^1 = \{3, 4, 5\}$, $C_2^1 = \{2\}$, $C_{head}^1 = \{head, 1\}$ and $C_{head}^2 = \{head, 1, 2, \dots, 5\}$. At the same time, $S_4 = \{4, 5\}$, $S_3 = \{3\}$ and $S_2 = \{2, 3\}$. In the Figure 2, the cell of u at height $h > 0$ contains $hits_u^h$, while the cell at height 0 contains sh_u. For example, $sh_3 = 1$ and $hits_3^1 = sh_4 + sh_5 = 2$, $sh_2 = 1$ and $hits_2^1 = 0$, $sh_1 = 1$ and $hits_{head}^2 = 5$.

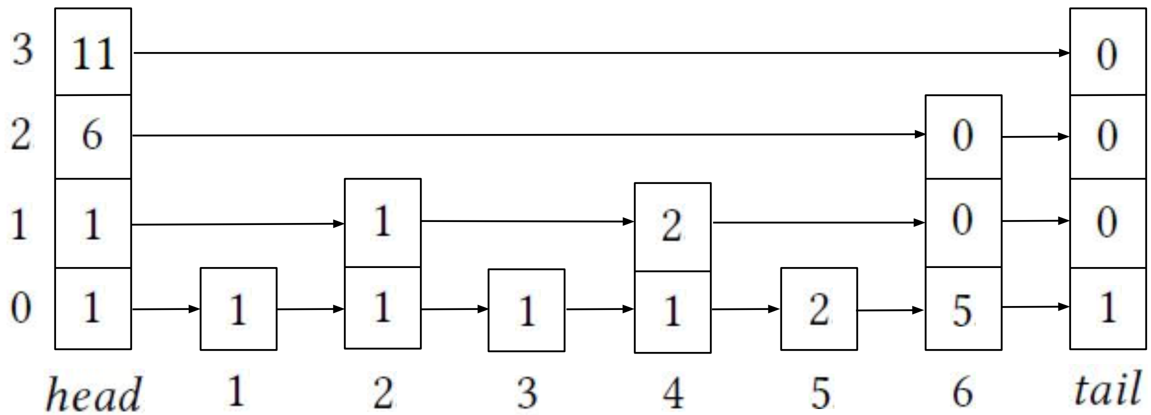


Figure 3 – The state of the splay-list after **contains(5)**

Assume we execute `contains(5)`. On the forward path, we find 5 and the path to it is $2 \to 3 \to 4 \to 5$. We increment m, sh_5, $hits_3^1$ and $hits_{head}^2$ by one. Now, we have to adjust our splay-list on the backward path. We start with 5: we check the descent condition by comparing $hits(C_4^0) + hits(C_5^0) = 3$ with

$\frac{m}{2^{k-0}} = \frac{11}{8}$ and the ascent condition by comparing $hits(S_5) = 2$ with $\frac{m}{2^{k-0-1}} = \frac{11}{4}$. Obviously, neither condition is satisfied. We continue with 4: the descent condition by comparing $hits(C_3^0) + hits(C_4^0) = 2$ with $\frac{11}{8}$ and the ascent condition by comparing $hits(S_4) = 3$ with $\frac{11}{4}$ — the ascent condition is satisfied and we promote object 4 to height 1 and change the counter $hits_3^1$ to 2. For 3, we compared $hits(C_2^1) + hits(C_3^1) = 2$ with $\frac{11}{4}$ and $hits(S_3) = 4$ with $\frac{11}{2}$ — the descent condition is satisfied and we demote object 3 to height 0 and change the counter $hits_2^1$ to 1. Finally, for 2 we compared $hits(C_1^1) + hits(C_2^1) = 4$ with $\frac{11}{4}$ and $hits(S_2) = 5$ with $\frac{11}{2}$ — none of the conditions are satisfied. As a result we get the splay-list shown on Figure 3.

2.1.2. Insert and Delete Operations

Inserting a key u is done by first finding the object with the largest key lower than or equal to u. In case an object with the key is found, but is marked as logically deleted, the insertion unmarks the object, increases its hits counter and completes successfully. Otherwise, u is inserted on the lowest level after the found object. This item has hits count set to 1. In both cases, the structure has to be re-balanced on the backward pass as in `contains` operation. Unlike the skip-list, splay-lists always physically insert into the lowest-level list.

The **delete** operation needs additional care. The operation first searches for an object with the specified key. If the object is found, then the operation logically deletes it by marking it as `deleted`, increases the hits counter and performs the backward pass. Otherwise, the operation completes.

Notice that we maintain the total number of hits on currently logically deleted objects. When it becomes at least half of m, the total number of hits to all objects, we initialize a new structure, and move all non-deleted objects with corresponding hits to it.

The only question left is how to **build a new structure efficiently** enough to amortize the performed delete operations. Suppose that we are given a sorted list of n keys $k_1, \ldots, k_n$ with the number of hit-operations on them $h_1, \ldots, h_n$, where their sum is equal to M. We propose an algorithm that builds a splay-list such that no node satisfies the ascent and descent conditions, using $O(M)$ time and $O(n \log M)$ memory.

The idea behind the algorithm is the following. We provide a recursive procedure that takes the contiguous segment of keys $k_l, \ldots, k_r$ with the total number of

accesses $H = h_l + \ldots + h_r$. The procedure finds p such that $2^{p-1} \leqslant H < 2^p$. Then, it finds a key k_s such that $h_l + \ldots + h_{s-1}$ is less than or equal to $\frac{H}{2}$ and $h_{s+1} + \ldots + h_r$ is less than $\frac{H}{2}$. We create a node for the key k_s with the height p, and recursively call the procedure on segments $k_l, \ldots, k_{s-1}$ and $k_{s+1}, \ldots, k_r$. There exists a straightforward implementation which finds the split point s in $O(r - l)$, i.e., linear time. The resulting algorithm works in $O(n \log M)$ time and takes $O(n \log M)$ memory: the depth of the recursion is $\log M$ and on each level we spend $O(n)$ steps.

However, the described algorithm is not efficient if M is less than $n \log M$. To achieve $O(M)$ complexity, we would like to answer the query to find the split point s in $O(1)$ time. For that, we prepare a special array T which contains in sorted order h_1 times key k_1, h_2 times key k_2, …, h_n times key k_n. To get the required s, at first, we take a subarray of T that corresponds to the segment $[l, r]$ under the process, i.e., h_l times key k_l, …, h_r times key k_r. Then, we take the key k_i that is located in the middle cell $\lceil \frac{h_l + \ldots + h_r}{2} \rceil$ of the chosen subarray. This i is our required s. Let us calculate the total time spent: the depth of the recursion is $\log M$; there is one element on the topmost level which we insert in $\log M$ lists, there are at most two elements on the next to topmost level which we insert in $\log M - 1$ lists, and etc., there are at most 2^i elements on the i-th level from the top which we insert in $\log M - i$ lists. The total sum is clearly $O(M)$.

Thus, the final algorithm is: if M is larger than $n \log M$, then we execute the first algorithm, otherwise, we execute the second algorithm. The overall construction works in $O(M)$ time and uses $O(n \log M)$ memory.

2.1.3. The Sequential Splay-List Analysis

We begin by stating some invariants and general properties of the splay-list.

Lemma 2. After each operation, no object can satisfy the ascent condition.

Proof. Note that we only consider the hit-operations, i.e., the operations that change $hits$ counters, because other operations do not affect any conditions. We will proceed by induction on the total number m of hit-operations on the objects of splay-list.

For the base case $m = 0$, the splay-list is empty and the hypothesis trivially holds. For the induction step, we assume that the hypothesis holds before the start of the m-th operation, and we verify that it holds after the operation completes.

First, recall that, for a fixed object u, the set S_u is defined to include all objects of the same height between u and the successor of u with height *greater* than h_u.

Specifically, we name the sum $\sum_{x \in S_u} hits(C_x^h)$ in the ascent condition as the object u's **ascent potential**. Note that after the forward pass and the increment of sh_u and $hits_v^h$ counters where v is a parent of u on height h, only the objects on the path have their ascent potential increased by one and, thus, only they can satisfy the ascent condition.

Now, consider the restructuring done on the backward pass. If the object u satisfies the descent condition, i.e., v precedes u and

$$T = hits(C_v^{h_u}) + hits(C_u^{h_u}) \leqslant \frac{m}{2^{k-h}}$$

we have to demote it. After the descent, the ascent potential of the objects between v and u on the lower level $h_u - 1$ have changed. However, these potentials cannot exceed T, meaning that these objects cannot satisfy the ascent condition.

Consider the backward pass, and focus on the set of objects at height h. We claim that only the leftmost object at that height can be promoted, i.e., its preceding object has a height greater than h. This statement is proven by induction on the backward path. Suppose that we have ℓ objects with height h on the path, which we denote by $u_1, u_2, \ldots, u_\ell$. By induction, we know that none of the objects on the path with lower height can ascend higher than h: these objects appear to the right of u_1. We know that each object was accessed at least once, $sh_{u_i} \geqslant 1$, and, thus, we can guarantee that $hits(S_{u_1}) > hits(S_{u_2}) > \ldots > hits(S_{u_\ell})$. Since the ascent potentials $hits(S_{u_i})$ are increased only by one per operation, the first and the only object that can satisfy the ascent condition is u_1, i.e., the leftmost object with the height h. If it satisfies the condition, we promote it. Consider the predecessor of u_1 on the forward path: the object v with height $h_v > h$. Object u_1 can be promoted to height h_v, but not higher, since the ascent potential of the objects on the path with height h_v does not change after the promotion of u, and only the leftmost object on that level can ascend. However, note that $hits_v^{h_v}$ can decrease and, thus, it can satisfy the descent condition, while u_1 cannot since $hits_{u_1}^h$ was equal to $hits(S_{u_1})$ before the promotion and it satisfied the ascent condition.

Because the only objects that can satisfy the ascent condition lie on the path, and we promoted necessary objects during the backward pass, no object may satisfy the ascent condition at the end of the traversal. That is exactly what we set out to prove.

Lemma 3. Given a hit-operation with argument u, the number of sub-lists visited during the forward pass is at most $3 + \log \frac{m}{sh_u}$.

Proof. During the forward pass the number of hits does not change; thus, according to Lemma 2, the ascent condition does not hold for u. Hence $sh_u \leqslant \frac{m}{2^{k-h_u-1}}$. We get that $k - h_u - 1 \leqslant \log \frac{m}{sh_u}$. Since during the forward pass $(k+1) - h_u + 1$ sub-lists are visited (notice the sentinel sub-list), the claim follows.

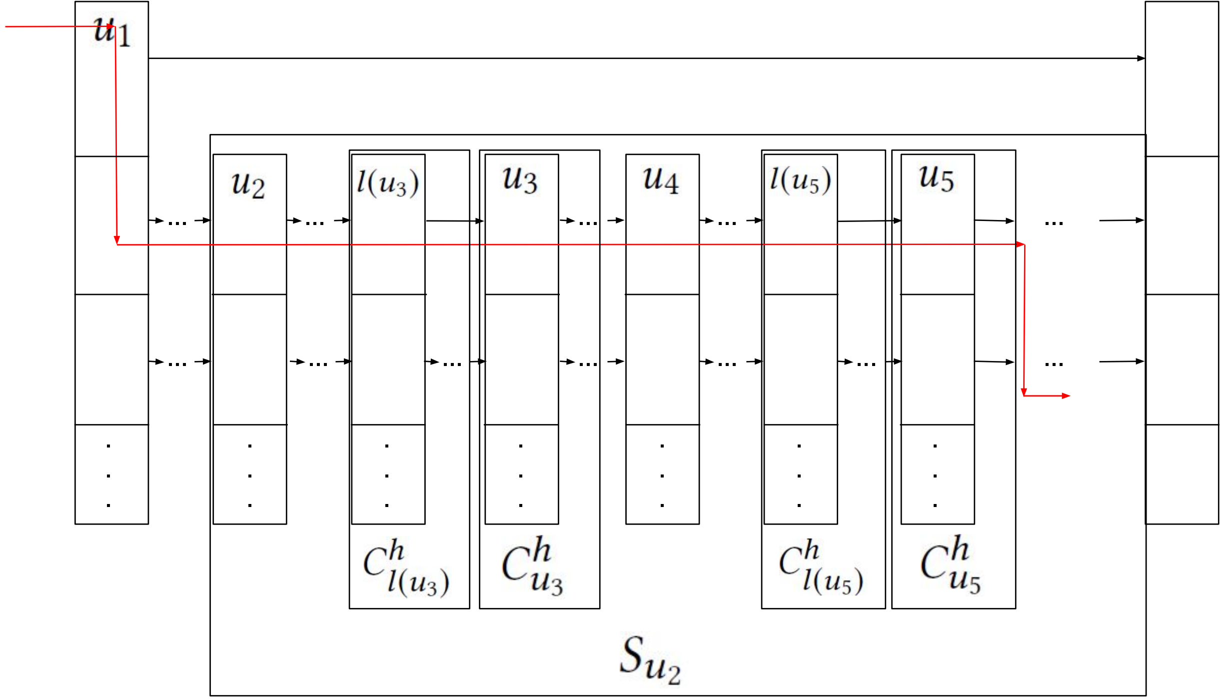


Figure 4 – Depiction of Lemma 4

Lemma 4. In each sub-list, the forward pass visits at most four objects that do not satisfy the descent condition.

Proof. Suppose the contrary and that the algorithm visits at least five objects $u_1, u_2, \ldots, u_5$ in order from left to right, that do not satisfy the descent condition in sub-list h. The height of the objects $u_2, \ldots, u_5$ is h, while the height of u_1 might be higher. See Figure 4.

Note that if the descent condition does not hold for an object u, the demotion of another object of the same height cannot make the descent condition for u satisfiable. Therefore, since the condition is not met for u_3 and u_5, the sum $hits(S_{u_2}) \geqslant (hits(C^h_{l(u_3)}) + hits(C^h_{u_3})) + (hits(C^h_{l(u_5)}) + hits(C^h_{u_5})) > \frac{m}{2^{k-h}} + \frac{m}{2^{k-h}} =$ $= \frac{m}{2^{k-h-1}}$, where $l(u_3)$ and $l(u_5)$ are the predecessors of u_3 and u_5 on height h. Note that it is possible that $l(u_3)$ and $l(u_5)$ would be the same as u_2 and u_4 respectively. This means that u_2 satisfies the ascent condition, which contradicts Lemma 2.

Note that we considered four objects since u_1 is an object of height greater than h.

Since only the leftmost object can be promoted, the backward path coincides with the forward path. Thus, the following lemma trivially holds.

Lemma 5. During the backward pass, in each sub-list h, at most four objects are visited that do not satisfy the descent condition.

Theorem 6. If d descents occur when accessing object u, the sum of the lengths of the forward and backward paths is at most $2d + 8y$, where $y = 3 + \log \frac{m}{sh_u}$.

Proof. Each object satisfying the descent condition is passed over twice, once in the forward and again in the backward pass. According to Lemma 3, there are at most y sub-lists that are visited during either passes. Excluding the descended objects, the total length of the forward path, according to Lemma 4 is $4y$. Lemma 5 gives the same result for the backward path. Hence, the total length is $2d+8y$ which is the desired result.

We can now finally state our main analytic result.

Theorem 7. The hit-operations with argument u take amortized $O\left(\log \frac{M}{sh_u}\right)$ time, where M is the total number of hits to non-marked objects of the splay-list. At the same time, all other operations take amortized $O(\log M)$ time.

Proof. We will prove the same bounds but with m instead of M. Please note that since the rebuild of the splay-list is triggered when M becomes less than $\frac{m}{2}$, we can always assume that $M \geqslant \frac{m}{2}$ and, thus, the bounds with m and M differ only by a constant.

First, we deal with the splay-list expansion procedure: it adds only $O(1)$ amortized time to an operation. The expansion happens when m is equal to the power of two and costs $O(m)$. Since, from the last expansion we performed at least $\frac{m}{2}$ hits operations we can amortize the cost $O(m)$ against them. Note that each operation will be amortized against only once, thus the amortization increases the complexity of an operation only by $O(1)$.

Since the primitive operations such as following the list pointer, a promotion with the ascent check and a demotion with the descent check are all $O(1)$, the cost of an operation is in the order of the length of the traversed path. According to Theorem 6, the total length of the traversed path during an operation is $2 \times d + 8 \times y$ where d is the number of vertices to demote and y is the number of traversed layers:

if the object u was found y is equal to $O\left(\log \frac{m}{sh_u}\right)$, otherwise, it is equal to $\log m$, the height of the splay-list.

Note that the number of promotions per operation cannot exceed the number of passed levels y, since only one object can satisfy the ascent condition per level. At the same time, the total number of demotions across all operations, i.e., the sum of all d terms, cannot exceed the total number of promotions. Thus, the amortized time of the operation can be bounded by O(number of levels passed) which is equal to what we required.

The amortized bound for `delete` operation needs some additional care. The operation can be split into two parts:

a) find the object in the splay-list, mark it as deleted and adjust the path;
b) the reconstruction part when the object is physically deleted.

The first part is performed in $O(\log \frac{m}{sh_u})$ as shown above. For the second part, we perform the reconstruction only when the number of hits on objects marked for deletion $m - M$ exceeds the number of hits on all objects m, and, thus, $M \leqslant \frac{m}{2}$. The reconstruction is performed in $O(M) = O(m)$ time as explained in Section 2.1.2. Thus we can amortize this $O(m)$ to hits operations performed on logically deleted items. Since there were $O(m - M) = O(m)$ such operations, the amortization "increases" their complexities only on some constant and only once, since after the reconstruction the corresponding objects are going to be deleted physically.

For example, if all our operations were successful `contains`, then the asymptotics for `contains(`u`)` will be $O(\log \frac{m}{sh_u})$ where m is the total number of operations performed.

Furthermore, under the same load we can prove the static optimality property [6]. Let $m_i \leqslant m$ be the total number of operations when we executed i-th operation on u, then the total time spent on operations with argument u is

$$O\left(\sum_{i=1}^{sh_u} \log \frac{m_i}{i}\right) = O\left(\sum_{i=1}^{sh_u} \log \frac{m}{i}\right)$$

which by Lemma 3 from [3] is equal to $O(sh_i + sh_i \times \log \frac{m}{sh_i})$. This is exactly the static optimality property.

2.2. The Sequential Distribution-Adaptive Interpolation Search Tree

The distribution-adaptive interpolation search tree design builds on the IST described in [8].

The overall idea, is that we can logically treat each access as a separate element. So, let's define a modified version of the IST.

Let a and b be reals, $a < b$. An Interpolation Search Tree for a parameter α, $0 < \alpha \leqslant \frac{1}{2}$ with boundaries a and b for a set $S = \{x_1 < x_2 < \ldots < x_n\} \subseteq [a, b]$ of n elements, with $ac_1, ac_2, \ldots, ac_n$ accesses made to respective elements, consists of:

a) An integer m—the total number of accesses to all elements $x_1, x_2, \ldots, x_n$.
b) A set REP of representatives $x_{i_1}, x_{i_2}, \ldots x_{i_k}$, $i_1 < i_2 < \ldots < i_k$ stored in an array $REP[1 \ldots k]$, that is, $REP[j] = x_{i_j}$. Furthermore, k not more than $2 \times m^{\alpha}$.
c) An array $AC[1 \ldots k]$—the number of accesses made to $x_{i_1}, x_{i_2}, \ldots x_{i_k}$ respectively.
d) An array $ID[1 \ldots lenId]$, where $lenId$ is some integer, with $ID[i] = j$ if $REP[j] < a + i \times \frac{b-a}{lenId} \leqslant REP[j+1]$.
e) Interpolation search trees $T_1, T_2, \ldots, T_{k+1}$ for the subfiles $S_1, S_2, \ldots, S_{k+1}$ where $S_j = \{x_{i_{j-1}+1}, \ldots x_{i_j-1}\}$ for $2 \leqslant j \leqslant k$, $S_1 = \{x_1, \ldots x_{i_1-1}\}$ and $S_{k+1} = \{x_{i_k+1}, \ldots x_n\}$, and the respective subsequences of ac. Furthermore, tree T_j, $2 \leqslant j \leqslant k$, has boundaries $x_{i_{j-1}}$ and x_{i_j}, T_1 has boundaries a and x_{i_1} and T_{k+1} has boundaries x_k and b.

2.2.1. Ideal IST and its properties

Obviously, we have to modify the definition of the ideal IST. An IST for set S, $|S| = n$, and number of accesses $ac_1, ac_2, \ldots, ac_n$ to $x_1, x_2, \ldots, x_n$ respectively, is *ideal* for a parameter α, $0 < \alpha \leqslant \frac{1}{2}$, if: (1) for each $j > 0$, i_j is the first element to the right of i_{j-1} such, that the number of accesses made to elements between it and i_{j-1}, including i_j is at least $m^{1-\alpha}$; and (2) if the interpolation search trees $T_1, T_2, \ldots, T_{k+1}$ are also *ideal*.

We are using m for the number of accesses made to all elements of an IST and $m(T_i)$ for the number of accesses made to all elements of a subtree T_i.

The number of representatives stored in a root of the subtree T_i of an ideal IST does not exceed $m(T_i)^{\alpha}$. This happens since the number of accesses made to all elements between two adjacent representatives is at least $m(T_i)^{1-\alpha}$ and, by the

Dirichlet principle, there cannot be more than $\frac{m(T_i)}{m(T_i)^{1-\alpha}} = m(T_i)^{\alpha}$ representatives. Also, it is obvious from the definition that the sum of accesses in the subtree T_i of tree T, $m(T_i) \leqslant m(T)^{1-\alpha}$.

We start with the algorithm on how to `build the ideal IST` for an ordered set augmented with the array of accesses. At the beginning, we count prefix sums of all accesses made to an ordered set. We do this once before counting representatives and ID arrays for each level. We use binary search on the prefix sums to find the next representative and recursively build the subtrees. Then, we use simple loop through the array of representatives to build array ID.

Let us prove the several properties of that algorithm and the ideal IST.

Lemma 8. An ideal IST for a parameter α for an ordered set and array of accesses can be built in time $O(m)$ and requires $O(m^{\alpha\times(1-\alpha)} \times n + m^{\alpha})$ memory. It has depth $O(\log\log m)$.

Proof. Let $Time(m)$ be the time needed to build an ideal IST, then

$$Time(m) \leqslant m^{\alpha} + \log_2(m) \times m^{\alpha} + (m^{\alpha} + 1) \times Time(m^{1-\alpha}) :$$

we spend no more than $\log_2(m) \times m^{\alpha}$, to get all the representatives using binary searches, m^{α} is spent to count ID array, and there are no more than $m^{\alpha} + 1$ subtrees T_i and for each of them $m(T_i) \leqslant m^{1-\alpha}$. Our tree is internal. So, every node is chosen as a representative only once and there are no more than m representatives. Also, each binary search spends no more than $\log_2(m)$ to find a new representative. So, $Time(m) \leqslant m \times \log_2(m)$. So, $Time(m) = 2 \times \log_2(m) \times m^{\alpha} + m^{\alpha} \times Time(m^{1-\alpha}) + o(m)$. The fact that it is $O(m)$ is proved in Lemma 9.

Now, let us count the required memory. It is obvious that the first level of IST requires $O(m^{\alpha})$ space. Also, the total number of nodes in IST does not exceed $3 \times n$, because our tree is internal and, thus, each node has at least one value inside. Now, note that for each node with subtree T_i on lower levels satisfies $m(T_i) < m^{1-\alpha}$. So, the lengths of REP and ID for each of these nodes are no more than $m^{\alpha\times(1-\alpha)}$, and, thus, the memory required for all of the nodes, except from the root, does not exceed $O(m^{\alpha\times(1-\alpha)} \times n)$. If sum everything up, we get that the memory does not exceed $O(m^{\alpha\times(1-\alpha)} \times n + m^{\alpha})$.

Let $d(m)$ be the depth of the IST for a parameter α, then $d(m) = 1 + d(m^{1-\alpha})$. So the IST with a parameter α has depth $O(\log\log m)$.

Lemma 9. If $T(n) = c \times \log_2(n) \times n^{\alpha} + n^{\alpha} \times T(n^{1-\alpha})$ where $0 < \alpha < 1$, then $T(n) = O(n)$ for $n \in \mathbb{N}$.

Proof. To simplify the presentation we choose n greater than 4. Consider $F(n) = \frac{T(n)}{n}$, then it is obvious that $F(n) = c \times \frac{\log_2(n)}{n^{1-\alpha}} + F(n^{1-\alpha})$. So $F(n) = c \times \sum\limits_{k=1}^{\infty} \frac{\log_2(n^{(1-\alpha)^{k-1}})}{n^{(1-\alpha)^k}}$, which means that $F(n) = c \times \sum\limits_{k=1}^{\infty} \frac{(1-\alpha)^{k-1} \times \log_2 n}{n^{(1-\alpha)^k}}$.

Consider two functions

$$Q(n) = \beta \times \frac{F(n)}{\log_2(n)} = \sum_{k=1}^{\infty} \frac{\beta^k}{n^{\beta^k}}$$

and

$$f(x) = \frac{\beta^x}{n^{\beta^x}},$$

where $\beta = 1 - \alpha$. We want to find intervals on which $f(x)$ is monotonous. For that we take a derivative

$$f'(x) = -\beta^x \times \ln \beta \times n^{-\beta^x} \times (\beta^x \times \ln n - 1)$$

and compare it with 0. One can see that $f'(x) = 0$ only if $x = \log_\beta \frac{1}{\ln n} = x_0$. Thus, the function f has only one point where the monotony changes. On the left of x_0 the derivative is positive. To prove that we substitute x with $x_0 - 1$:

$$f'(x_0 - 1) = f'(\log_\beta \frac{1}{\ln n} - 1) = -\frac{1}{\beta} \times \frac{1}{\ln n} \times \ln \beta \times n^{-\frac{1}{\beta} \times \frac{1}{\ln n}} \times (\frac{1}{\beta} - 1).$$

All the multipliers are positive except for $\ln \beta < 0$ which gives us the positive value in total. Similarly, we substitute x with $x_0 + 1$ and get the negative value. So, f' was positive and then negative, which means that x_0 is the point of the maximum.

Now, consider our target function $Q(n)$.

$$Q(n) = \sum_{k=1}^{\infty} \frac{\beta^k}{n^{\beta^k}} \leqslant \int_1^{\infty} f(x)\, dx + 2f(x_0)$$

.

$$\int f(x)dx = \int \frac{\beta^x}{n^{\beta^x}}\, dx = -\frac{n^{-\beta^x}}{\ln \beta \times \ln n} + C.$$

So, $Q(n) \leqslant \int_1^\infty f(x)dx + 2 \times f(x_0) = -\frac{1}{\ln\beta \times \ln n} + \frac{n^{-\beta}}{\ln\beta \times \ln n} + 2 \times \frac{1}{\ln n \times n^{\frac{1}{\ln n}}} \leqslant$
$\leqslant -\frac{1}{\ln\beta} \times \frac{1}{\ln n} + \frac{1}{\ln n} \times 2 = \frac{1}{\ln n} \times (2 - \frac{1}{\ln\beta})$

Now, if we count $T(n)$ using $Q(n)$, we get $T(n) = c \times Q(n) \times n \times \log_2 n \times \frac{1}{\beta} \leqslant$
$\leqslant c \times \frac{1}{\ln n} \times (2 - \frac{1}{\ln\beta}) \times n \times \frac{\ln n}{\ln 2} \times \frac{1}{\beta} \leqslant c \times n \times (2 - \frac{1}{\ln\beta}) \times \frac{1}{\beta \times \ln 2} = c \times n \times (2 - \frac{1}{\ln(1-\alpha)}) \frac{1}{(1-\alpha) \times \ln 2}$.
Since α is constant we get that $T(n) = O(n)$.

2.2.2. Insert, Delete and Rebuild operations

`Insert`, `delete` and `contains` are done in the same way as in [8]. Except we use operation counters m, im instead of $size$, $isize$, and rebuild a subtree when the number of operations from the last rebuilding exceeds $\frac{im}{4}$. The number of accesses for nodes is changed for each delete, insert, and contains operation.

Now, let us consider how the asymptotics changes for the modified version of the algorithm.

Lemma 10. The worst-case depth of an IST for a parameter α for a file of n keys is $O(\log m)$. It requires $O(m^\alpha \times n)$ space.

Proof. At first, we prove the bound on the depth. Consider a node v and its parent w. Then $m(T_v)$ (current number of accesses made to that subtree) is not more, than $\frac{im(T_w)}{4}$, which is the maximum number of accesses made to its parent since last rebuild, plus $im(T_w)^{1-\alpha}$ which is the number of accesses in it after the last rebuild of its parent. So, $m(T_v) \leqslant \frac{im(T_w)}{4} + im(T_w)^{1-\alpha} \leqslant \frac{im(T_w)}{2}$. So, the depth is obviously $O(\log m)$.

At any moment, each node requires no more than $O(m^\alpha)$ memory per ID and REP, because the amount of memory which is used by node doesn't change between rebuild operations and new nodes take only constant memory. Also, the number of nodes is proportional to number of elements in the data structure. So, the total memory is $O(m^\alpha \times n)$.

Lemma 11. The amortized number of searches on level of `contains`, `insert` or `delete` is $O(\log m)$. The total expected amortized number of searches on level of the first m contains, insertions and deletions is $O(m \times \log m)$.

Proof. We have to amortize the rebuild over all operations. For that we use the following accounting scheme is used: every operation puts one token on each node on the path to the target element. Each operation puts no more than $O(\log m)$ tokens by Lemma 10. Suppose that $C(v)$ be the number of tokens in node v holds $C(v)$. Since the total number of operation before the rebuilding to the subtree are $\frac{m(T_v)}{4}$,

we have that $C(v) \geqslant \frac{m(T_v)}{4}$. We know that the rebuild operation takes $O(m(T_v))$, so we have enough tokens to amortize over it.

2.2.3. Searching in the distribution-adaptive IST

In this subsection we discuss how to search in our IST in $O\Big(\log \frac{\log m}{\log ac(u)}\Big)$ expected time and we prove several facts which are important for the analysis.

Let μ be the probability density function on reals. A random file of size n is generated by drawing independently n reals according to density μ. A `random IST` for a parameter α with number of accesses m to all its elements is generated as follows:

a) Take a random file F of size n' for some n' and build an ideal IST with a parameter α (the number of accesses to each element equals to the number of its occurrences in the file).

b) Perform a sequence of i μ-random insertions, d random deletions and c μ-random contains $Op_1, \ldots, Op_{i+c+d}$. Such that the number of accesses on non-deleted elements is m. An insertion is μ-random if it inserts a random real drawn according to density μ into the tree. A deletion is random if it deletes a random element from the tree, all elements in the tree are equally likely to be chosen for the deletion. A contains operation is μ-random if it finds a random real drawn according to density μ in the tree.

The following lemma can be proven the same way as in [8].

Lemma 12. Let μ be a density function with a finite support, let T be a μ-random IST for a parameter α with boundaries a and b and let T' be a subtree of T. Then there are reals c, d such that T' is a $\mu[c, d]$-random IST, where $a \leqslant c < d \leqslant b$, $\mu[c, d](x) = 0$ for $x < c$ or $x > d$ and $\mu[c, d](x) = \frac{\mu(x)}{\int_c^d \mu(x)}$ for $c \leqslant x \leqslant d$.

Now, we can state the following important lemmas.

Lemma 13. Let μ be a density function with a finite support $[a, b]$, let T be a μ-random IST for a parameter α with number of accesses to all its elements equal to m, and let T' be a direct subtree of T. Then, $m(T')$ is $O(m^{1-\alpha})$ with probability at least $1 - O(m^{-(1-\alpha)})$.

Proof. Each access can be treated as a separate element with only one access. Looking at the world from that point of view, a definition of an ideal IST is the same as in [8] and this theorem is even easier than the respective one. We repeat the proof of the corresponding lemma from [8], except that $m_0^{1-\alpha}$ is used instead of $\sqrt{m_0}$. The

only change for the theorem from the Appendix of [8] is that in the case of the same values an additional key sampled uniformly at random, so they can be in any order in the sorted sequence with equal probability.

Lemma 14. Let μ be a density with finite support $[a, b]$. Then the expected total number of searches on level for processing a sequence of m μ-random insertions and μ-random contains to an initially empty IST is $O(m \log\log m)$, that is, expected amortized number of searches on level of insertion and contains is $O(\log\log m)$.

Proof. Let $f(m)$ be the expected number of tokens put down by the m-th operation. Then

$$f(m) \leqslant 1 + f\Big(O\Big(m^{1-\alpha}\Big)\Big) + O\Big(m^{-(1-\alpha)}\Big) \times O\Big(\log m\Big).$$

This can be seen as follows: If the operation goes into a subtree of size $O(m^{1-\alpha})$, then we put down $1 + f(O(m^{1-\alpha}))$ tokens. If it does not, then we put down at most $O(\log m)$ tokens by Lemma 10. The probability of the latter event is $O(m^{-(1-\alpha)})$ by Lemma 13. Thus, $f(m) = O(\log\log m)$.

The search on the level is made in the same manner as in [8]. However, we have to change the definition of *smoothness* of the density since we have a little bit different setting.

A density μ is `smooth` for a parameter α, $0 < \alpha < 1$, if there are constants a, b and d such that $\mu(x) = 0$ for $x < a$ and $x > b$ and such that for all c_1, c_2, c_3, $a \leqslant c_1 < c_2 < c_3 \leqslant b$, and all integers n and m with $m = \lceil n^{\alpha} \rceil$,

$$\int_{c_2 - \frac{c_3 - c_1}{m}}^{c_2} \mu[c_1, c_3](x)\, dx \leqslant d \times n^{-\alpha},$$

where $\mu[c_1, c_3](x) = 0$ for $x < c_1$ or $x > c_3$ and $\mu[c_1, c_3](x) = \frac{\mu(x)}{p}$ for $c_1 \leqslant x \leqslant c_3$ where $p = \int_{c_1}^{c_3} \mu(x) dx$. This definition of smoothness is still pretty wide, and distributions like `zipf`, `90/10`, `95/5` and `99/1` are smooth for any parameter. The following lemma proves it.

Lemma 15. Let μ be a probability density function that operates on $[a, b]$ and which values on $[a, b]$ lies in $[x, y]$, where $x > 0$. Then, μ is smooth for any parameter α ($0 < \alpha < 1$).

Proof. At first, we fix a parameter α, n and $m = \lceil n^{\alpha} \rceil$. Then, consider any triple of reals c_1, c_2, c_3 with $a \leqslant c_1 < c_2 < c_3 \leqslant b$. Let us look on the integral from

the definition of smoothness. Now, denote $c_3 - c_1$ as len, and $\frac{c_3-c_1}{m}$ as $mlen$. Then, $\int_{c_2-\frac{c_3-c_1}{m}}^{c_2} \mu[c_1,c_3](x)\,dx = \int_{c_2-mlen}^{c_2} \mu[c_1,c_3](x)\,dx \leqslant \frac{y\times mlen}{y\times mlen + x\times(len-mlen)} =$
$= \frac{y\times len}{y\times len + x\times len\times(m-1)} = \frac{y}{y+x\times(m-1)}$.

By that we upperbounded the integral. Now, we have to find d that satisfies the definition. For that, we check an upper bound on $\frac{y}{y+x\times(m-1)} \times n^{\alpha}$. We have two cases where $n < (\lceil\frac{y}{x}\rceil + 1)^{\frac{1}{\alpha}}$ and where $n \geqslant (\lceil\frac{y}{x}\rceil + 1)^{\frac{1}{\alpha}}$.

In first case

$$\frac{y}{y+x\times(m-1)} \times n^{\alpha} \leqslant \frac{y}{y} \times n^{\alpha} \leqslant n^{\alpha} < \lceil\frac{y}{x}\rceil + 1.$$

In second case

$$\frac{y}{y+x\times(m-1)} \times n^{\alpha} \leqslant \frac{y}{x\times(m-1)} \times n^{\alpha} \leqslant \frac{2\times y}{x\times n^{\alpha}} \times n^{\alpha} \leqslant 2\times\frac{y}{x}.$$

Therefore

$$\int_{c_2-\frac{c_3-c_1}{m}}^{c_2} \mu[c_1,c_3](x)\,dx \leqslant \frac{y}{y+x\times(\lceil n^{\alpha}\rceil - 1)} \leqslant \max\left(\left\lceil\frac{y}{x}\right\rceil + 1, 2\times\frac{y}{x}\right) \times \frac{1}{n^{\alpha}}$$

for every n, c_1, c_2, c_3. So, there exist constant d for which for every n, c_1, c_2, c_3 the value of integral is no more than $\frac{d}{n^{\alpha}}$. So, μ is smooth for a chosen parameters α and α.

Lemma 16. Let μ be a smooth density for a parameter α and let T be a μ-random IST for a parameter α. Then the expected search time in the root array is $O(1)$.

Proof. Suppose that we spent $l+1$ iterations while searching for an element in the root array. Similar to the reasoning in [8], at least $t = l\times m_0^{1-\alpha}$ accesses lie in the interval of ID $\left[y - \frac{b-a}{m}, y\right]$, where m_0 is the number of accesses before the last rebuild. The probability of this event is at most $\left(\frac{3\times d}{l}\right)^t$ by Lemma 17 , and, hence, the expected number of iterations over the array is

$$\sum_{l\geqslant 1} \min\left(1, \left(\frac{3\times d}{l}\right)^t\right) \leqslant 6\times d + \sum_{l\geqslant 1}\left(\frac{1}{2}\right)^t \leqslant 6\times d + \sum_{l\geqslant 1}\left(\frac{1}{2}\right)^l \leqslant 6\times d + 1 = O(1)$$

Lemma 17. Let μ be a smooth density for a parameter α, with support $[a, b]$. Let $y \in [a, b]$, let F be a μ-random file of size n_0, and let $m = \lceil n_0^\alpha \rceil$. Then

$$pr\Big(\Big|F \cap \Big[y - \frac{b-a}{m}, y\Big]\Big| \geqslant l \times n_0^{1-\alpha}\Big) \leqslant \Big(\frac{3 \times d}{l}\Big)^{l \times n_0^{1-\alpha}}$$

Proof. To proof it we use the technique given in [8]. Let

$$p = \int_{y - \frac{b-a}{m}}^{y} \mu(x) dx.$$

$p \leqslant d \times n_0^{-\alpha}$ for some constant d since μ is smooth. Let $F = X_1, X_2, \ldots, X_{n_0}$. Remember that we use t instead of $l \times n_0^{1-\alpha}$. If $\Big|F \cap \Big[y - \frac{b-a}{m}, y\Big]\Big| \geqslant t$ then there must be at least t X_i's with $X_i \in \Big[y - \frac{b-a}{m}, y\Big]$. We know that for each particular X_i $Pr\Big[X_i \in \Big[y - \frac{b-a}{m}, y\Big]\Big] = p$. Thus,

$$Pr\Big[\Big|F \cap \Big[y - \frac{b-a}{m}, y\Big]\Big| \geqslant t\Big] \leqslant \binom{n_0}{t} \times p^t \leqslant \frac{n_0^t}{t!} \times (d \times n_0^{-\alpha})^t$$

Finally, if we use Stirling approximation $s! \geqslant (\frac{s}{e})^s$:

$$\leqslant \Big(\frac{n_0 \times e \times d}{l \times n_0^{1-\alpha} \times n_0^{\alpha}}\Big)^t \leqslant \Big(\frac{e \times d}{l}\Big)^t \leqslant \Big(\frac{3 \times d}{l}\Big)^{l \times n_0^{1-\alpha}}.$$

Now, we are ready to proof the bound on the expected search time.

Theorem 18. Let μ be a smooth density for a parameter α. Then the expected search time of an element with the number of accesses ac in a μ-random IST for a parameter α is $O(\log \frac{\log m}{\log ac})$.

Proof. Let $T(m)$ be the expected search time in a μ-random IST of size m. Then

$$T(m) = O(1) + T\Big(O\Big(m^{1-\alpha}\Big)\Big) + O(m^{-(1-\alpha)}) \times O(m^{(1-\alpha)}).$$

The search time in the root array is $O(1)$ by Lemma 16. The next subtree in which we search is μ-random by Lemma 12. Its total number of accesses is $O(m^{1-\alpha})$ with probability exceeding $1 - O(m^{-(1-\alpha)})$ by Lemma 13, otherwise the search time is bounded by $O(m^{1-\alpha})$.

The search stops when the total number of accesses to a subtree is less or equal to ac. So, $T(m) = O(\log \frac{\log m}{\log ac})$

Conclusions on Chapter 2

In this chapter, we presented two new designs based on the skip-list and IST. Then, we proved the amortized running time of the operations on these data structures. By that, we showed that the splay-list holds static optimality property and that its time and memory complexities are not worse, than for the CBTree, which is the only existing concurrent self-adjusting data-structure. For the self-adjusting IST, we proved the better expected complexity for the search on a big class of access distributions, but its worst case memory bound is worse than one for the CBTree and for the splay-list.

CHAPTER 3. THE CONCURRENT DISTRIBUTION-ADAPTIVE DATA STRUCTURES

In this chapter we discuss how to make concurrent versions of algorithms, presented in the previous chapter. Also, we describe an updated version of the splay-list.

3.1. The Concurrent Splay-List

In this section we describe on how to implement scalable lock-based implementation of the splay-list described in the previous section.

3.1.1. Overview

The first idea, on how to make splay-list concurrent, that comes to the mind is to implement the operations as in Lazy Skip-list [5]: (1) we traverse the data structure in a lock-free manner in the search of x and fill the array of predecessors of x on each level; (2) if x is not found then the operation stops; otherwise, we try to lock all the stored predecessors; if some of them are no longer the predecessors of x we find the real ones or, if not possible, we restart the operation; (3) when all the predecessors are locked we can traverse and modify the backwards path using the presented sequential algorithm without being interleaved. When the total number of operations m becomes a power of two, we have to increase the height of the splay-list by one: in a straightforward manner, we have to take the lock on the whole data structure and then rebuild it.

There are several major issues with this straightforward implementation. At first, the *balancing* part of the operation is too coarse-grained—there are a lot of locks to be taken and, for example, the lock on the topmost level forces the operations to serialize. The second is that the list expansion by freezing the data structure and the following rebuild when m exceeds some power of two is very costly.

3.1.2. The Relaxed Rebalancing Analysis

If we build the straightforward concurrent implementation on top of the sequential implementation described in the previous section, it will obviously suffer in terms of performance since each operation (either `contains`, `insert`, or `delete`) must take locks on the whole path to update hits counters. This is not a reasonable approach, especially in the case of the frequent `contains` operation. Luckily for us, `contains` can be split into two phases: the *search* phase, which

traverses the splay-list and is lock-free, and the *balancing* phase, which updates the counters and maintains ascent and descent conditions.

A straightforward heuristic is to perform rebalancing infrequently—for example, only once in c operations. For this, we propose that the operation perform the rebalancing, i.e., update of the hit counters with ascents and descents, only with a fixed probability $1/c$. Conveniently, if the operation does not perform the global operation counter update and the balancing, the counters will not change and, so, all the conditions will still be satisfied. The only remaining question is how much this relaxation will affect the guarantees of the data structure. The next result characterizes the effects of this relaxation.

Theorem 19. Fix a parameter $c \geqslant 1$. In the relaxed sequential algorithm where operation updates hits counters and performs balancing with probability $\frac{1}{c}$, the hit-operation takes $O\left(c \times \log \frac{m}{sh_u}\right)$ expected amortized time, where m is the total number of hit-operations performed on all objects in splay-list up to the current point in the execution.

Proof. The theoretical analysis above (Theorems 6 and 7) is based on the assumption that the algorithm maintains exact values of the counters m and sh_u — the total number of hit-operations performed to the existing objects and the current number of hit-operations to u. However, given the relaxation, the algorithm can no longer rely on m and sh_u since they are now updated only with probability c. We denote by m' and sh'_u the relaxed versions of the real counters m and sh_u.

The proof consists of two parts. First, we show that the amortized complexity of hits operation to u is equal to $O\left(c \times \log \frac{m'}{sh'_u}\right)$ in expectation. Secondly, we show that the approximate counters behave well, i.e., $\mathbb{E}\left[\log \frac{m'}{sh'_u}\right] = O\left(\log \frac{m}{sh_u}\right)$. Bringing these two together yields that the amortized complexity of hits operations is $O\left(c \times \log \frac{m}{sh_u}\right)$ in expectation.

The first part is proven similarly to Theorem 7. We start with the statement that follows from Theorem 6: the complexity of any contains operation is equal to $2d + 8y$ where d is the number of objects satisfying the descent condition and $y = 3 + \log \frac{m'}{sh'_u}$. Obviously, we cannot use the same argument as in Theorem 7 since now d is not equal to the number of descents: the objects which satisfy the descent condition are descended only with probability $\frac{1}{c}$. Thus, we have to bound the sum of d by the total number of descents.

Consider some object x that satisfies the descent condition, i.e., it is counted in d term of the complexity. Then x will either be descended, or will not satisfy the descent condition after c operations passing through it in expectation. Mathematically, the event that x is descended follows an exponential distribution with success (demotion) probability $\frac{1}{c}$. Hence, the expected number of operations that traverse x before it is descended is c. This means that the object x will be counted in terms of type d no more than c times in expectation. By that, the total complexity of all operations is equal to the sum of $8y$ terms plus $2c$ times the number of descents. Since the number of descents cannot exceed the number of ascents, which in turn cannot exceed the sum of the y terms, the total complexity does not exceed the sum of $10 \times c \times y$ terms. Finally, this means that the amortized complexity complexity of hits operation is $O(c \times y) = O\left(c \times \log \frac{m'}{sh'_u}\right)$ in expectation.

Next, we prove the second main claim, i.e., that

$$\mathbb{E}\left(\log \frac{m'}{sh'_u}\right) = O\left(\log \frac{m}{sh_u}\right).$$

Note that the relaxed counters m' and sh'_u are Binomial random variables with probability parameter $p = \frac{1}{c}$, and number of trials m and sh_u, respectively.

To avoid issues with taking the logarithm of zero, let us bound $\mathbb{E}\left(\log \frac{m'+1}{sh'_u+1}\right)$, which induces only a constant offset. We have:

$$\begin{aligned}
\mathbb{E}\left[\log \frac{m'+1}{sh'_u+1}\right] &= \mathbb{E}\left[\log(m'+1)\right] - \mathbb{E}\left[\log(sh'_u+1)\right] \\
&\underset{\text{Jensen}}{\leqslant} \log(\mathbb{E}m'+1) - \mathbb{E}\log(sh'_u+1) \\
&= \log(mp+1) - \mathbb{E}\log(sh'_u+1).
\end{aligned}$$

The next step in our argument will be to lower bound $\mathbb{E}\log(sh'_u+1)$. For this, we can use the observation that $sh'_u \sim Bin_{sh_u,p}$, the Chernoff bound, and a careful derivation to obtain the following result.

Lemma 20. If $X \sim Bin_{n,p}$ and $np \geqslant 3n^{2/3}$ then $\mathbb{E}\left[\log(X+1)\right] \geqslant \log np - 4$.

Proof. Recall the standard Chernoff bound, which says that if $X \sim Bin_{n,p}$, then $P(|X - np| > \delta np) \leqslant 2e^{-\mu\delta^2/3}$. Applying this with $\delta = \frac{1}{n^{1/3}p}$, we obtain $P(|X - np| > n^{\frac{2}{3}}) \leqslant 2e^{-\frac{n^{1/3}}{3p^2}}$.

$$\mathbb{E}\log(X+1) = \mathbb{E}\log(np + (X - np + 1)) = \log np + \mathbb{E}\log\left(1 + \frac{X-np+1}{np}\right)$$
$$= \log np + \sum_{k=0}^{n} p_k \log\left(1 + \frac{k-np+1}{np}\right) \underset{\substack{\text{Taylor series and} \\ 1+\frac{k-np+1}{np} \geqslant \frac{1}{np}}}{\geqslant} \log np +$$
$$+ \sum_{k=np-n^{2/3}}^{np+n^{2/3}} p_k \left(\frac{k-np+1}{np} - \frac{(k-np+1)^2}{2n^2p^2} + \ldots\right) + P(|X - np| > n^{\frac{2}{3}}) \times \log\frac{1}{np} \geqslant \log np -$$
$$- \sum_{k=np-n^{2/3}}^{np+n^{2/3}} p_k \left(\frac{2n^{2/3}}{np} + \frac{(2n^{2/3})^2}{2(np)^2} + \ldots\right) - 2\log np \times e^{-\frac{n^{1/3}}{3p^2}} \underset{\sum_{k=np-n^{2/3}}^{np+n^{2/3}} p_k \leqslant 1}{\geqslant} \log np -$$
$$-\left(\frac{2n^{2/3}}{np} + \frac{(2n^{2/3})^2}{(np)^2} + \ldots\right) - 2\log np \times e^{-\frac{n^{1/3}}{3p^2}} = \log np - \frac{np}{np - 2n^{2/3}} - 2\log np \times e^{-\frac{n^{1/3}}{3p^2}} \geqslant$$
$$\geqslant \log np - 3 - 2\log np \times e^{-\frac{n^{1/3}}{3p^2}} \geqslant \log np - 4.$$

Based on this Claim we obtain

$$\log(mp+1) - \mathbb{E}[\log(sh'_u + 1)] \leqslant \log(mp+1) - \log(sh_u \times p) + 4 \leqslant \log\frac{m}{sh_u} + 5.$$

However, this bound works only for the case when $sh_u \times p \geqslant 3 \times (sh_u)^{2/3}$. Consider the opposite: $sh_u \leqslant \frac{27}{p^3}$. Then, $\mathbb{E}[\log(sh'_u + 1)] \geqslant 0 \geqslant \log sh_u - \log\frac{27}{p^3}$. Note that the last term is constant, so we can conclude that $\mathbb{E}[\log\frac{m'+1}{sh'_u+1}] \leqslant \log\frac{m}{sh_u} + C$. This matches our initial claim that $\mathbb{E}[\log\frac{m'+1}{sh'_u+1}] = O(\log\frac{m}{sh_u})$.

3.1.3. Relaxed and Forward Rebalancing

The first problem from Section 3.1.1 can be fixed in two steps. The most important one is to relax guarantees and perform *rebalancing* only periodically, for example, with probability $\frac{1}{c}$ for each operation. Of course, this relaxation will affect the bounds—please see Section 3.1.2 for the proofs.

However, this relaxation is not sufficient, since we cannot relax the balancing phase of `insert`(u) which physically links an object. All these `insert` functions are going to be serialized due to the lock on the topmost level. Note that without further improvements we cannot avoid taking locks on each predecessor of x, since we have to update their counters.

We would like to have more fine-grained implementation. However, our current sequential algorithm does not allow this, since it updates the path only backwards and, thus, needs the whole path to be locked. To address this issue, we introduce a different variant of our algorithm, which does rebalancing *on the forward traversal*.

We briefly describe how this *forward-pass algorithm* works. We maintain the basic structure of the algorithm. At first, we make a lock-free traversal to find x. Only after this we perform forward traversal with rebalancing if necessary. We traverse the splay-list in the search of x, and suppose that we are now at the last node v on the level h which precedes x. The only node on level $h-1$ which can be ascended is v's successor on that level, node u: we check the ascent condition on u or, in other words, compare $\sum_{w \in S_u} hits(C_w^{h-1}) = hits_v^h - hits_v^{h-1}$ with $\frac{m}{2^{k-h}}$, and promote u, if necessary. Then, we iterate through all the nodes on the level $h-1$ while the keys are less than x: if the node satisfies the descent condition, we demote it. Note that the complexity bounds for that algorithm are the same as for the previous one and can be proven exactly the same way (see Theorem 7).

The main improvement brought by this forward-pass algorithm is that now the locks can be taken in a hand-over-hand manner: take a lock on the highest level h and update everything on level $h-1$; take a lock on level $h-1$, release the lock on level h and update everything on level $h-2$; take a lock on level $h-2$, release the lock on level $h-1$ and update everything on level $h-3$; and so on. By this locking pattern, the balancing part of different operations is performed in a sequential manner: an operation cannot overtake the previous one and, thus, the $hits$ counters cannot be updated asynchronously. However, at the same time we reduce contention: locks are not taken for the whole duration of the operation.

3.1.4. Lazy Expansion.

The expansion issue is resolved in a lazy manner. The splay-list maintains the counter $zeroLevel$ which represents the current lowest level. When m reaches the next power of two, $zeroLevel$ is decremented, i.e., we need one more level. (To be more precise, we decrement $zeroLevel$ also lazily: we do this only when some node is going to be demoted from the current lowest level.) Each node is allocated with an array of $next$ pointers with length 64 (as discussed, the height 64 allows us to perform 2^{64} operations which is more than enough) and maintains the lowest level to which the node belonged during the last traverse. When we traverse a node and it appears to have the lowest level higher than $zeroLevel$, we update its lowest level and fill the necessary cells of $next$ pointers. By doing that we make a lazy expansion of splay-list and we do not have to freeze whole data structure to rebuild.

For the pseudo-code of the splay-list, we refer to the Section 4.1.

3.2. The Concurrent Distribution-Adaptive Interpolation Search Tree

We use the same ideas as described in [14] for the IST to make the concurrent version of self-adjusting interpolation search tree. The only difference is that we store the number of accesses and how it changes, instead of storing sizes. So, in *markAndCount* [14] function we count the number of accesses made to the subtree and choose representatives on the first level using it. Then, we do exactly the same trick with the collaborative rebuild described in [14], except we use our modified ideal builder, which uses binary searches. See Section 2.2.1.

Conclusions on Chapter 3

In this chapter, we developed the concurrent designs of splay-list and self-adjusting interpolation search tree. To do this, we came up with the forward-pass rebalancing algorithm instead of backward pass from the previous chapter. Also, we proved that if we rebalance the splay-list only with some probability, than the expected hit-operation time would be $O\left(c \times \log \frac{m}{sh_u}\right)$.

Overall, we created a simple lock-based design for the splay-list and a lock-free design for the self-adjusting IST.

CHAPTER 4. EXPERIMENTS AND RESULTS

In this chapter, we describe more precisely, how our algorithms work, state key points of the implementation. Also, we provide results of the experiments, which show how the height of the element in the splay-list depends on its number of accesses. Moreover, we compare skip-list and CBTree with the splay-list with different parameters on several workloads.

4.1. The Splay-List Implementation

In this section, we introduce the implementation for `contains` operation. `Insert` and `delete` (that simply marks) operations are performed similarly. The rebuild is a little bit complicated since we have to freeze whole data structure, however, since we talk about lock-based implementations it can be simply done by providing the global lock on the data structure.

Listing 1 – The data structure class definitions.

```
class Node:
  K key
  V value
  int zeroLevel
  int topLevel
  Lock lock
  int selfhits
  Node next[MAX_LEVEL]
  int hits[MAX_LEVEL]
  bool deleted

class SplayList:
  int m
  int M
  int zeroLevel
  Node head
  Node tail

SplayList list
double p
```

The main class that is used in our implementation is `Node` (Listing 1). It contains nine fields:

a) key field stores the corresponding key,
b) $value$ field stores the value for the corresponding key,
c) $zeroLevel$ field indicates the lowest sub-list to which the object belongs (for lazy expansion),

d) $topLevel$ field indicates the topmost sub-list to which the object belongs,
e) $lock$ field allows to lock the object,
f) $selfhits$ field stores the total number of hit-operations performed to key, i.e., sh_{key},
g) $next[h]$ is the succesor of the object in the sub-list of height h,
h) $hits[h]$ equals to $hits^h_{key}$ or, in other words, $hits(C^h_{key}) - selfhits$, and, finally,
i) $deleted$ mark that indicates whether the key is logically deleted.

The splay-list itself is represented by class `SplayList` with five fields:

a) m field stores the total number of performed hit-operations to all the keys,
b) M field stores the total number of hit-operations to non-marked objects,
c) $zeroLevel$ indicates the current lowest level of splay-list(for lazy expansion),
d) $head$ and $tail$ are sentinel nodes with $-\infty$ and $+\infty$ keys, correspondingly.

Moreover, the algorithm has a parameter p which is the probability how often we should perform the rebalancing.

Listing 2 – Contains function

```
fun contains(K key):
  Node node := find(key)
  if node = null:
    return false
  if random() < p:
    rebalance(key)
  return not node.deleted
```

The `contains` function is depicted at Listing 2. If `find` does not find an object with the corresponding key we return `false`. Otherwise, we rebalance, i.e., call function `rebalance`, with the probability p.

The `find` method which checks the existence of the key is almost identical to the standard `find` function in skip-lists. It is presented on the following Listing 3.

Note, that we use the lazy expansion: when we pass an object we check (Listing 3) whether it should belong to lower levels, i.e., the expansion should be performed, and if it should be we update the node. For the lazy expansion functions we refer to the Listing 4. In function `updateZeroLevel` we check whether the current level of a node is higher than the level of the list. If this happens we link the node on the lower level, decrement its `zeroLevel` field, and set the `next` field as the `next` on a higher level. In function `updateUpToLevel` we link the node to the levels until the argument `level`.

Listing 3 – Find function

```
fun find(K key):
  pred := list.head
  succ := head.next[MAX_LEVEL]
  for level := MAX_LEVEL−1 .. zeroLevel:
    updateUpToLevel(pred, level)
    succ := pred.next[level]
    if succ = null:
      continue
    updateUpToLevel(succ, level)
    while succ.key < key:
      pred := succ
      succ := pred.next[level]
      if succ = null:
        break
      updateUpToLevel(succ, level)
    if succ != null and succ.key = key:
      return succ
  return null
```

Listing 4 – Lazy expansion functions

```
// this function is called only when node.lock is taken
fun updateZeroLevel(Node node):
  if node.zeroLevel > list.zeroLevel:
    node.hits[node.zeroLevel − 1] := 0
    node.next[node.zeroLevel − 1] :=
        node.next[node.zeroLevel]
    node.zeroLevel−−
  return

fun updateUpToLevel(Node node, int level):
  node.lock.lock()
  while node.zeroLevel > level:
    updateZeroLevel(node)
  node.lock.unlock()
  return
```

The method `rebalance` that performs the rebalancing in forward pass is presented on Listing A.1.

At first, we introduce the function `getHits` that returns the total number of hits at the level h to the "subtree" of node $node$: this corresponds to $hits(C^h_{node}) = sh_{node} + hits^h_{node}$, where sh_{node} is `node.selfhits` and $hits^h_{node}$ is `node.hits[h]`.

Now, we describe how `rebalance` function works. It takes a key key as an argument, increments counters on the way down to the node with the corresponding key, and checks ascent and descent conditions. It starts by taking the lock on the head, by incrementing the total number of hit-operations on the list, `list.m`, and by incrementing the total number of hits in the "subtree" of the head.

Then, it starts the standard traversal procedure for the skip-list operation: we iterate over levels in order to find a node with the requested key. For that, we maintain three pointers `curr`, `pred` and `predpred` — the current node in the traversal as in skip-list, the previous node that has been passed (typically, `pred.next` is equal to `curr`), and the last traversed node on the previous level on which we already have a lock.

At each iteration h of the loop, the function first takes `pred` from the level $h + 1$ which now becomes `predpred` and its next neighbour `curr` on level h. When we touch a node we lazily update it (`pred` and `curr`) so that their lowest level becomes at most h. If the key of `curr` is already bigger than the argument key the function just increments the number of hit-operations in "subtree" of `pred` and continues with the lower level. Otherwise, it traverses the list at level h and stops when the key of `curr` becomes bigger than the argument key.

When touching a new node `curr` the function always lazily updates its height. If we find that the next node at the level has a key bigger than key, it locks `curr`, checks whether the next node is still has a bigger key, and updates either the `selfhits` field if the key in `curr` is equal to key or updates the number of hits at the "subtree". Also, note that if the function finds a node with the requested key, it has to change the counter M of our splay-list if the node is not marked.

After that the function checks ascent and descent conditions. We start with the ascent condition. If the number of hit-operations to $\bigcup C_x^h$ for $x \in S_{curr}$ which is calculated as `predpred.hits[h + 1] - predpred.hits[h]` is big enough, the function should promote the node `curr` to the higher levels and should recalculate `hits` field for `predpred` and `curr`. Please note that in this case `curr` is always the neighbour of `predpred`.

Then, the function checks the descent condition: whether the total number of hit-operations at "subtrees" of `pred` and `curr` at the current level is small enough. If the condition is satisfied the function locks `pred` and `curr`. If after the lock the condition is not satisfied (this can happen if someone inserted a new node, the

function unlocks all the nodes and continue with the next neighbour of `pred` on level h. If the descent condition is satisfied the function should demote `curr` node from level h to lower level $h - 1$. At first, it checks the current lowest level of our splay-list and decreases it lazily if necessary, also after that it tries to lazily increase the height of `curr` and `pred`. Finally, the function demotes `curr`, updates the counters and `next` field of `pred`, unlocks `curr` and `pred`, and continues with the next neighbour of `pred`.

If none of the conditions are satisfied we simply continue with the next node at the level.

4.2. The Splay-List Experiments

We aim to determine whether:

a) the splay-list can improve over the throughput of the baseline skip-list by successfully leveraging the skewed access distribution;
b) it scales, and what is the impact of update rates and number of threads;
c) it can be competitive with the CBTree data structure in sequential and concurrent scenarios;
d) we get the further improvement in the throughput during the long run under the same workload.

We split our experimental evaluation into two parts: on read-only workloads where there are only `contains` operations and general workloads with moderate amount of update (`insert` and `delete`) calls.

4.2.1. Read-Only Workloads

In our read-only experiments, we describe a family of **workloads** by $n{-}x{-}y$, which should be read as: given n keys, x% of the `contains` are performed on y% of the keys while other `contains` operations are on the rest of the keys. More precisely, we first populate the splay-list with n keys and randomly choose a set of "popular" keys S of size $y \times n$. We then start T threads, each of which iteratively picks an element and performs the `contains` operation, for 10 seconds. With probability x we choose a random element from S, otherwise, we choose an element outside of S uniformly at random.

For our experiments, we choose the following five workloads: uniform (or $10^5 - 100 - 100$), $10^5 - 90 - 10$, $10^5 - 95 - 5$, $10^5 - 99 - 1$, and Zipf distribution with the parameter 1. That is, 100%, 90%, 95%, and 99% of the operations go into

100%, 10%, 5%, and 1% of the keys, respectively. Further, we vary the *balancing rate/probability*, which we denote by p: this is the probability that a given operation will update hit counters and perform rebalancing.

Table 1 – Operations per second and average length of a path on uniform workload.

uniform	Skip-list	SL $p=1$	SL $p=\frac{1}{2}$	SL $p=\frac{1}{5}$
ops/sec	2747000.0	0.35x	0.46x	0.57x
length	30.86	25.53	25.54	25.55
		CBTree $p=1$	CBTree $p=\frac{1}{2}$	CBTree $p=\frac{1}{5}$
ops/secs		0.64x	0.74x	0.84x
length		9.84	9.84	9.84
		SL $p=\frac{1}{10}$	SL $p=\frac{1}{100}$	SL $p=\frac{1}{1000}$
ops/secs		0.63x	0.69x	0.70x
length		25.56	25.59	25.65
		CBTree $p=\frac{1}{10}$	CBTree $p=\frac{1}{100}$	CBTree $p=\frac{1}{1000}$
ops/secs		0.89x	0.94x	0.95x
length		9.84	9.88	9.93

Table 2 – Operations per second and average length of a path on $10^5-90-10$ workload.

$10^5-90-10$	Skip-list	SL $p=1$	SL $p=\frac{1}{2}$	SL $p=\frac{1}{5}$
ops/sec	2874600.0	0.60x	0.78x	1.00x
length	30.81	23.06	23.07	23.08
		CBTree $p=1$	CBTree $p=\frac{1}{2}$	CBTree $p=\frac{1}{5}$
ops/secs		1.15x	1.36x	1.59x
length		9.13	9.14	9.15
		SL $p=\frac{1}{10}$	SL $p=\frac{1}{100}$	SL $p=\frac{1}{1000}$
ops/secs		1.10x	1.12x	1.02x
length		23.13	23.75	25.06
		CBTree $p=\frac{1}{10}$	CBTree $p=\frac{1}{100}$	CBTree $p=\frac{1}{1000}$
ops/secs		1.71x	1.71x	1.52x
length		9.17	9.37	9.81

In the first round of experiments, we compare how the **single-threaded** splay-list performs under the chosen workloads. We execute it with different settings of p, the probability of adjustment, taking values 1, $\frac{1}{2}$, $\frac{1}{5}$, $\frac{1}{10}$, $\frac{1}{100}$ and $\frac{1}{1000}$. We compare against the sequential skip-list and CBTree. We measure two values: the number of operations per second and the average length of the path traversed. The results are

Table 3 – Operations per second and average length of a path on $10^5 - 95 - 5$ workload.

$10^5 - 95 - 5$	Skip-list	SL $p = 1$	SL $p = \frac{1}{2}$	SL $p = \frac{1}{5}$
ops/sec	2844520.0	0.69x	0.93x	1.21x
length	30.84	21.62	21.63	21.65
		CBTree $p = 1$	CBTree $p = \frac{1}{2}$	CBTree $p = \frac{1}{5}$
ops/secs		1.33x	1.61x	1.90x
length		8.61	8.61	8.62
		SL $p = \frac{1}{10}$	SL $p = \frac{1}{100}$	SL $p = \frac{1}{1000}$
ops/secs		1.34x	1.39x	1.17x
length		21.70	22.33	24.46
		CBTree $p = \frac{1}{10}$	CBTree $p = \frac{1}{100}$	CBTree $p = \frac{1}{1000}$
ops/secs		2.04x	2.09x	1.79x
length		8.65	8.90	9.58

Table 4 – Operations per second and average length of a path on $10^5 - 99 - 1$ workload.

$10^5 - 99 - 1$	Skip-list	SL $p = 1$	SL $p = \frac{1}{2}$	SL $p = \frac{1}{5}$
ops/sec	3559320.0	0.85x	1.19x	1.65x
length	31.00	17.13	17.16	17.23
		CBTree $p = 1$	CBTree $p = \frac{1}{2}$	CBTree $p = \frac{1}{5}$
ops/secs		1.37x	1.72x	2.06x
length		7.25	7.23	7.26
		SL $p = \frac{1}{10}$	SL $p = \frac{1}{100}$	SL $p = \frac{1}{1000}$
ops/secs		1.89x	2.01x	1.64x
length		17.30	18.59	21.00
		CBTree $p = \frac{1}{10}$	CBTree $p = \frac{1}{100}$	CBTree $p = \frac{1}{1000}$
ops/secs		2.25x	2.36x	2.04x
length		7.28	7.52	8.53

presented in Tables 1-5 (splay-list is abbreviated SL). For readability, throughput results are presented relative to the skip-list baseline.

Relative to the skip-list, the first observation is that, for high update rates (1 through 1/5), the splay-list predictably only matches or even loses performance. However, this trend improves as we reduce the update rate, and, more significantly, as we increase the access rate imbalance: for $99 - 1$, the sequential splay-list obtains a throughput improvement of $2\times$. This improvement directly correlates with the length of the access path (see third row). At the same time, notice the negative impact of very low update rates (last column), as the average path length increases,

Table 5 – Operations per second and average length of a path on Zipf 1 workload.

Zipf 1	Skip-list	SL $p = 1$	SL $p = \frac{1}{2}$	SL $p = \frac{1}{5}$
ops/sec	2700290.0	0.91x	1.15x	1.40x
length	30.76	14.14	14.14	14.12
		CBTree $p = 1$	CBTree $p = \frac{1}{2}$	CBTree $p = \frac{1}{5}$
ops/secs		1.26x	1.47x	1.66x
length		5.80	5.80	5.80
		SL $p = \frac{1}{10}$	SL $p = \frac{1}{100}$	SL $p = \frac{1}{1000}$
ops/secs		1.51x	1.63x	1.64x
length		14.11	14.06	14.12
		CBTree $p = \frac{1}{10}$	CBTree $p = \frac{1}{100}$	CBTree $p = \frac{1}{1000}$
ops/secs		1.75x	1.85x	1.86x
length		5.80	5.81	5.81

which leads to higher average latency and decreased throughput. We empirically found the best update rate to be around $1/100$, trading off latency with per-operation cost.

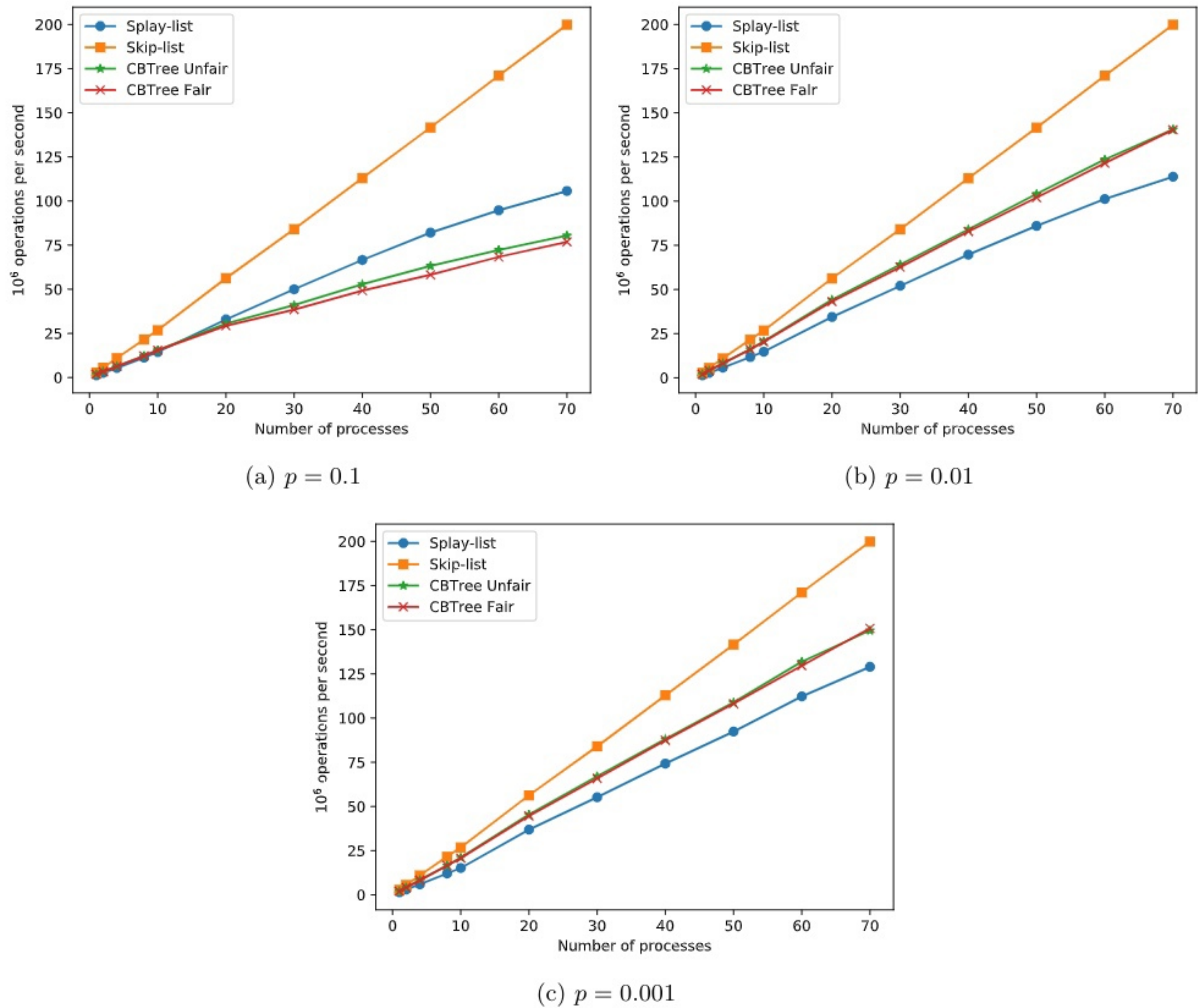
Relative to the sequential CBTree, we notice that the splay-list generally yields lower throughput. This is due to two factors:

a) the CBTree is able to yield shorter access paths, due to its structure and constants;
b) the tree tends to have better cache behavior relative to the skip-list backbone, i.e., more nodes can be stored in cache.

Given the large difference in terms of average path length, it may seem surprising that the splay-list is able to provide close performance. This is because of the caching mechanism: as long as the path length for popular elements is short enough so that all the topmost nodes are mostly in cache, the average path length is not critical. We will revisit this observation in the concurrent case.

Next, we analyze concurrent performance. Unfortunately, the original implementation of the CBTree is not available, and we therefore re-implemented it in our framework. Here, we make an important distinction relative to usage: the authors of the CBTree paper propose to use a single thread to perform all the rebalancing. However, this approach is not standard, as in practice, updates could come from different threads. Therefore, we implement two versions of the CBTree, one in which updates are performed by a single thread (CBTree-Unfair), and one in which updates can be performed by every thread with some probability (CBTree-Fair). In

both cases, synchronization between readers and writers is performed via an efficient readers-writers lock [4], which prevents concurrent updates to the tree. We note that in theory we could further optimize the CBTree to allow fully-concurrent updates via fine-grained synchronization. However, this would require a significant re-working of their algorithm and, as we will see below, this would not change results significantly.

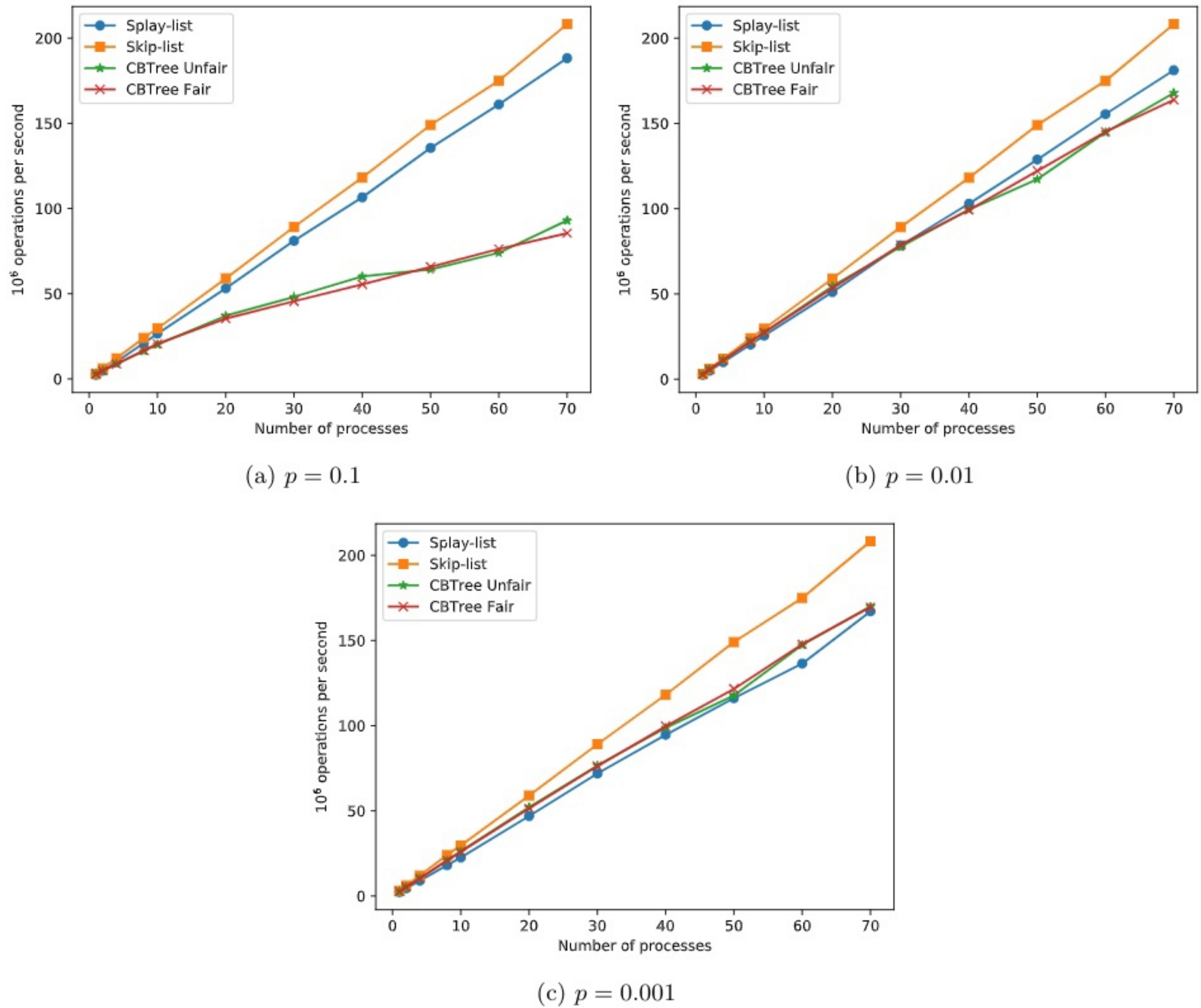


(a) $p = 0.1$ (b) $p = 0.01$ (c) $p = 0.001$

Figure 5 – Concurrent throughput for uniform workload

Our experiments, presented in Figures 5—9, analyze the performance of the splay-list relative to standard skip-list and the CBTree across different workloads (one per figure), different update rates (one per panel), and thread counts (X axis).

Examining the figures, at first, notice the relatively good scalability of the splay-list under all chosen update rates and workloads. By contrast, the CBTree scales well for moderately skewed workloads and low update rates, but performance decays for skewed workloads and high update rates (see for instance Figure 8 for $p = \frac{1}{10}$). We note that, in the former case the CBTree matches the performance of



(a) $p = 0.1$

(b) $p = 0.01$

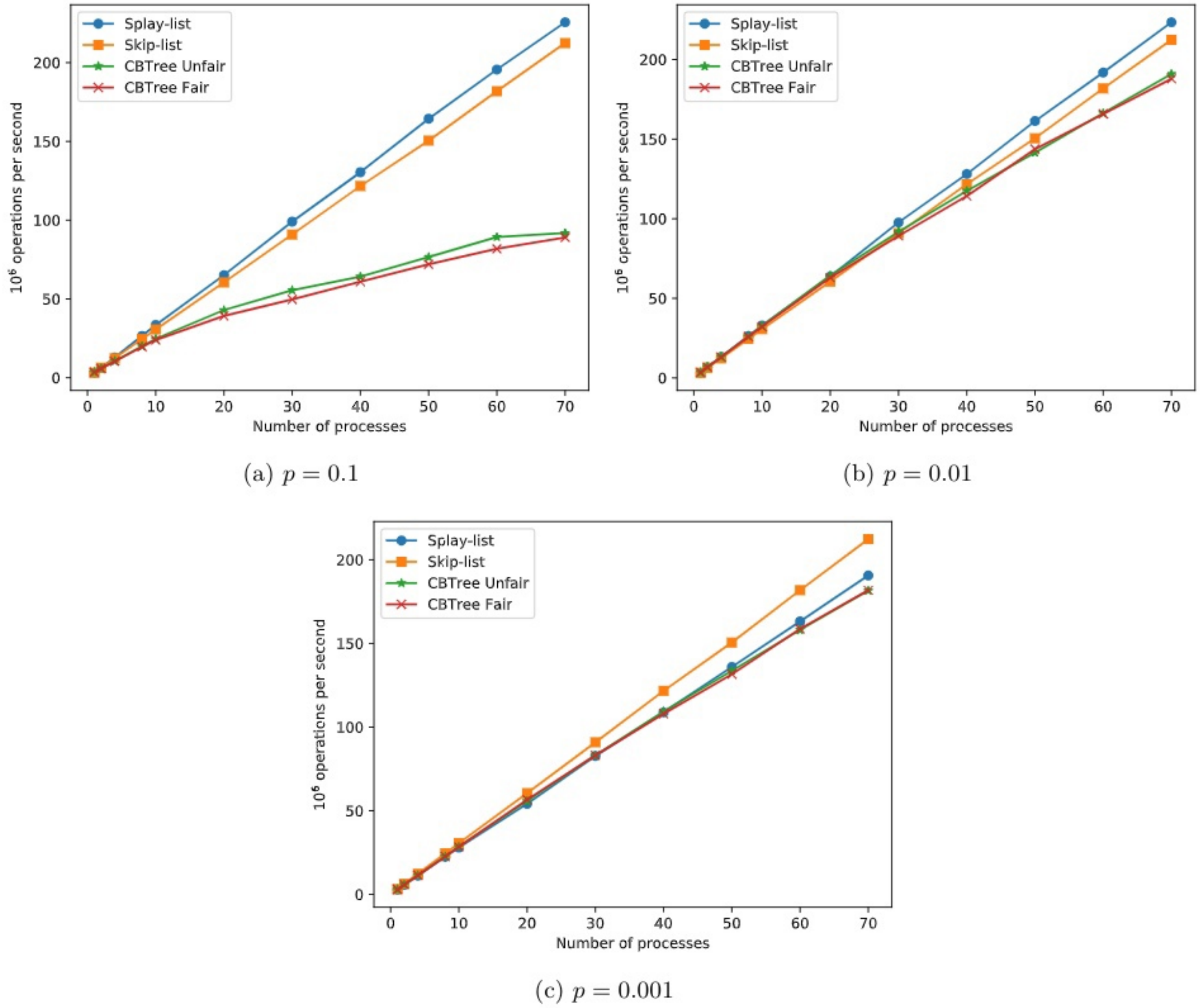
(c) $p = 0.001$

Figure 6 – Concurrent throughput for $10^5 - 90 - 10$ workload

the splay-list in the low-update case (see Figure 6 for $p = \frac{1}{1000}$), but its performance can decrease significantly if the update rates are reasonably high ($p = \frac{1}{100}$). We further note the limited impact of whether we consider the fair or unfair variant of the CBTree (although the Unfair variant usually performs better).

We consider a uniform workload $10^5 - 100 - 100$, i.e., the arguments of `contains` operations are chosen uniformly at random (Figure 5). As expected we lose performance lose relative to the skip-list due to the additional work our data structure performs. Note also that the CBTree outperforms the Splay-List in this setting. This is also to be expected, since the access cost, i.e., the number of links to traverse, is less for the CBTree.

We also ran the data structures on an input coming from a Zipf distribution with the skew parameter set to 1, which is the standard value: for instance, the frequency of words in the English language satisfies this parameter. As one can see on Figure 9, our splay-list outperforms or matches all other data structures.



(a) $p = 0.1$

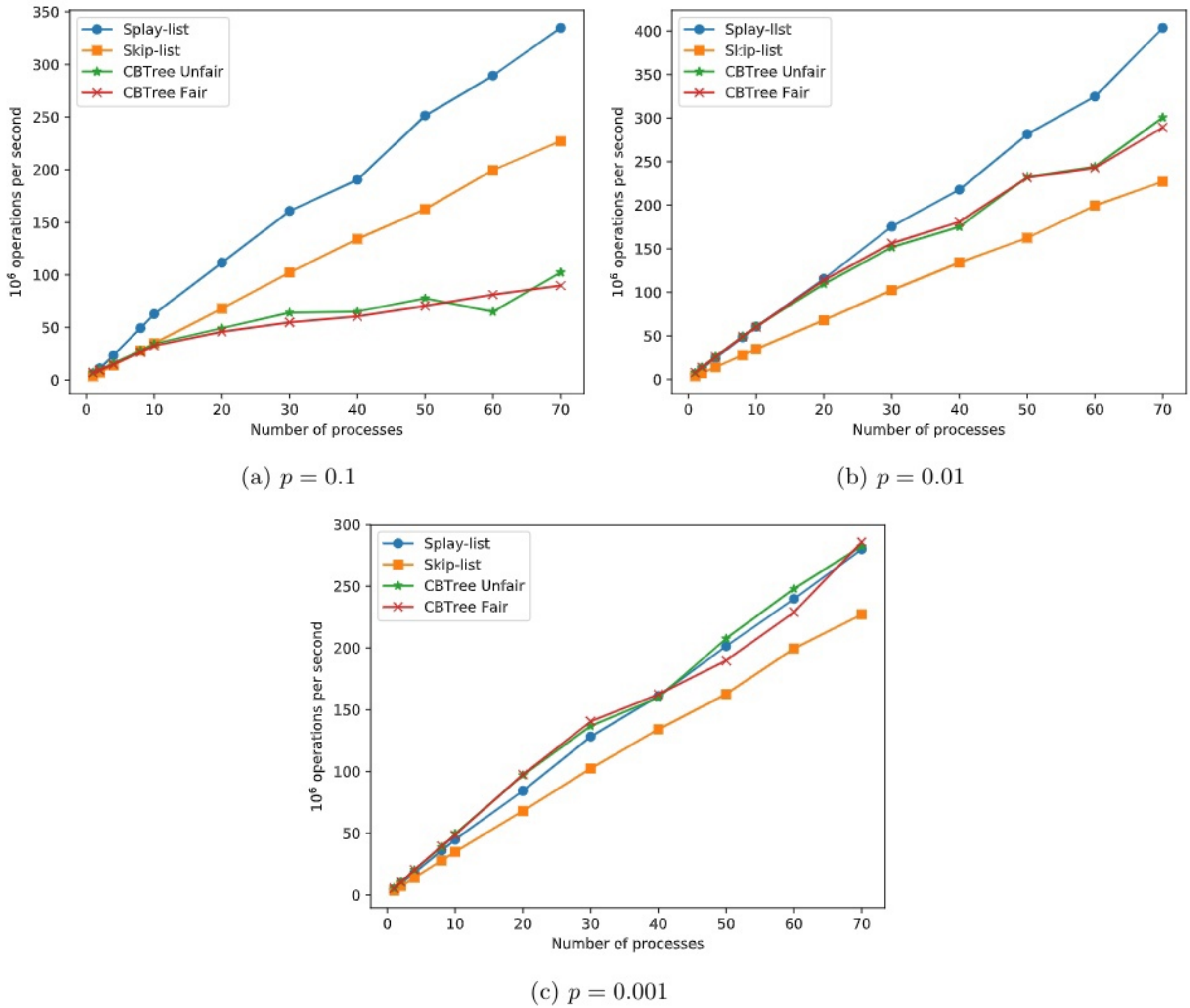
(b) $p = 0.01$

(c) $p = 0.001$

Figure 7 – Concurrent throughput for $10^5 - 95 - 5$ workload

These results may appear surprising given that the splay-list generally has longer access paths. However, it benefits significantly from the fact that it allows additional concurrency, and that the caching mechanism serves to hide some of its additional access cost. Our intuition here is that one critical measure is which fraction of the "popular" part of the data structure fits into the cache. This suggests that the splay-list can be practically competitive relative to the CBTree on a subset of workloads.

We run the splay-list with the best parameter $p = \frac{1}{100}$ for ten minutes on one process on the following distributions: $10^5 - 90 - 10$, $10^5 - 95 - 5$, $10^5 - 99 - 1$, and Zipf with the parameter 1. Then, we compare the measured throughput per second with the throughput per second on runs of ten seconds. Obviously, we expect that the throughput increases since the data structure learns more and more about the distribution after each operation. And it indeed happens as we can see on Table 6. In the long run, the improvement is up to 30%.



(a) $p = 0.1$

(b) $p = 0.01$

(c) $p = 0.001$

Figure 8 – Concurrent throughput for $10^5 - 99 - 1$ workload

4.2.2. General workloads

In addition to read-only workloads we implemented general workloads, allowing for inserts and deletes, in our framework. General workloads are specified by five parameters $n - r - x - y - s$:

a) n, the size of the workset of keys;
b) r%, the amount of `contains` performed;
c) x% of `contains` are performed on y% of keys;
d) `insert` and `delete` choose a key uniformly at random from s% of keys.

Talking in more details, we choose n keys as set S and we pre-populate the splay-list: we add a key from S with probability 50%. Then, we choose $s \times n$ keys uniformly at random to get a key set named W. Also, we choose $y \times n$ keys from *inserted* keys to get a key set named R. We start T threads, each of which chooses an operation: with probability r% it chooses `contains` and with probabilities $\frac{100-r}{2}$% it chooses `insert` or `delete`. Now, the thread has to choose an argument of the

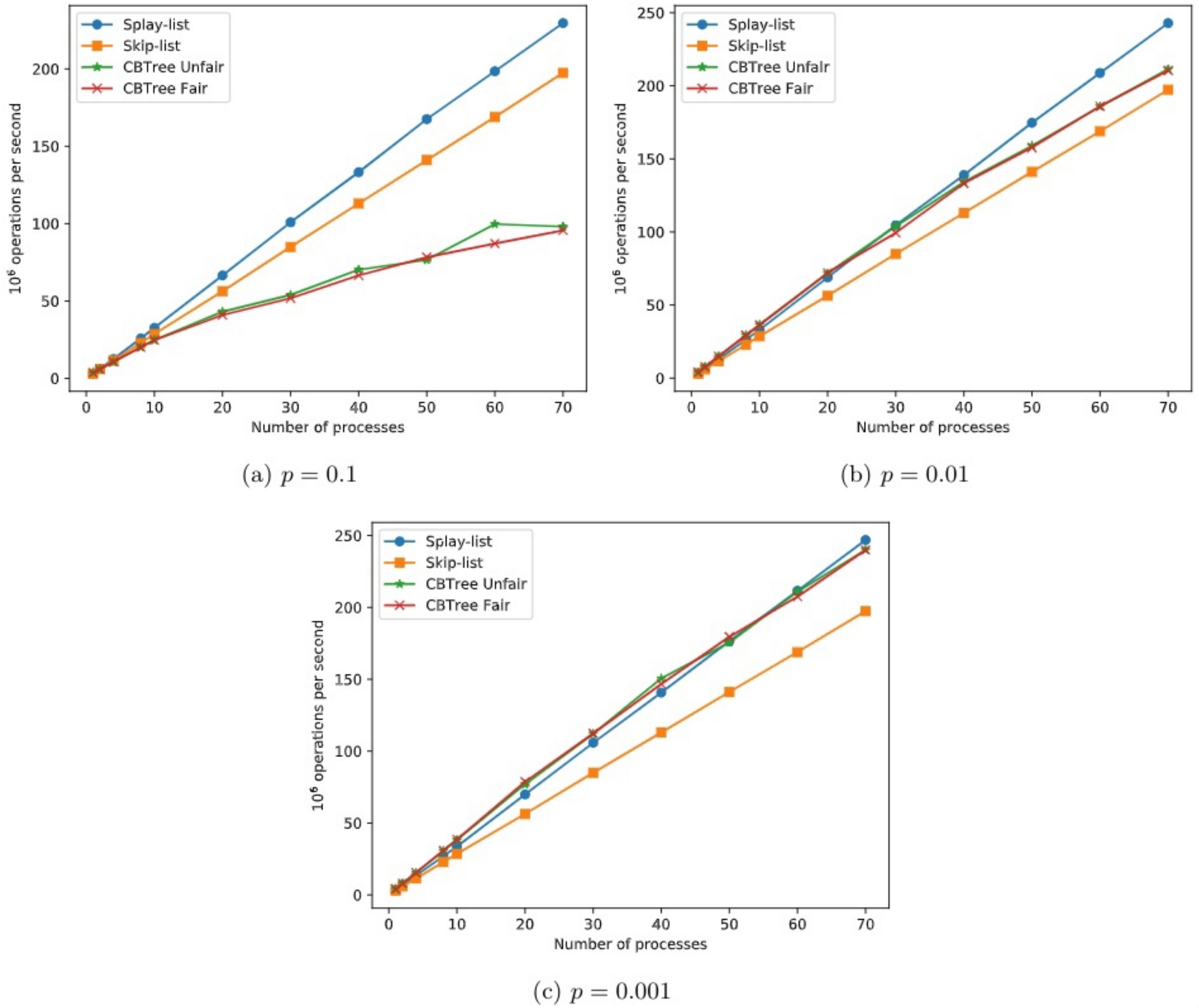


Figure 9 – Concurrent throughput on Zipf 1 workload

operation: for `contains` operation it chooses an argument from R with probability $x\%$, otherwise, it chooses an argument from $S \setminus R$; for `insert` and `delete` operations it chooses an argument from W uniformly at random.

We did not perform a full comparison with all other data structures (skip-list and the CBTree). However, we did a comparison to the splay-list itself on the following two types of workloads: read-write workloads, $10^5 - 98 - 90 - 10 - 25$, $10^5 - 98 - 95 - 5 - 25$ and $10^5 - 98 - 99 - 1 - 25$ — choosing `contains` operation with probability 98%, and `insert` and `delete` operations takes one quarter of elements as arguments; and read-only workloads, $10^5 - 0 - 90 - 10 - 0$, $10^5 - 0 - 95 - 5 - 0$, and $10^5 - 0 - 99 - 1 - 0$ which relate to the read-only workloads $10^5 - 90 - 10$, $10^5 - 95 - 5$, and $10^5 - 99 - 1$ described above.

The intuition is that the splay-list should perform better on the second type of workloads, but by how much? We answer this question: the overhead does not

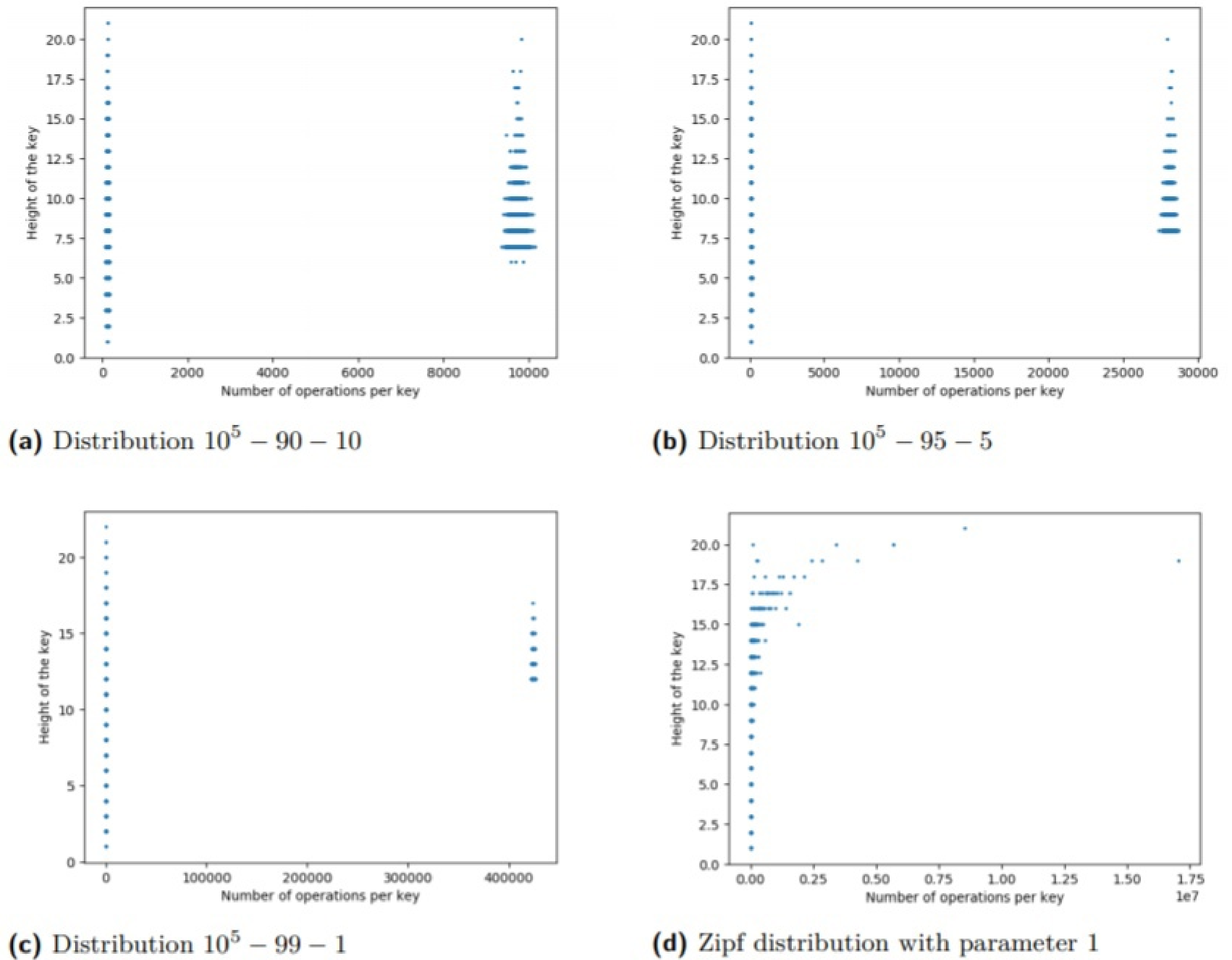
Table 6 – Comparison of the throughput on runs for 10 seconds and 10 minutes

Distribution	10 sec	10 min
$10^5 - 90 - 10$	2777150	3630640 (+30%)
$10^5 - 95 - 5$	3401220	4403906 (+29%)
$10^5 - 99 - 1$	6707690	8184215 (+22%)
Zipf 1	3806500	4261981 (+12%)

exceed 15% on $99-1$-workloads, does not exceed 7% on $95-5$-workloads, and does not exceed 5% on $90-1$-workloads. As expected, the less a workload is skewed, the less the overhead. By that, we obtain that the small amount of `insert` and `delete` operations does not affect the performance significantly.

4.2.3. The Correlation between Key Popularity and Height

We run the splay-list with the best parameter $p = \frac{1}{100}$ for 100 seconds on one process on the following distributions: $10^5 - 90 - 10$, $10^5 - 95 - 5$, $10^5 - 99 - 1$ and Zipf with parameter 1. Then, we build the plots (see Figures 10) where for each key we draw a point (x, y) where x is the number of operations per key and y is the height of the key. We would expect that the larger the number of operations, the higher the nodes will be. This is obviously the case under Zipf distribution. With other distributions the correlation is not immediately obvious, however, one can see that if the number of operations per key is high, then the lowest height of the key is much higher than 1.



(a) Distribution $10^5 - 90 - 10$

(b) Distribution $10^5 - 95 - 5$

(c) Distribution $10^5 - 99 - 1$

(d) Zipf distribution with parameter 1

Figure 10 – The correlation between the popularity and the height

Conclusions on Chapter 4

In this chapter we discussed important details of the implementation. Also, we provided results of the experiments, which indicate, that the sequential and concurrent implementations of splay-list show overall better results on the general and read-only highly skewed workloads, than the skip-list and the CBTree. Thirdly, we showed, that performance of the splay-list becomes better if it runs for longer time period.

Fourthly, we investigated the correlation between height of the elements in the splay-list and number of accesses made to them, and our experiments showed, that for more popular elements minimal height is bigger, and they tend to be presented on the higher levels, than less popular ones.

CONCLUSION

In this work, we revisited the question of efficient self-adjusting concurrent data structures, we presented the first self-adjusting concurrent instances of skip-list and IST.

The splay-list design ensures static optimality, and has an arguably simple structure and implementation, which allows for additional concurrency and good performance under skewed access. In addition, this design provides guarantees under approximate access counts, required for good practical behavior. Also, we empirically showed, that the splay-list shows better performance, than the skip-list and the CBTree for highly skewed distributions of accesses.

For the self-adjusting IST design we proved theoretical bounds that show for the big class of access distributions it has better expected amortized time complexity, than the splay-list and the CBTree, and can be made lock-free.

REFERENCES

1 A simple optimistic skiplist algorithm / M. Herlihy [et al.] // Proceedings of the 14th international conference on Structural information and communication complexity. — Castiglioncello, LI, Italy : Springer-Verlag, 2007. — P. 124–138. — (SIROCCO'07).

2 Benchmarking cloud serving systems with YCSB / B. F. Cooper [et al.] // Proceedings of the 1st ACM symposium on Cloud computing. — 2010. — P. 143–154.

3 CBTree: A Practical Concurrent Self-adjusting Search Tree / Y. Afek [et al.] // Proceedings of the 26th International Conference on Distributed Computing. — Salvador, Brazil : Springer-Verlag, 2012. — P. 1–15. — (DISC'12). — URL: `http://dx.doi.org/10.1007/978-3-642-33651-5_1`.

4 *Correia A.*, *Ramalhete P.* Scalable Reader-Writer Lock in C++1x [Electronic resource]. — 2015. — URL: `http://concurrencyfreaks.blogspot.com/2015/01/scalable-reader-writer-lock-in-c1x.html`.

5 *Herlihy M.*, *Shavit N.* The Art of Multiprocessor Programming. — San Francisco, CA, USA : Morgan Kaufmann Publishers Inc., 2008. — 536 p.

6 *Knuth D. E.* The art of computer programming. Vol. 3. — Pearson Education, 1997.

7 *Lea D.* Concurrent Skip-List Map [Electronic resource]. — 2007. — URL: `http://java.sun.com/javase/6/docs/api/java/util/concurrent/ConcurrentSkipListMap.html`.

8 *Mehlhorn K.*, *Tsakalidis A.* Dynamic interpolation search. — 1991. — URL: `https://people.mpi-inf.mpg.de/~mehlhorn/ftp/DynamicInterpolationSearch.pdf`.

9 *Michael M. M.* High performance dynamic lock-free hash tables and list-based sets // Proceedings of the fourteenth annual ACM symposium on Parallel algorithms and architectures. — 2002. — P. 73–82.

10 *Natarajan A.*, *Mittal N.* Fast Concurrent Lock-free Binary Search Trees // Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. — Orlando, Florida, USA : ACM, 2014. — P. 317–328. — (PPoPP '14). — URL: http://doi.acm.org/10.1145/2555243.2555256.

11 Non-blocking Binary Search Trees / F. Ellen [et al.] // Proceedings of the 29th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing. — Zurich, Switzerland : ACM, 2010. — P. 131–140. — (PODC '10). — URL: http://doi.acm.org/10.1145/1835698.1835736.

12 *Poess M.*, *Floyd C.* New TPC benchmarks for decision support and web commerce // ACM Sigmod Record. — 2000. — Vol. 29, no. 4. — P. 64–71.

13 Prefix sum [Electronic resource]. — 2021. — URL: https://en.wikipedia.org/wiki/Prefix_sum.

14 *Prokopec A.*, *Brown T.*, *Alistarh D.* Analysis and Evaluation of Non-Blocking Interpolation Search Trees // Proceedings of Principles and Practice of Parallel Programming 2020. — 2020. — (PPoPP'20).

15 *Pugh W.* A Probabilistic Alternative to Balanced Trees. — 1989.

16 *Pugh W.* Concurrent maintenance of skip lists. — 1998.

17 *Tarjan R.*, *Sleator D.* Self-Adjusting Binary Search Trees [Electronic resource]. — 1985. — URL: https://www.cs.princeton.edu/courses/archive/spring04/cos423/handouts/splay%5C%20trees.pdf.

APPENDIX A. PSEUDO CODE OF REBALANCING OPERATION

Listing A.1 – Pseudocode of the rebalance function.

```
fun getHits(Node node, int h):
  if node.zeroLevel > h:
    return node.selfhits
  return node.selfhits + node.hits[h]

fun rebalance(K key):
  list.head.lock()
  list.m++
  currM := list.m
  list.head.hits[MAX_LEVEL]++
  Node pred := list.head
  for h := MAX_LEVEL−1 .. zeroLevel:
    while pred.zeroLevel > h:
      updateZeroLevel(pred)
    predpred := pred
    curr := pred.next[h]
    updateUpToLevel(curr, h)
    if curr.key > key:
      pred.hits[h]++
      continue

    found_key := false
    while curr.key ⩽ key:
      updateUpToLevel(curr, h)
      acquired := false
      if curr.next[h].key > key:
        curr.lock.lock()
        if curr.next[h].key ⩽ key:
          curr.lock.unlock()
        else:
          acquired := true
          if curr.key = key:
            curr.selfhits++
            found_key := true
            if !curr.deleted:
              fetch_and_add(list.M)
        else:
          curr.hits[h]++
```

```
// Ascent condition
if h + 1 < MAX_LEVEL and h < predpred.topLevel and
    predpred.hits[h + 1] − predpred.hits[h] >
      currM / 2^(MAX_LEVEL−1−h−1):
  if not acquired:
    curr.lock.lock()
  curh := curr.topLevel
  while curh + 1 < MAX_LEVEL and
      curh < predpred.topLevel and
      predpred.hits[curh + 1] −
        predpred.hits[curh] >
        currM / 2^(MAX_LEVEL−1−curh−1):
    curr.topLevel++
    curh++
    curr.hits[curh] := predpred.hits[curh] −
        predpred.hits[curh − 1] − curr.selfhits
    curr.next[curh] := predpred.next[curh]
    predpred.hits[curh] := predpred.hits[curh − 1]
    predpred.next[curh] := curr
  predpred := curr
  pred := curr
  curr := curr.next[h]
  continue
// Descent condition
elif curr.topLevel = h and
     curr.next[h].key ⩽ key and
     getHits(curr, h) + getHits(pred, h)
       ⩽ currM / 2^(MAX_LEVEL−1−h):
  currZeroLevel := list.zeroLevel
  if pred ≠ predpred:
    pred.lock.lock()
  curr.lock.lock()
  // Check the conditions that nothing has changed
  if curr.topLevel ≠ h or
      getHits(curr, h) + getHits(pred, h) >
        currM / 2^(MAX_LEVEL−1−h) or
      curr.next[h].key > key or
      pred.next[h] ≠ curr:
    if pred ≠ predpred:
      pred.lock.unlock()
    curr.lock.unlock()
```

```
          curr := pred.next[h]
          continue
        else:
          if h = currZeroLevel:
            CAS(list.zeroLevel, currZeroLevel,
              currZeroLevel - 1)
          if curr.zeroLevel > h - 1:
            updateZeroLevel(curr)
          if pred.zeroLevel > h - 1:
            updateZeroLevel(pred)
          pred.hits[h] := pred.hits[h] +
                                   getHits(curr, h)
          curr.hits[h] := 0
          pred.next[h] := curr.next[h]
          curr.next[h] := null
          if pred ≠ predpred:
            pred.lock.unlock()
          curr.topLevel--
          curr.lock.unlock()
          curr := pred.next[h]
          continue
      pred := curr
    if predpred ≠ pred:
      predpred.lock.unlock()
    if found_key:
      pred.lock.unlock()
      return
  pred.lock.unlock()
```