Searching with Dynamic Optimality: In Theory and Practice

Henrik Bitsch Kirk, 20030612
Master thesis

March 1, 2009

DEPARTMENT OF COMPUTER SCIENCE
FACULTY OF SCIENCE
AARHUS UNIVERSITY

Supervisor:
Gerth Stølting Brodal

Abstract

In this thesis we are presenting different techniques for analysing algorithms designed for skew access sequences along with properties, which categorises these algorithms. Furthermore are we describing some related data structures, which support the *working set*, *queueish*, emphdynamic finger and the *unified property*. We also describe how to implement data structures which supports these properties in practice.

One of the main results in this thesis is the description and implementation of the *dynamic self adjusting skip lists*, which is developed by Prosenjit Bose, Karim Douïeb and Stefan Langerman from their 2008 paper "Dynamic optimality for skip lists and *B*-trees". The implementation is accompanied with a detailed description of the data structure and its common operations *search*, *insert*, and *delete*. This implementation is tested against an implementation of *splay* trees developed by Daniel Slaetor and Robert E. Tarjan and presented in the paper from 1985 "Self-adjusting binary search trees". The *splay* trees are a commonly used data structure for skew access sequences due to its easy implementation. The disadvantage of this data structure is its potential bad *I/O* and cache performance. Minimising cache faults and keeping locality is difficult in this structure. The dynamic self adjusting skip lists has some advantages in preserving locality, as we will see in this thesis, but the structure is hard to implement and has a constant factor hidden in the *Big-O* notation. The results are therefore not surprisingly that dynamic self adjusting skip lists are performing worse than *splay* trees even though they are better at preserving locality. The results of the empirical tests in this thesis are therefore not surprising that *splay* trees performs equally good or better under the different tested scenarios.

To compensate for the growing number of elements in data collections we are in this thesis also presenting P. Bose et al. dynamic *I/O* effective *B*-trees, which they contribute in the same paper, to solve the adaptive dictionary problem in external memory. They have used Brian C. Dean and Zachary H. Jones' translation scheme from 2007 to translate dynamic self adjusting skip lists to dynamic *I/O* efficient *B*-trees. We are in details describing the translation scheme and the resulting dynamic *I/O* effecient *B*-trees data structure.

Resume

I denne afhandling præsenteres forskellige teknikker til analyse af algoritmer, der er designet til skæve tilgangs sekvenser, vi præsentere sammen med disse teknikker også egenskaber som kan hjælpe med at kategorisere disse algoritmer. Ydermere beskriver vi nogle relevante data strukturer, som supporterer *working set*, *queueish*, *dynamic finger* og *unified* egenskaberne. Vi beskriver ligeledes hvordan algoritmer der opfylder disse egenskaber kan implementeres i praksis.

Et af denne afhandlings hovedresultater er beskrivelse samt implementation af *dynamic self adjusting skip listerne*, der er udviklet af Prosenjit Bose, Karim Douïeb and Stefan Langerman, udgivet i deres 2008 afhandling "Dynamic optimality for skip lists and *B*-trees". Implementationen bliver suppleret med en detaljeret beskrivelse af data strukturen og dens almindelig brugte operationer *search*, *insert* og *delete*. *Splay* træet er publiceret af Daniel Slaetor og Robert E. Tarjan i deres afhandling fra 1985 "Self-adjusting binary search trees". *Splay* træet er en gængs data struktur, til skæve tilgangs sekvenser, da den er meget enkel at implementerer. En af ulemperne ved denne struktur er den potentielt meget slemmen *I/O* og *cache* præstationsevne. At minimerer antallet af cache fejl og opretholde lokalitet er meget svært, hvis ikke umuligt i denne struktur. Den selv justerende skip liste har derimod nogle fordele når der skal opretholdes lokalitet, som vil blive synliggjort i denne afhandling, modsat er strukturen meget svær at implementere. Der er desuden en store konstant factor gemt i *Big-O* notationen. Resultaterne af de udførte test er derfor ikke overraskende at *dynamic self adjusting skip listerne* kører ligeså hurtigt eller bedre end *splay* træerne under de sekvenser vi har testet de to data strukturer med.

For at kompenserer for det stigende antal elementer der er i data samlingerne, præsenteres der også dynamisk *I/O* effektiv *B*-træ struktur også udviklet i sammen 2008 afhandling af P. Bose et al., som kan klare det adaptive ordbogs problem i eksterne hukommelse. P. Bose et al. har brugt et oversættelse system af Brian C. Dean og Zachary H. Jones fra 2007, som oversætter mellem skip lister og *B*-træer. Detaljerne omkring denne ekstern dynamisk optimale *B*-træ algoritme vil også være at finde i denne afhandling.

Acknowledgements

First and foremost I want to thank my supervisor Gerth Stølting Brodal for helping me whenever I needed help on both technical and non technical matters or I were stuck. Thanks for his always needed constructive criticism when reviewing my thesis, his insights and suggestions when seeing a flows in my argumentation or missing pieces in my text.

Also a great thanks to Natasha Russo, Jacob Ulrich, and Claus Andersen for reviewing this thesis and helping correcting grammatical and logical faults.

On a more practical side, thanks to the staff on *DAIMI* for lending me a test machine and letting me tread it as my own for this half year.

This thesis was written in LaTeX. Figures made by *dia*, *ipe* and *gnuplot*. The source code was implemented in *C++* and compiled with *GCC* and *SCons*.

February 28, 2009

Contents

Introduction

The dictionary problem is a well known and well studied problem in computer science. The use of fast and reliable searching is reaching well beyond that of computer science and into everyday problems as well as specific software solutions like databases, web-search engines and many more.

Searching for an element[1] in a large collection is something everyone is doing everyday, when looking through phone books, searching the content or glossary of books, or searching our desktops for the right piece of paper. In computer science the problem was first considered interesting in the fifties, when the amount of computer memory started growing. The access to larger and larger random access memory (*RAM*) in the 1950's, gave computer scientist reasons to find a more efficient way of handling available memory.

The simplest heuristic for searching in a potential large collection of n elements for a key k, would be a sequential scan over all elements. This will find the element k if present in the collection. The time complexity of scanning all elements is clearly $O(n)$, in the *RAM*-model presented by von Neumann [TG98, pages 16-19]. If we are to search only once this is feasible, otherwise we should try and do better. We can initially sort the collection, then a faster method could be a binary search [Knu98, pages 409-417], here the idea is to jump forward or backwards half the number elements between the last jump point or the end. This idea is illustrated in Figure 1 and will reduce the searching time[2] from $O(n)$ to $(\lg n)$.

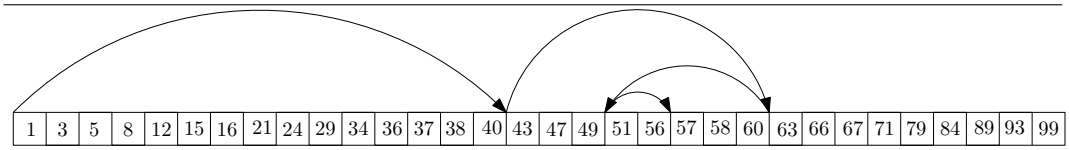


Figure 1: Searching for 57 in sorted collection using binary search

A more complex solution to "the searching problem" would be to build perfectly binary search trees, see Figure 2 on top of the n elements; this is essentially how

[1] In this thesis we are using items or elements interchangeably.
[2] Let $\lg x$ be the binary logarithm $\log_2(x+1)$ in this thesis.

binary search works. Searches would start at the root of the tree and work its way down, by comparing the value of k to the value of the node[3] x and then going left or right depending on whether k being smaller or larger than the value of x. The number of comparisons when searching in a perfectly balanced binary trees is at most $O(\lg n)$, which is the same bound as binary searching and equal to the height of the tree. Other binary search structures such as *red-black*-trees, *AVL*-trees, and B-trees can also be used as search structures with equal bounds, though the constants hidden by the *Big-O* are different, since they only have height $O(\lg n)$ instead of $O(\lfloor \log_2 n \rfloor + 1)$.

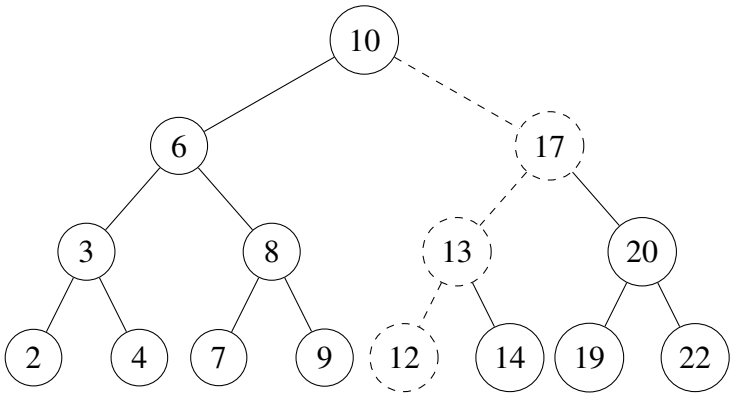


Figure 2: Searching for 12 in a binary tree.

Ideally access sequences of length m, like the one described in Sequences (1), (2), (3), should be handled faster than $O(m \lg(n))$ in search structures with n elements. In the offline case, where the access sequence is known in advance it is easy to build static search trees that supports these queries in constant $O(1)$ time per access, for each of these sequences. The problem is more difficult in a dynamic scenario, where the access sequence is not known in advance and where the search trees dynamically have to "learn" and "adapt" to the input sequences.

$$1, 2, 1, 2, 1, 2, \ldots, 1, 2, 1, 2 \tag{1}$$
$$1, 2, \ldots, n, 1, 2, \ldots, n, 1, 2, \ldots, n \tag{2}$$
$$1, n, 1, n, 1, n, \ldots, 1, n, 1, n\ . \tag{3}$$

Throughout the litterateur there has been different ways to solve skew access sequences, like the ones shown in Sequences (1), (2), (3). The techniques can be divided into static and dynamic search structures, which we have briefly mentioned above. We could also divide them into structures which is designed respectively for access frequencies and access sequences. For sequence (2) it is easy to argue

[3] In this thesis we define nodes residing in a tree as internal nodes, and leafs as nodes with no children.

that in static binary search structures can achieve better than $\Omega(n \lg n)$. An example of static data structures which are build according to access frequency, could be the *biased* search tree presented in [BST85] where each element is assigned a weight w_i according to the access frequency of w_i and the access cost for an element should be $O(\lg W/w_i)$, where $W = \sum_i w$. Likewise an example of dynamic data structures designed for handling different access sequences well are *splay* trees.

In 1985 D. Sleator and R. E. Tarjan proposed a class of search trees called *self-adjusting* search trees, specifically they introduces the *splay* tree data structure. The *splay* trees has inspired others to create *self-adjusting* data structures, each promising optimality for dynamic searches. A small set of these *self-adjusting* data structures are presented in Chapter 2. The *splay* trees are shown optimal within a constant factor of the static optimal offline search tree over a sequence of $m = a_1, a_2, a_3, \ldots, a_m$ accesses. *Splay* trees also support what is called the *working set property* (WS). The WS property states that for an element a with working set number $t_i(a)$, the working set bound is defined:

$$WS(\mathbb{X}) = O\left(\sum_{i=1}^{m} \lg(t_i(a))\right) ,$$

where $\mathbb{X}$ is the access sequence and the function t_i is the number of distinct accesses at time i since a was last accessed. To give an example on function t_i we define the following access sequence of keys from the set $\{1, 2, 3, 4, 5, 6, 7, 8, 9\}$:

$$1, 2, 4, 5, 6, 1, 5, 6, 7, 1, 6, 7, 8, 8, 9, 4, 2, 4, 4, 1, 1, 5, 6, 8, 1, 5, 6, 7, 4 \, . \qquad (4)$$

The working set number for key 4 is seen in Table 1.

i^{th} access	$t_i(4)$
3	-
16	6
18	1
19	0
29	5

Table 1: Example of values taken by $t_i(4)$ for access Sequence 4.

The working set bound gives an amortised access bound of each access bounded by $O(\lg(t_i(a)))$. In Chapter 3 we are going into details concerning *splay* trees. The details of the *dynamic self adjusting skip lists* are likewise presented in Chapter 5. This data structure was presented by Prosenjit Bose, Karim Douïeb and Stefan Langerman [BDL08]. It works under the same access bounds as the *splay* trees, at the same time laying ground for expanding the structure to dynamic external

memory B-trees. We compare theoretical equal bounds of the *splay* trees and the dynamic self adjusting skip lists with empirical tests in Chapter 6.

The tests conducted on *splay* trees and dynamic self adjusting skip lists show that even though dynamic self adjusting skip lists are better at preserving locality, *splay* trees performs better in most cases because of the *dynamic finger* property this structure also supports. Only when the access sequence are purely working set friendly dynamic self adjusting skip lists performs just as good as *splay* trees.

Handling increasing data

In this thesis we are working within two algorithmic models. The first is the *RAM* model, which is modelling how algorithms behaves in internal memory on contemporary computers[4].

The *RAM* model or von Neumann model, is a simplification of the RAM computer, where it is possible to reach a memory cell in constant $O(1)$ time, each memory cell is able to store an $O(\lg n)$ sized integer. This is a simplified model compared to modern computers, which have multiple levels of memory, and where the rule of thumb is; as memory grows in size, its access times are also getting slower.

"The searching problem" has in these last years presented itself as an even harder problem, with the increasing amounts of data in especially different branches of science, a new model was needed. The expressiveness of the *RAM* model is failing to model the scenario when memory requirements are exceeding the space available in the main memory. The *I/O*-model [AV88] was introduced to fill this vacuum in designing algorithms. The *I/O*-model was presented by Allok Aggervel and Jeffrey E. Vittel [AV88] in 1988. We are describing this model in Chapter 7.

In 1972 R. Bayer and E. McCreight presented the *B*-trees data structure in the paper [BM72], an example of a *B*-tree can be seen in Figure 8.1 on page 76. *B*-trees works very well on large collections of data. We are here presenting both the original *B*-trees and the external *B*-trees in Chapter 8. P. Bose et al. [BDL08] presented an extension of the *B*-trees data structure which is *I/O* efficient and satisfy the working set property in external memory. We are presenting the details concerning translation between skip lists and *B*-trees in Chapter 9 and *I/O* data structure in Chapter 10.

[4]This thesis is written in 2009.

1 Comparing algorithms

People like choices
Michael T. Goodrich & Roberto Tamassia, Algorithm Design (2002)

When analysing adaptive search algorithms and comparing their running time complexities, we normally compare a concrete algorithm with an optimal offline algorithm. The optimal offline algorithm knows the whole access sequence $\mathbb{X}$ in advance, hence it can make decision based on the whole of $\mathbb{X}$ and in particular the next access. When referring to an online algorithm in this thesis, we refer to an algorithm which does not know the access sequence in advance.

1.1 Analysing algorithms

In this section we are presenting some related work, which makes analysis of dynamic data structures, that depends on sequences of updates and searches possible. Different ways to compare these data structures, against optimal offline algorithms, is furthermore presented in this section

1.1.1 c-competitiveness

When analysing online algorithms D. Sleator and R. Tarjan [ST85a] proposed a competitive analysis, where the *deterministic* online algorithm A is compared to the optimal offline algorithm OPT. The optimal offline algorithm knows the entire request sequence $\sigma = \sigma(1), \sigma(2), \ldots, \sigma(m)$ in advance. This is in contrast to the online algorithm, which does not depend on knowing element $\sigma(t')$, where $t < t'$, when answering the query $\sigma(t)$, where $1 \leq t \leq m$.

Now we can define the cost $C_A(\sigma)$ of an online algorithm A over an access sequence σ. Equally we can define the cost of an optimal offline algorithm is $C_{OPT}(\sigma)$ over the same access sequence.

An online algorithm A is now called c-*competitive* if there exist constants c and a, such that the following inequality holds,

$$C_A(\sigma) \leq c \cdot C_{OPT}(\sigma) + a \ ,$$

for all request sequences σ. If the inequality holds then c is called the *competitive ratio*. It should be mentioned that finding an optimal offline algorithm is hard in some cases.

1.1.2 Potential functions

In algorithm design and analysis it is sometimes practical to analyse an algorithms average performance, which is called the *amortised cost*. In a dictionary example we therefore calculating the amortised cost over an sequence of accesses. The technique used for this type of analysis is *potential functions*.

For a fixed input sequence σ of length m, the *potential function* of algorithm A is donated by a positive real number Φ. Now let Φ_i be the potential of the algorithm after the i^{th} operation of the sequence σ by algorithm A. Let t_i donate the cost of the i^{th} operation by A on access σ_i. The difference in potential of A after the i^{th} step is defined by $\Delta\Phi_i = \Phi_i - \Phi_{i-1}$, the amortised cost a_i of A at the i^{th} step is then defined by $a_i = t_i + \Delta\Phi_i$.

The hard thing is then to find a suitable potential function for the specific algorithm.

For algorithm A, we can analyse A by using the found potential function as follows; Run both A and OPT simultaneously on the sequence σ and let o_i be the cost of the i^{th} operation of OPT, defined above. An upper bound on the algorithm A over the entire sequence σ, can be proved, by bounding the amortised cost of A for each operation. Consider a non-negative potential function, which initial value is equal to zero and for each i it should apply that $1 \leq i \leq m$ and $a_i \leq c \cdot o_i$. Then

$$\begin{aligned} C_A(\sigma) &= \sum_{i=1}^{m} t_i \\ &\leq \Phi_m - \Phi_0 + \sum_{i=1}^{m} t_i = \sum_{i=1}^{m} (t_i + \Delta\Phi_i) = \sum_{i=1}^{m} a_i \\ &\leq \sum_{i=1}^{m} o_i = c \cdot C_{OPT}(\sigma) \ . \end{aligned}$$

This way of using potential functions as a method to analyse an algorithms performance over a sequence of operations is called an *amortised analysis*. This method was first shown in the paper [Tar85a] by Robert E. Tarjan from 1985 . This simply states that an individual operation can be costly, however the average cost over a sequence can be bounded to an average cost.

A common technique presented in [Hoc97] to prove upper bounds for competitive ratio, is to construct a potential function.

1.2 Algorithmic properties

Now we have a way to analyse our algorithms, but we need a way to compare different algorithmic properties. The online algorithms presented throughout this thesis, are compared by the different properties presented in this section.

1.2.1 Working set property

The working set property (WS) states that for an element x with working set number $t_i(x)$, the working set bound is defined:

$$WS(\mathbb{X}) = O\left(\sum_{i=1}^{m} \lg t_i(x)\right) ,$$

where $\mathbb{X}$ is the access sequence and the function $t_i(x)$ is the number of distinct accesses at time i since x was last accessed. This property was presented by Daniel Sleator and Robert E. Tarjan [ST85b, Theorem 4]. It states that over an access sequence of size m, which requests the n_i element, where $1 \leq i \leq m$. The request in an amortised sense only costs $O(\lg t(n_i))$ as opposite to $O(\lg n)$ cost per operation on other search trees of size n.

The theorem states that recently accessed elements are faster to access than elements, which have not been accessed in a long time. Informally the theorem states that accessing an element x should depend on its working set number $t_i(x)$ and not solely on the number of elements in the data structure. Hence the cost is $O(\lg(t_i(x)))$.

1.2.2 Queueish property

John Iacono and Stefan Langerman introduces in [IL02] a property, which they call the *queueish* property and is useful to compare priority queues. It is a complementary property to the working set property. The main idea is that a data structure supporting the queueish property, should bound amortised *removeMin* cost for the smallest element x to $O(\lg q(x))$. Where $q(x)$ is the number of elements that have been in the priority queue longer than x. Besides bounding *removeMin* by $O(\lg q(x))$, a priority queue should do inserts in amortised constant time to satisfy the queueish property

Inserting elements into a priority queue structure in almost the same order as elements are removed, shows the advantages of the queueish property. Elements inserted in the above order are implying *removeMin* operations, which uses amortised constant $O(1)$ time per operation.

1.2.3 Sequential access property

The *splay* tree described in Chapter 3 has a sub-logarithmic access bound, in the number of elements in the structure, on certain sequences. Some of which is not captured by the working-set theorem. When repeatedly searching a data structure for a sorted sequence of n elements like in this example $\{1, 2, 3, \ldots, n, 1, 2, 3, \ldots, n, 1, 2, \ldots, n\}$ it is natural to believe that we can do better than $O(\lg n)$ per operation. This access sequence only has an amortised cost of $O(1)$ per access on *splay* trees. This result is known as the *sequential access lemm*a and was first proved by Robert Tarjan [Tar85b, Lemma 4-8]. This bound is proved by induction, rather than by amortisation like the working set theorem. The proof is not presented in this thesis.

1.2.4 Dynamic finger search property

The *sequential access lemma* can be generalised to the *dynamic finger theorem*. It was first conjectured in [ST85b, Conjecture 2] and later proved in the paper [Col00] from 2000 by Richard Cole. The dynamic finger theorem states that an access to an element, close in term of *rank distance* to a dynamic finger should be fast. The *rank distance* $d_i(x, y)$, is the number of elements in the data structure at time i, which are between x and y in key space, where x is included. In *splay* trees this access can be amortised bounded to $O(\lg [d_i(x_i, x_{i-1}) + 2])$. Another example of a data structure supporting this property is the level-linked tree by Mark Brown and Robert Tarjan [BT78, Section 4], which also supports accesses in $O(\lg(d_i(x_i, x_{i-1}) + 2))$ worst case cost.

1.2.5 Unified property

The working set and dynamic finger theorems are the best known bounds to analyse *splay* trees with respect to an access sequences. It is easily seen that none of the above methods capture all types of access sequences in one property. The following m length access sequences, from the set $\{1, 2, 3, \ldots, n-1, n\}$, where $m \geq n$ are examples, where some or none of the above methods bounds the access sequence tight enough:

$$1, 2, \ldots, n, 1, 2, \ldots, n, 1, 2, \ldots \tag{1.1}$$
$$1, n, 1, n, 1, n, \ldots, 1, n, 1, \ldots \tag{1.2}$$
$$1, \frac{n}{2}, 2, \frac{n}{2}+1, 3, \frac{n}{2}+2, 4, \ldots \ . \tag{1.3}$$

In Sequence (1.1) the dynamic finger theorem tightly bounds the access sequences to an access time of $O(m)$, here the working set theorem clearly states that the running time is bounded by $O(m \cdot \lg(n))$. In (1.2) the opposite is true, here the working set theorem tightly bounds the running time to $O(m)$, where the dynamic finger theorem claims that the running time equals $O(m \cdot \lg(n))$. In (1.3) none of the two properties result in the right bound. Both are strong enough to conclude that the bound is at most $O(m \cdot \lg(n))$ comparisons. This bound is not tight enough, it is clear that the real running time is $O(m)$. This is due to the fact that the element requested, always is close in key space, to an element which has a recently been accessed.

An comparison of access bounds can also be seen in Table 1.1.

Access Sequence	Working set	Dynamic Finger Search	Unified conjecture
(1.1)	$O(m \cdot \lg n)$	$O(m)$	$O(m)$
(1.1)	$O(m)$	$O(m \cdot \lg n)$	$O(m)$
(1.3)	$O(m \cdot \lg n)$	$O(m \cdot \lg n)$	$O(m)$

Table 1.1: Table explaining bounding different access sequences

Here is where the Unified Conjecture 1.1 also presented in [ST85b, Theorem 5, p. 660] is useful.

Conjecture 1.1. *(Unified conjecture) The amortised complexity for search, inserts and deletes x_i in a data structure supporting the unified property is*

$$O(\min_{y \in N_i} \lg\left(t_i(y) + d_i(x_i, y) + 2\right)) \ ,$$

where $t_i(x)$ is the working set described in Section 1.2.1, and $d_i(x_i, x_j)$ is the dynamic finger access time from [ST85b, Conjecture 2] presented

in Section 1.2.4. The number N_i donates the elements in the structure just before operation i.

It should be noted that this conjecture is strong enough to rightly bound Equation (1.3) to the amortised cost of $O(\lg(1+2+2)) = O(1)$ per access. Informally, the unified bound states that an access is fast, if n is close in key space, to some recently accessed element.

As stated above this conjecture was stated in 1985, but not until 2007 where M. Bŏdoiu et al. in [BCDI07] presented the *Unified Structure*, which we are describing in Section 2.2, where there a structure supporting this property.

2 Adaptive search algorithms

Look at the record.
Al Smith (1873 - 1944)

In this section we will present a few data structures that support some of the described properties like working set, queueish and unified property, all described in Chapter 1.

Adaptive algorithms is normally a term used for algorithms, which can adapt to the given input. In this thesis it is synonymous for search algorithms which perform faster than $O(\lg n)$ per access, for n sized collections on specific sequences. The disadvantages for adaptive algorithms is that their normally are complicated an hides a big hidden factor in the *Big-O* notation.

In Section 2.2 we describe the unified structure, which satisfy the stronger unified property, described in Section 1.2.5, this is a combination of the working set and the dynamic finger search property. Then in Section 2.3 we are going to describe the queaps structure, which where first described by John Iacono and Stefan Langerman [IL02] and satisfies the queueish property Section 1.2.2. Finally in Chapter 3, we are presenting the *splay* trees [ST85b] in details, since we are using the *splay* trees to compare with dynamic self adjusting skip lists in our empiric tests later since it satisfy the working set property. But before this we will present a simple scheme consisting of a linked list in Section 2.1.

2.1 Move to front heuristic

We are starting by describing one of the first algorithms designed to try and make newly accessed elements easy reachable. This algorithm is a *linked* list and the main idea is to move elements around in the list, when they are accessed.

The *move-to-front* (MF) heuristic was introduced in [ST85a]. It is an earlier scheme for making recently accessed elements in a collection easy reachable. We have a linked list of size n and accessing an element x on the i^{th} position is bounded by

the distance from the front to the element, which is $O(i)$. The linked list $\mathcal{L}$ contains n unordered elements and it supports the following three operations:

access(x) Locate x in $\mathcal{L}$ followed by moving x to the front of $\mathcal{L}$ and returning the element. The worst case cost is $O(i)$ if x is located otherwise $O(n)$.

insert(x) Check if x is already inserted, otherwise insert it as the first element. The worst case cost is $O(n)$.

delete(x) Locate x in $\mathcal{L}$ and remove it if found. The worst case cost is $O(i)$ if i is in $\mathcal{L}$, $O(n)$ otherwise.

Theorem 2.1. *The amortised time for accessing an element is at most $O(i)$ for an element at position i in the list. Insertion is bounded by $O(n)$ for a list of size n and deletions is bounded by $O(n)$, for a list with most n elements. This heuristic is in an amortised sense, within factor two compared to the optimal offline algorithm, which also represent data as a linked list.*

Proof. To prove this we use a potential function. Given two algorithms A_{MF} and A_{OPT}, where A_{OPT} is any list algorithm. Each algorithm, MF and OPT respectively, maintains a list containing the same n elements.

We define an inversion (*INV*) for two lists containing the same elements; Given two elements i and j, such that i appear anywhere before j in one list and anywhere after j in the other. We can hereafter define the potential function:

$$\Phi_{MF}(n) = |INV_{MF}| \ .$$

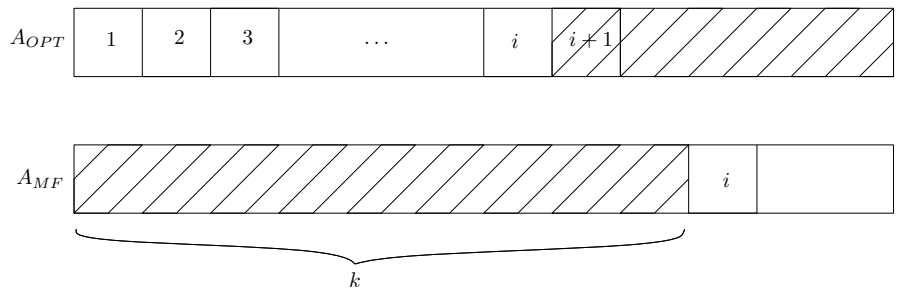


Figure 2.1: The hatched area in both MF and OPT shares x_i elements.

Consider an access in MF's list to element i. Let i be located at the k^{th} position in MF and let x_i be the number of elements which is preceding i in MF and succeeds i in OPT. The number of elements which precede i in both MF and OPT is then $k - x_i - 1$ (see Figure 2.1). This means that moving i from its position to the front

in MF creates $k - x_i - 1$ new inversions and remove x_i existing inversions. The amortised cost of any access in MF is then:

$$\begin{aligned} i^{th} \textit{ placement} + |\textit{new INV}| - |\textit{removed INV}| &= k + (k - x_i - 1) - x_i \\ &= 2(k - x_1) - 1 \ . \end{aligned}$$

It is clear that $k - x_i \leq i$, since there is $k - 1$ elements, which is preceding i in MF and there are only $i - 1$ preceding elements in OPT. Thus the amortised cost for an access is at most $2i - 1 = O(i)$.

The argument is virtually the same for deletes and inserts. Deletions do not create new inversions, so the amortised cost is $k - x_i \leq i = O(i)$.

We have showed that access costs at most $O(i)$ and inserts/deletes also is bounded by $O(i)$ if the element exists in $\mathcal{L}$ at position i and only a factor two worse than the optimal offline algorithm.

■

2.2 Unified structure

The paper by M. Bŏdoiu et al. [BCDI07] introduces a data structure called Unified structure. This data structure satisfy the unified access bound, described in Section 1.2.5.

The idea in this structure is to keep recently accessed elements close to the front of a list, in this case in one of the first smaller trees. An element which is close in key space to an element which has recently been accessed is only a constant distance away from each other in the maintained finger search tree. In this structure we will therefore place any element within a $\sum_{j=k+1}^{\infty} 4c2^j \cdot 1/2^{2^{j+1}} = O(1)$ distance of an element placed in a search tree.

The unified structure meets the requirement of Conjecture 1.1, and have a amortised access time of $O(\lg n)$.

2.2.1 The structure

The unified structure consists of $O(\lg \lg n)$ balanced search trees $\mathcal{T}_0, \mathcal{T}_1, \ldots, \mathcal{T}_l$, each having size 2^{2^j} after they have been rebuild, where $j \leq l$.

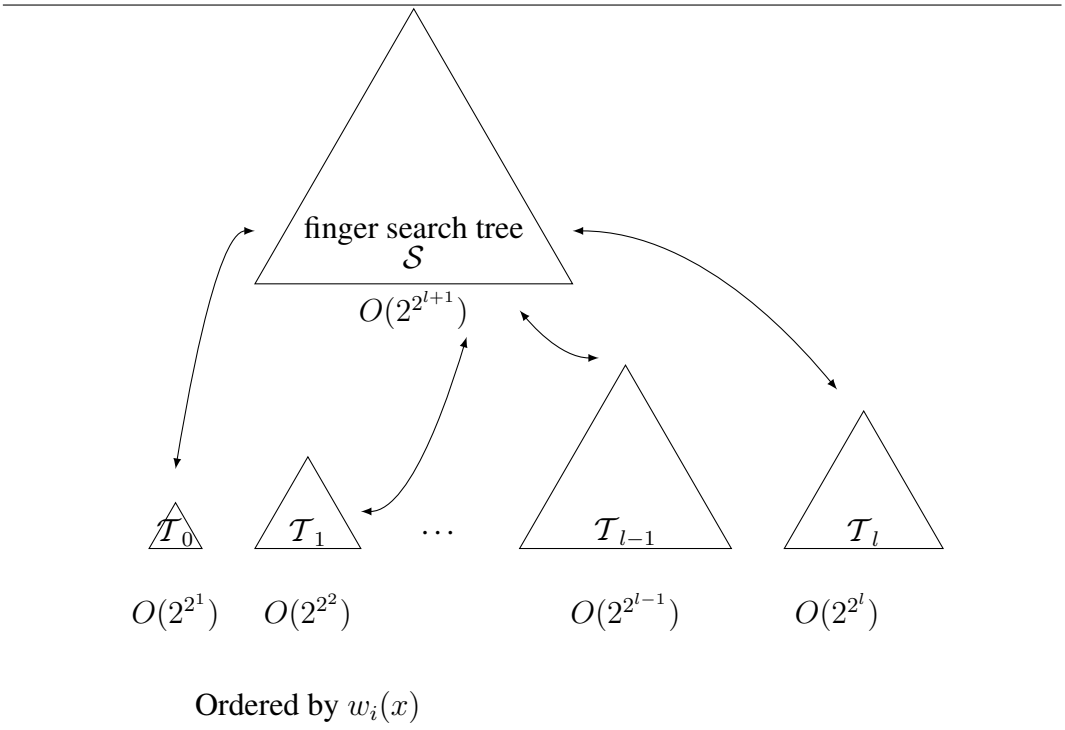


Figure 2.2: Unified structure containing n elements, which are placed in the *finger search tree* and in the $O(\lg \lg n)$ search trees.

During an operation, a tree $\mathcal{T}_j$ can not grow larger than $|\mathcal{T}_j| \leq |\mathcal{T}_j| + |\mathcal{T}_{j-1}| + \ldots + |\mathcal{T}_1| + |\mathcal{T}_0|$. After a dictionary operation *search*, *insert*, or *delete*, a tree should furthermore contain less than $2^{2^{j+1}}$ elements, otherwise an overflow operation is performed, this is described in Section 2.2.2.

Each element in the unified structure is placed in balanced search trees of growing size according to their working set number. An element is only represented in at most one search tree structure at once. Every element residing in $\mathcal{T}_i$ has all been accessed more recently than elements in $\mathcal{T}_j$ for $i < j$.

All n elements in the unified structure are placed in the finger search tree $\mathcal{S}$. All elements x_k are stored as indirect nodes, with a pointer to a node in $\mathcal{S}$ and a pointer to the search tree $\mathcal{T}_k$, where x_k are placed. This makes it easy to update nodes in $\mathcal{S}$ and the search trees when elements are moved around.

The search trees $\mathcal{T}_0, \mathcal{T}_1, \ldots, \mathcal{T}_l$ can be any tree structure that supports *insert*, *delete* and *search* operations in $O(\lg |\mathcal{T}_k|)$ time and updates a pointer to a node in constant $O(1)$ time. Similarly the finger search tree should support a search query, to a dynamic finger in $O(\lg(r+1))$ time, where r is the rank distance between elements.

2.2.2 Overflow

An overflow in the unified structure happens when a tree $\mathcal{T}_k$ becomes to large, $|\mathcal{T}_k| \geq 2^{2^{k+1}} - 1$. When this happens we rebuild $\mathcal{T}_k$ and all smaller trees $\mathcal{T}_j$, where $j < k$, so they have size 2^{2^j} for the j^{th} tree.

When tree $\mathcal{T}_k$ overflows, all elements residing in $\mathcal{T}_0, \mathcal{T}_1, \mathcal{T}_2, \ldots, \mathcal{T}_k$ trees are redistributed into trees $\mathcal{T}_j$, where $j \leq k$, such that no tree is larger than 2^{2^j} and the elements are inserted according to their working set number. This means elements with working set number $\leq 2^{2^j}$ are placed in a smaller tree than elements with working set number $> 2^{2^j}$. The remaining elements are then divided into two groups.

- The first group are elements x_i where there exists a smaller element $y < x_i$, within a rank distance of $2^{2^{k+1}}$ to x_i in $\mathcal{S}$. These elements are removed.
- The remaining elements are inserted into tree $\mathcal{T}_{k+1}$. The overflow can then cascade in $\mathcal{T}_{k+1}$.

After an overflow in $\mathcal{T}_k$, the sizes of the trees $\mathcal{T}_k, \mathcal{T}_{k-1}, \ldots, \mathcal{T}_1, \mathcal{T}_0$ all contains 2^{2^j} elements, for $j \leq k$ and tree $\mathcal{T}_{k+1}$ contains maximum $2^{2^{k+1}}$. The size of $\mathcal{T}_k, \mathcal{T}_{k-1}, \ldots, \mathcal{T}_1, \mathcal{T}_0$ all hold by construction. The elements added to $\mathcal{T}_{k+1}$ is at most $2^{2^{k+1}} + 2^{2^k} + \cdots + 2^{2^1} + 2^{2^1}$. This could potentially mean that $|\mathcal{T}_{k+1}| \geq 2^{2^{k+2}}$, but still means that $|\mathcal{T}_{k+1}| \leq 2^{2^{k+2}} + 2^{2^k} + \cdots + 2^{2^1} + 2^{2^1}$.

2.2.3 Search

A search for element x in the unified structure works by traversing each search tree $\mathcal{T}_k$ for $k = 0, 1, 2, \ldots, l$, and for each search tree updating a pair of elements $\{L, U\}$, where $L \leq x \leq U$. The pairs L and U are pointers to the elements closest to x in $\mathcal{S}$. Thus we can avoid searching in the same place in $\mathcal{S}$ more than once when searching for x.

In each tree $\mathcal{T}_k$, we search for the elements L_k and U_k closest to x, for $k = 1, 2, 3, \ldots, l$, such that $L_k \leq x \leq U_k$. In $\mathcal{S}$ we search for x within the ranges $[L, L + (k+4) \cdot 2^{2^k}]$ and $[U - (k+4) \cdot 2^{2^k}, U]$. If x is found in the finger search tree, then delete x from the search tree $\mathcal{T}_k$ if x is present in this. To maintain the unified searching bound, x is inserted into $\mathcal{T}_0$. This can of course result in an overflow.

The data structure maintains the invariant that every element x in $\mathcal{S}$ is within rank distance $(k+4) \cdot 2^{2^k}$ of some element x' in $\mathcal{T}_0 \cup \mathcal{T}_1 \cup \mathcal{T}_2 \cup \cdots \cup \mathcal{T}_l$. This invariant guaranties than an element x is within rank distance of an element in a search tree $\mathcal{T}_i$.

2.2.4 Inserts and deletes

An insertion of element x in the unified structure is performed by searching for x in $\mathcal{T}_0, \mathcal{T}_1, \ldots, \mathcal{T}_k$ and stopping when finding the predecessor x' in $\mathcal{T}_k$. Then x is inserted in both the search tree $\mathcal{T}_0$ and the finger search tree $\mathcal{S}$. The running time is as we will see in Section 2.2.7 dominated by the search for the predecessor, which is within the unified bound of the predecessor.

Deleting element x in the unified structure is simply a matter of finding x. If x is present in the search tree $\mathcal{T}_k$, then delete it. The element x should also be deleted from the finger search tree.

2.2.5 Potential in the unified structure

To analyse this data structure, we are using potential functions and to define this, we need to introduce a construct *j-graphs*. A j-graph consists of all nodes in $\mathcal{T}_0, \mathcal{T}_1, \ldots, \mathcal{T}_j$. There is furthermore an edge between all pair of nodes, where the rank-distance is less than $2^{2^{j+1}}$. A j-component is a connected subset of a j-graph. Let *extent* be the rank difference between smallest and largest element in a j-component.

The overflow potential of the unified structure is then

- Overflow potential of an j-component with *extent* e is $4c2^j(1 + e/2^{2^{j+1}})$.
- The j-overflow potential of the entire structure is the sum of overflow potential in each j-component.

This potential is chosen, because the overflow at a j-component from tree $\mathcal{T}_j$ to $\mathcal{T}_{j+1}$ will involve $\lfloor 1 + e/2^{2^{j+1}} \rfloor$ elements, and each element will cost $\Theta(2^j)$ to move.

Furthermore the structure keeps some *dead* potential equal to $4c \cdot \sum_{j=0}^{l} |T_j|$ or four times the size of the search trees times a constant.

2.2.6 Analysis of overflow operation

We will show that the change in potential of an overflow operation will be sufficient to pay for the actual work done in this operation. Merging the k trees will cost,

$$\begin{aligned}\sum_{j=0}^{k} O\left(|T_j| + \sum_{h=0}^{j-1} |T_h|\right) &= \sum_{j=0}^{k} O\left(2^{2^{j+1}} + \sum_{h=0}^{j-1} 2^{2^h}\right) \\ &= \sum_{j=0}^{k} O\left(2^{2^{j+1}}\right) \\ &= O\left(2^{2^{k+1}}\right) \\ &= O\left(r + d + p\right) \ ,\end{aligned}$$

where $r + d + p$ is the total number of elements in $\mathcal{T}_0, \mathcal{T}_1, \ldots, \mathcal{T}_k, \mathcal{T}_{k+1}$. More specific r is the elements still in $\mathcal{T}_0, \mathcal{T}_1, \ldots, \mathcal{T}_k \mathcal{T}_{k+1}$ after an overflow operation, p are elements inserted into $\mathcal{T}_{k+1}$ and finally d are elements, which have been deleted from the search trees. Removing the old elements from the trees is done in $O(r + d)$ time, and inserting the p elements into $\mathcal{T}_{k+1}$ with size $O(2^{2^{k+2}})$ takes $O(p \cdot 2^k)$. So the actual time for overflow operation is $O(r + d + p2^k)$.

The drop in dead potential is $4cd$ by construction. Adding the p elements to $\mathcal{T}_{k+1}$ does not change the j-overflow potential, for $j \leq k$ and the j-overflow potential of $j > k$ does not increase, because we have just deleted d elements. For each tree $j \leq k$ we remove all elements and re-insert $\sum_{h=0}^{j} 2^{2^h}$ elements into the j-graph. This could potential result in separate j-components, which would increase the j-overflow potential with $\sum_{h=0}^{j} 4c2^j 2^{2^h} \leq 6c2^j 2^{2^j}$. This results in an increase of overflow potential for $j \leq k$, which is at most $\sum_{j=0}^{k} 6c2^j 2^{2^j} \leq 7.5c2^k 2^{2^k}$. For each k-component with extend e we will insert at most $\lfloor 1 + e/2^{2^{k+1}} \rfloor$ elements into T_{k+1}, thus the loss of $4cp2^k$ potential in a k-overflow.

The amortised cost is the actual cost minus the increase in potential. The increase in potential is

$$7.5c2^k 2^{2^k} - 4cp2^k - 4cd \leq 7.5c2^k 2^{2^k} - cp2^{k+1} - cd - 2c(p + d) \ .$$

It holds that $p+d \geq |\mathcal{T}_k| - r \geq 2^{2^{k+1}} - \sum_{j=0}^{k} 2^{2^j}$ and $7.5c2^k 2^{2^k} \leq 2^{2^{k+1}} - \sum_{j=0}^{k} 2^{2^j}$, for $k \geq 3$, hence the gain in potential is no more than $-cp2^{k+1} - cd - c(p + d)$. Because

$$p + r \geq r\left(2^{2^{k+1}} - \sum_{j=0}^{k} 2^{2^j}\right) \geq \sum_{j=0}^{k} 2^{2^j} \ ,$$

for all $k \geq 0$, the increase is at most

$$-cp2^{k+1} - cd - cr \leq -c(p2^k + d + r) = -O(r + d + p2^k) \ ,$$

which is a negation of the actual cost, hence the amortised time for an overflow operation is $O(1)$.

2.2.7 Running time of dictionary operations

The running time of a search in the unified structure can be analysed as follows: Finding x if the element is within rank distance $(k+4) \cdot 2^{2^k}$ of an element in $\mathcal{T}_0 \cup \mathcal{T}_1 \cup \mathcal{T}_2 \cup \cdots \cup \mathcal{T}_l$ is completed at the k^{th} tree. The actual cost of searching k trees is $\sum_{j=0}^{k} O(\lg |\mathcal{T}_j|) = \sum_{j=0}^{k} 2^j = O(2^k)$. The running time for inserting and possible deleting from the search tree is $O(1)$. We need the difference in potential after completing a search operation to calculate the amortised running time of the operation. As described in Section 2.2.5, we do not consider the potential change of any overflow operation.

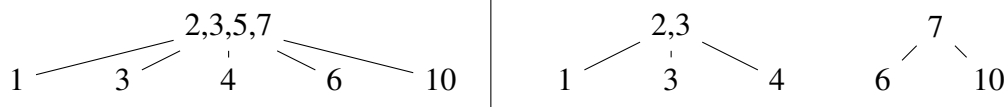
For each j-graph, where $j \leq k$, the worst case is to form a new connected component, which increases the potential with $O(2^j)$. This increases the total potential with at most $\sum_{j=o}^{k} O(2^j) = O(2^k)$. Donate y to the element in tree $\mathcal{T}_k$, from which the successfully finger search was initiated. We know that for $j \leq k$, x will appear in the same j-graph as y and that each of these will grow at most $\sum_{j=k+1}^{\infty} 4c2^j \cdot 1/2^{2^{j+1}} = O(1)$. Consequently the amortised running time for the search operation is within the unified bound.

The running time is within the unified bound because it is the search for x that is the dominating factor of the operation.

2.3 Queaps

In this section we are going to describe the *queaps* data structure, presented in by J. Iacono et al. [IL02], which is a priority queue data structure that satisfy the queueish property. The property states that *removeMin* cost is bounded by $O(\lg q_i(x))$ i.e. the number elements that have been longer in the structure than x.

In the *queaps* we have elements $x_1, x_2, \ldots, x_n$ in the order they are inserted. These are split around some k, the elements from $x_1, x_2, \ldots, x_k$ are inserted as leaves in a *(2-4)*-tree in the order they are inserted. The rest are represented in a linked list also in the order they are inserted.



(a) Before *split* (b) After *split*

Figure 2.3: *Split* operation on *(2-4)*-trees

Before going into the details surrounding the *queaps* structure, we are briefly describing the *(2-4)*-trees, which are used in the *queaps* data structure.

2.3.1 *(2-4)*-tree

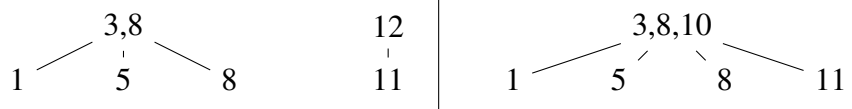
The *(2-4)*-trees are multiple directional search trees, which is a specialised version of the *(a,b)*-trees described by S. Huddleston and K. Mehlhorn [HM82, p. 157-166]. It supports updates (*delete*, *insert*) and searches in worst case $O(\lg n)$, where n is the number of elements in the tree. The tree also supports a sequence of insert and updates in amortised $O(1)$ time, which is explained below. The amortised complexity is the reason that the *queap* structure is utilising this structure as part of the main structure. In this section we will show the main ideas of the *(2-4)*-trees.

The structure consists of nodes and leaves, just like a normal binary search tree. The difference is that each internal node contains between 2 and 4 children, both inclusive. All elements inserted into the tree are placed at leaves, which all are at the same level. When a node is overflowing respectively emptying, i.e. the size is ≤ 2 or becoming to large, > 4, there is performed a *join* or *split* operation. These operations only affects a constant number of nodes, hence their time complexity can be bounded to $O(1)$.

Splitting node v with 5 or more children, is done by dividing v's children in two approximately equal sized sets v_1 and v_2, where $|v_1| = \lceil \frac{|v|}{2} \rceil$ and $|v_2| = \lfloor \frac{|v|}{2} \rfloor$. An example could be dividing node $v = k_1, k_2, k_3, k_4, k_5$ into nodes $v_1 = k_1, k_2, k_3$ and $v_2 = k_4, k_5$, see Figure (2.3a and 2.3b), where both nodes are within tolerance level.

Joining two nodes v_1 and v_2, where one consist of only a single child, is done by merging v_1 and v_2 into v. A *join* operation could result in a node v becoming to large, hence a *split* operation is needed. See figure (2.4a and 2.4b).

The height of *(2-4)*-trees $\mathcal{T}$ storing n elements are $\Theta(\lg n)$, because a given layer j contains between $2^j \leq |j| \leq 4^j$ elements. Therefore a *search* is in the worst case taking $O(\lg n)$ time to complete, because we only touch one node per level.



(a) Before *join* (b) After *join*

Figure 2.4: *Join* operation on *(2-4)*-trees

Now that we have the height of $\mathcal{T}$ and we established that any *join* or *split* operation takes $O(1)$ time per level, we can argue that an amortised bound of a *delete* or *insert* operation is at most $O(1)$.

We can define a non negative potential function $\Phi(n)$, which is equal to the number of nodes with 4 elements. We have two operations that effect our potential:

1. The potential is decreased with 1 for each split operation.
2. The potential is increased with 1 for each insert into a node with 3 elements.

Inserting an element x at node v containing 4 element, will overflow v, which can cascade all the way to the top. If this happens, all internal nodes on the path p from v to the root, in the worst case only contains 2 or 3 children after the overflow. We have $O(\lg n)$ split operation if a split cascade all the way to the root. The total drop in potential is $O(\log n)$. Before an leaf on this path overflow from leaf to root we can insert at least $\sum_{j=0}^{O(\lg n)} 2 = O(\lg n)$ elements, before we have to rebuild p again, which increases the potential with $\lg n$.

This means that these $O(\lg n)$ inserts can pay for rebuilding the tree by adding a constant amount of work to each insert operation, hence the amortised cost for an *insert* is $O(1)$. The argument for deleting an element is the same.

2.3.2 *Queaps* data structure

The elements in the data structure at time i, $\{x_1, x_2, \ldots, x_{k-1}, x_k, x_{k+1}, \ldots, x_n\}$, are ordered by insertion, hence element x_i is the i^{th} element inserted. These elements are divided around some k into two sets, which is donated O and N. The set O is containing the elements inserted before the $(k+1)^{th}$element and N are the new elements inserted after the k^{th} element. The elements are ordered in the same way as they are inserted into the *queaps*.

All elements in the set $O = \{x_{-\infty}, x_1, x_2, \ldots, x_{k-1}, x_k\}$ are stored in a *(2-4)*-tree $\mathcal{T}$ as leafs. Note that we are not using $\mathcal{T}$ as a search tree, but using it

for its amortised insert property. The special element $x_{-\infty}$ is a dummy leaf with an minus infinite key, that always sits as the leftmost element in $\mathcal{T}$, i.e the element that has resided longest in $\mathcal{T}$. All internal nodes v, except the ones on the path from the root to the leftmost node $x_{-\infty}$, contains a pointer h_v to the smallest element in the sub-tree T_v rooted v. The nodes from the root down to the leftmost leaf contains a pointer c_v to smallest element in $\mathcal{T}/\mathcal{T}_v$. This indicates that $x_{-\infty}$ has a pointer to the smallest element in $\mathcal{T}$. We also maintain an external pointer *minO* to $x_{-\infty}$.

All elements in the new set $N = x_{k+1}, \ldots, x_n$ are stored in a linked list $\mathcal{L}$, these elements are also store in the order they are inserted. We hold a pointer *minL* to the minimum element in this list. This data structure is visualised on Figure 2.5.

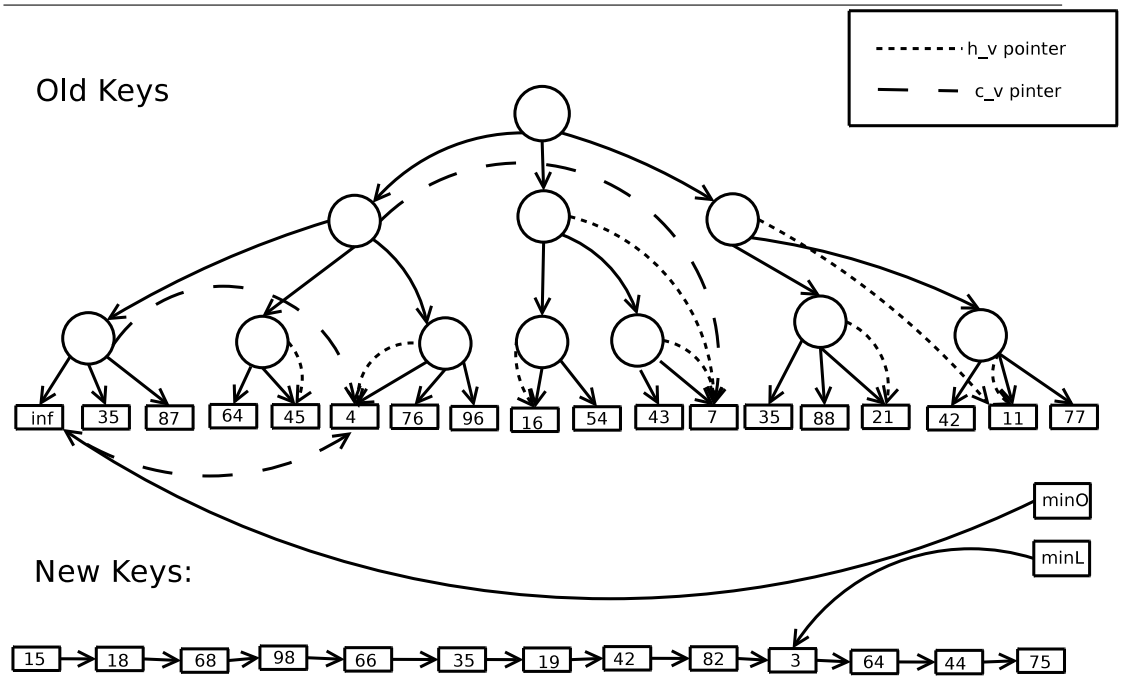


Figure 2.5: Queaps data structure

This data structure maintains a potential; so we can analyse the amortised cost of a heap operations. The potential function is $\Phi(Q) = c \cdot |\mathcal{L}|$, where $Q = \{\mathcal{T}, \mathcal{L}\}$, which is some constant c times the number of elements in N.

2.3.3 Inserts

Inserting x into the *queaps* is done by adding the element to the front of the linked list $\mathcal{L}$, and update the minimum pointer *minL* if necessary. This operation clearly takes $O(1)$ time and increases the potential by c.

2.3.4 Minimum

This operation do not change the data structure or the potential, but only returning a pointer to the minimum element in the *queaps*. This can be done by comparing the pointers *minL* and the *minO*. This operation is done in $O(1)$, thus it is only making a constant number of pointer operations.

2.3.5 Delete(x)

To delete an element x in the *queaps*, we first locate x. The search is started in $\mathcal{T}$, if x is not found in $\mathcal{T}$ then the search is continued to $\mathcal{L}$. If x is located in $\mathcal{L}$ we insert all elements from $\mathcal{L}$ into $\mathcal{T}$ and update all h_v and c_v pointers. If x is found in $\mathcal{T}$we can just delete x from $\mathcal{T}$ and update the affected h_v and c_v pointers.

The cost for this operation can be analysed by dividing the operation into two cases.

Case i) If x is located in $\mathcal{L}$, all elements from the linked list, are inserted into $\mathcal{T}$, which can be done at a cost $a \cdot |N|$ for some constant a. This is possible because the *(2-4)*-tree supports *insert* and *delete* operations in amortised constant time. When inserting into $\mathcal{T}$ we are updating pointers h_v in the tree $\mathcal{T}$, for all nodes that are effected. This is going to cost at most $a \cdot |N|$, because this does not affect more nodes than we are inserting into $\mathcal{T}$. This is the same as charging each insert a little extra. Updating the pointers c_v can be done by traversing the path from the root to x_∞ which also costs $O(\lg |O|)$.

The cost of this case is then $2a \cdot |N| + O(\lg |O|)$. We can conclude that all elements in $\mathcal{T}$ has been inserted before x which gives $q(x) > |O|$. By setting the potential $c > 2 \cdot a$, we have enough potential to pay for the drop of $c \cdot |N|$, which concludes that the total amortised running time is $O(\lg q(x))$.

Case ii) The other case is where x is already in the $\mathcal{T}$. We can delete x in amortised constant time and then we need to update the pointers h_v and c_v on a path from x to x_∞. The highest node on that path r is on the leftmost path from the root to x_∞ and is k levels above the leaves. This means that the left sub-tree of r donated r_{lchild} contains 2^{k-1} elements, which have been inserted before x, see Figure 2.6. This gives $q(x) \geq 2^{k-1}$. Hence the running time is $k = O(\lg q(x))$ which is the running time of the operation, since there is no change of potential.

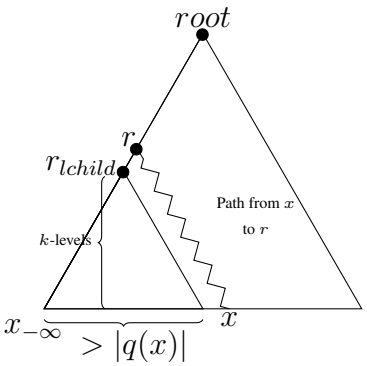


Figure 2.6: Path from x to $x_{-\infty}$

2.3.6 DeleteMin

This can be done by locating the minimum element x, hence the cost is $O(1)$ and then deleting x which costs $O(\lg q(x))$. The dominant factor here is the deletion of x, which therefore is the running time of the *deleteMin* operation.

3 Splay trees

Fall is my favorite season in Los Angeles,
watching the birds change color and fall
from the trees.
David Letterman (1947 -)

The *splay* trees data structure, are described by D. Slaetor and R. Tarjan [ST85b]. *Splay* trees are *self-adjusting* data structures, which support sequences of dictionary operations: *searches*, *inserts*, and *deletes*. These operations can be carried out quicker in an amortised sense by *splay* trees, than in ordinary search trees such as *balanced binary search* trees, *red-black* trees, *(2-4)*-trees, and their like, if the access sequence is skew.

Splay trees are binary search trees, which uses a heuristic called *splaying*, which moves accessed elements to the top of the tree by doing a sequence of rotations, called *splaying* steps. These steps are described in details in Section 3.1. This heuristic has a number of advantages as well as disadvantages. First of all, using *splay* trees over a sequence of accesses, is never more than a constant factor slower than optimal offline search tree, as described in Section 1.1.1. Secondly, as we will see in Section 3.3 when analysing the algorithm, *splay* trees can perform better than binary search trees in some cases like when the access sequences are skew. Thirdly as we are going to described in Section 3.4, the *splay* tree is easy to implement and consumes no space to maintain balance information. This makes the *splay* trees data structure very space efficient compared to other self balanced search trees and self-adjusting data structures.

The disadvantages are the potential long worst case access time for accessing a single element, this makes *splay* trees unusable in a real time application[1]. When the *splay* trees grow larger, rotation makes it very hard or even impossible to maintain locality in the stored elements and therefore accesses will potential result in many cache faults.

The data structure is very simple. It consists of elements, where each element contains the search data and child/parent pointers (see Figure 3.6 on page 34).

[1] Application where we cannot wait for long access queries i.e. where worst case access time are important.

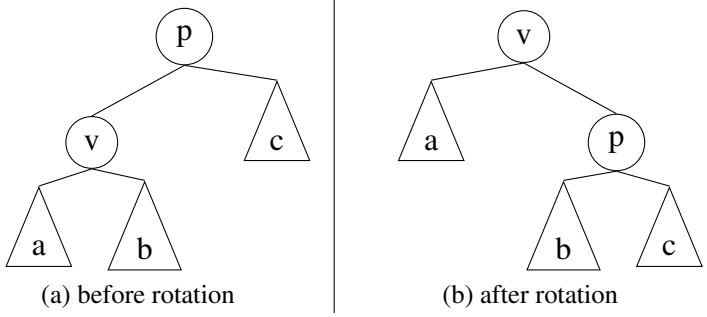


Figure 3.1: *Zig* step.

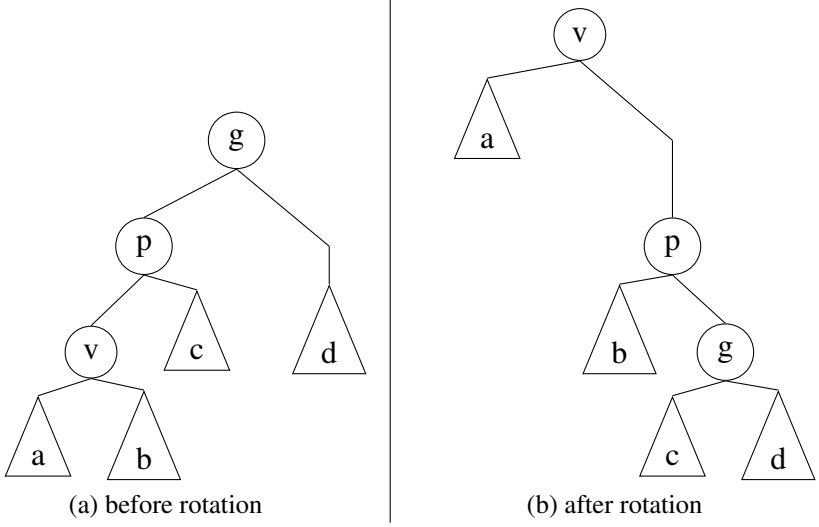


Figure 3.2: *Zig-zig* step.

3.1 *Splaying*

The *splay* trees data structure supports the following operations: *search*, *insert*, *delete*, *split* and *join*. Each of these operations are described below. Before going into details about each of the dictionary operations, a presentation of the *splaying* steps are needed.

In *splay* trees, *splaying* is the heuristic used to move an element from its present location to the root. This is done by using a combination of rotations called *zig*, *zig-zig* or *zig-zag*.

In Figures (3.1-3.3) it is shown how each of the three rotation types effects the tree around an accessed element v. All three operations can of course be mirrored, so

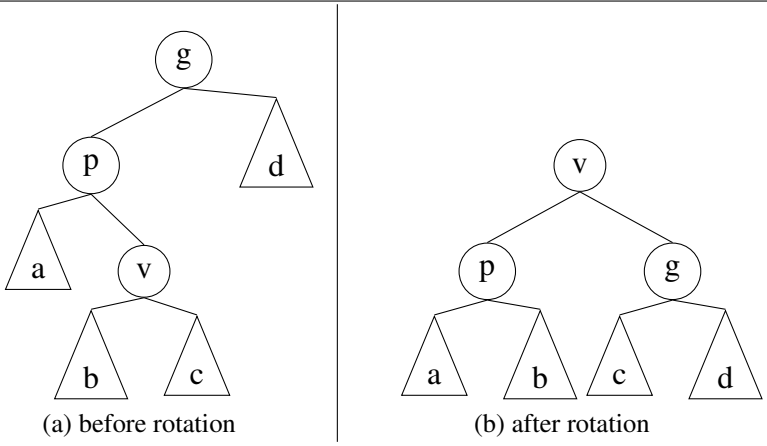


Figure 3.3: *Zig-zag* step.

that we in total have six types of possible rotations.

A sequence of *splaying* steps invoked on the same element v, will bring it from its present position to the root. Note that the *zig* step is only used when moving v the last step to the root, otherwise we are preferring*zig-zig* or *zig-zag* steps. A sequence of *zig-zig* on v steps half the length of the path from any node on search path to root.

It is important to note that a, b, c, and d is sub-trees and not just single elements. These sub-trees are all unaffected by the rotation. Each *splay* operation maintains the invariant that all elements in the left sub-tree of v has a smaller or equal key than the key in v, i.e. keys are increasing when visited in *inorder*

3.2 Dictionary operations

In this section we will present the dictionary operations *search*, *insert* and *delete* along with the help functions *split* and *join*. These help functions divide a *splay* tree into two parts around a target node v or join two *splay* trees. *Joining* two *splay* trees happens under the assumption that all keys in the left tree are smaller or equal to the every key in the right tree.

search(v) Searching in a *splay* tree $\mathcal{T}$, consists of a *search* part, where v is located and a part where v is *splayed* to the root. Locating v in $\mathcal{T}$ is done like in a binary search tree by moving downwards in $\mathcal{T}$. Going left or right at element w, if $k_v > k_w$ or $k_v < k_w$ respectively. When v is located, it is moved to the top of $\mathcal{T}$, with a series of *splaying* steps. If $\mathcal{T}$ does not hold a copy of v, then there are two possible

solutions: 1) stop the search operation here or 2) *splay* one of the elements v_- or v_+ to the top[2].

insert(x) An *insert* is done by locating key x in $\mathcal{T}$. This results in an element v, with key either equal to x, x_-, or x_+. This means that we can insert x as left or right child of v according to the value of v. Afterwards, x is *splayed* to the root of $\mathcal{T}$.

delete(v) Deleting is done by conducting a search for v. If v exists in $\mathcal{T}$, v is *splayed* to the top, and we perform a ***split_around***(v), as described in the next paragraph. This returns the two sub-trees[3] $\mathcal{T}_{<v}$ and $\mathcal{T}_{>v}$ and the element v. The element v is just deleted, and the two sub-trees are joined accordingly to the ***join***($\mathcal{T}_{<v}$, $\mathcal{T}_{>v}$) operation.

($\mathcal{T}_{<v}, n, \mathcal{T}_{>v}$) **split_around**($v$) A split around v in $\mathcal{T}$ assumes v are root in $\mathcal{T}$. The left sub-tree of v is now assigned to $\mathcal{T}_{<v}$, while the right sub-tree is assigned to $\mathcal{T}_{>v}$ and v's child pointers are set to *nil*. The triple $(\mathcal{T}_{<v}, v, \mathcal{T}_{>v})$ are now three distinct *splay* trees, which are returned.

join($\mathcal{T}_{<v}$, $\mathcal{T}_{>v}$) The join operations assumes that all elements in $\mathcal{T}_{<v}$ are smaller than elements in $\mathcal{T}_{>v}$, for some v. In $\mathcal{T}_{>v}$, we *splay* the smallest element (respectively largest element in $\mathcal{T}_{<v}$) to the root. This results in $\mathcal{T}_{>v}$ having the smallest element $\mathcal{T}_r$ (respectively the largest element in $\mathcal{T}_{<v}$) as root, and no left (right) child. Joining is then a simple matter of attaching $\mathcal{T}_{<v}$ as the left child ($\mathcal{T}_{>v}$ as right child) of $\mathcal{T}_r$, see Figure 3.4.

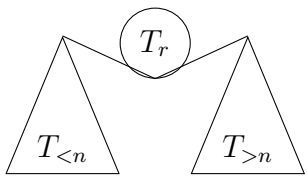


Figure 3.4: *Joining* $\mathcal{T}_{<v}$ to the root $\mathcal{T}_r$ of $\mathcal{T}_{>v}$.

[2] Let v_- be the largest element, which is smaller than v. Equally let v_+ be the smallest element which is larger than v.

[3] Here $\mathcal{T}_{<v}$ contains all elements smaller, respectively larger in $\mathcal{T}_{<v}$, than k_v.

3.3 Analysis

The interesting part of the analysis of the *splay* trees by D. Slaetor and R. Tarjan are the *Working set theorem* (Theorem 3.2) presented below. This states an access of element v is only using $O(\lg t_i(v))$ time amortised, where $t_i(v)$ is the number of unique accesses at time i since element v was last accessed. The other interesting part is the *Balance Theorem* (Theorem 3.3), which states that the amortised cost of an access is only $O(\lg |\mathcal{T}|)$. In this section we will present these two profs along with analysis of the worst case access bound.

Amortised access cost In defining the potential of the *splay* tree, $\mathcal{T}$, we give each node v in $\mathcal{T}$ a fixed weight w_v. Let $s(v)$ be the sum of weights for each node v' residing in the sub-tree of v and we donate the rank r. The rank is defined as $r(v) = \lg s(v)$.

We will as the main thing, in the **Access Theorem** below, show that when accessing elements the potential in the *splay* tree increases, whenever performing a *zig-zig* step.

Theorem 3.1. ***(Access Theorem)*** *The amortised cost of* splaying *a tree $\mathcal{T}$ with root t, at node v is at most* $3(r(t) - r(v)) + 1 = O(\lg(s(t))/\lg(s(v)))$.

Proof. If there is no rotation done, then the bound is trivial. If we suppose there are at least one rotation, then let us consider any *splaying* step. Let s and s', r and r' donate the respective functions just before and after a *splay* step.

Let p be the parent of v, and g be the parent of p, if these nodes exists.

Case A) A *zig* step is done, then the amortised cost is:

$$\begin{aligned}
\Phi(i-1) - \Phi(i) &= r'(v) + r'(p) - r(v) - r(p) && \text{Only } v \text{ and } p \text{ are rearranged.} \\
&\leq 1 + r'(v) - r(v) && r(p) \leq r'(p), \text{ since } p \text{ is moved down in } \mathcal{T}. \\
&\leq 1 + 3(r'(v) - r(v)) && r'(v) \geq r(v), \text{ since } v \text{ is moved up in } \mathcal{T}.
\end{aligned}$$

Case B) A *zig-zig* step is done, then the amortised cost is then:

$$
\begin{aligned}
\Phi(i-1) - \Phi(i) &= 2 + r'(v) + r'(p) + r'(g) \\
&\quad -r(v) - r(p) - r(g) && \text{Only } v, p, \text{ and } g \text{ are moved in } \mathcal{T}. \\
&= 2 + r'(p) + r'(g) - r(v) - r(p) && \text{Since } v \text{ is in } g\text{'s old place in } \mathcal{T} \text{ after the } \textit{splay} \text{ step, and the sub-trees have same size, } r'(v) = r(g). \\
&\leq 2 + r'(v) + r'(g) - 2r(v)\ . && g \text{ is now hanging under } v, \text{ hence } r'(v) \geq r'(g). \text{ Before the } \textit{splay} \text{ step } p \text{ was located above } v, \text{ thus } r(y) \geq r(v).
\end{aligned}
$$

The claim is then that:

$$
\begin{aligned}
2 + r'(v) + r'(g) - 2r(v) &\leq 3(r'(v) - r(v)) \\
2 + r'(g) &\leq 2r'(v) - r(v) \\
2 &\leq 2r'(v) - r(v) - r'(g)\ .
\end{aligned}
$$

To maximise the function $\lg x + \lg y$ for $x, y \geq 0$ and $x + y \leq 1$ we can look at the convexity of the *log(x)* function. We can conclude the function is maximised where $x = y = \frac{1}{2}$, where it takes the value -2. From this it follows that

$$
\begin{aligned}
r(v) + r'(g) - 2r'(v) &= \lg s(v) + \lg s'(g) - 2\lg s'(v) \\
&= \frac{\lg s(v)}{\lg s'(v)} + \frac{\lg s'(g)}{\lg s'(v)} \\
&\leq -2\ ,
\end{aligned}
$$

since $s(v) + s'(g) \leq s'(v)$, see Figure 3.5. Here we gain some potential, and therefore concludes the amortised cost spent at **Case B** is $< 3(r'(v) - r(v))$.

Case C) The amortised cost of a *zig-zag* step is:

$$
\begin{aligned}
\Phi(i-1) - \Phi(i) &= 2 + r'(v) + r'(p) + r'(g) \\
&\quad -r(v) + r(p) + r(g) && v, p \text{ and } g \text{ are the only nodes moved.} \\
&\leq 2 + r'(p) + r'(g) - 2r(v)\ . && r'(v) \leq r(g).
\end{aligned}
$$

In the same manner as in Case 2, it follows that:

$$
2 + r'(p) + r'(g) - 2r(v) < 2(r(v) - r'(v))\ ,
$$

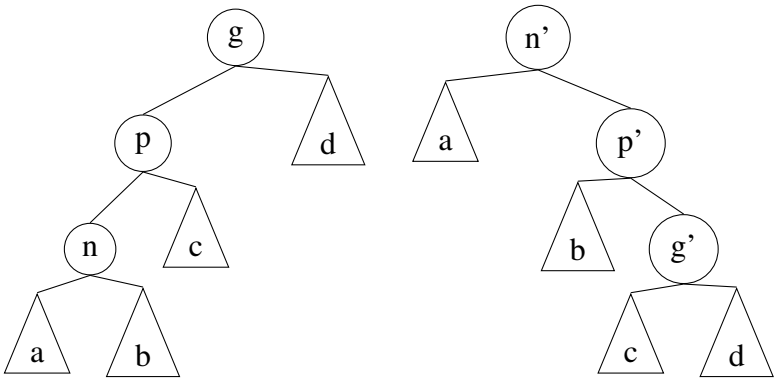


Figure 3.5: It is clear to see that $s'(v) \geq s'(g) + s(v)$

which means:

$$\begin{aligned} 2 + r'(p) + r'(g) - 2r(v) &\leq 2(r(v) - 2r'(v)) \\ 2 &\leq r'(p) + r'(g) - 2r'(v) \ , \end{aligned}$$

when $s'(p) + s'(g) \leq s'(v)$. It then holds that the amortised cost of *zig-zag* is $\leq 3(r'(v) - r(v))$.

This theorem is concluded by summing all the *splay* steps together. The amortised time for the sum of all the *splay* steps is clearly $\leq 3(r'(v) - r(v)) = 3(r(t) - r(v))$, which concludes the proof.

■

Now that we established that *splaying* a node can be done in amortised time equal to $O(\lg(s(\mathcal{T})/s(n)))$, it is time to establish the time for accessing a sequence of elements. But before proving the **Balance Theorem** we are presenting and proving the **Working set theorem** below.

3.3.1 Working set

The working set property, presented in Section 1.2.1, states that it is easier to access an element, which has recently been accessed. The *splay* tree satisfy this property over a sequence of accesses.

Theorem 3.2. ***(Working set theorem)*** *For a series of m accesses to a* splay *tree of size n, where $t_i(v)$ is the working set number at time i, where $1 \leq i \leq m$. The the total time for the m accesses is $O(n \lg n + m + \sum_{i=1}^{m}(\lg(t_i(v) + 1)))$.*

Proof. Start by assigning the following weights to all elements in the *splay* tree $1, \frac{1}{4}, \frac{1}{9}, \frac{1}{16}, \ldots, \frac{1}{n^2}$. The weights are assigned in increasing order to the nodes, which has most recently been accessed, such that the last accessed node has weight 1, the next most recently accessed node weight is assigned $\frac{1}{4}$ etc.. Whenever an access occur, we redefine the weights in the following order: Given an access to element v_j during the j^{th} access, the weight of v_j is equal $\frac{1}{k^2}$, where k is the working set number of v_j. After access j, assign weight 1 to v_j and reassign weights to the elements $l' = 1, 2, \ldots, j-1$, such that the new weight is equal to $\frac{1}{(k_l+1)^2}$, for $l \leq j-1$. This reassigning permutes the weights $1, \frac{1}{4}, \frac{1}{9}, \frac{1}{16}, ..., \frac{1}{n^2}$ among the elements and guaranties that each element weight is equal to $\frac{1}{t_i(v)+1^2}$.

The sum of the set of weights W after an access is $W = \sum_{k=1}^{n} \frac{1}{k^2} = O(1)$. Because the element is *splayed* to the root after an access, the weight of the root node is increased as result of an access. The sizes of the roots sub-trees are unchanged, but the sizes of other nodes can decrease as a result of rotations done during the *splaying*. The potential of the *splay* tree is decreasing because of the reassignment. The amortised time for this reassignment is either zero or negative.

The amortised access time to element v_i is $O(\lg(t_i(v)+1))$. The drop of the potential over the m length sequence is $O(n \lg n)$, because we pay $O(\lg n)$ for the first n accesses.

All together this means that a sequence of m accesses to a n node *splay* tree costs, the *drop in potential* plus m plus *sum of access*, which equals,

$$O\left(n \lg n + m + \sum_{i=0}^{m} \lg(t_i(v)+1)\right).$$

This concludes the proof. ■

3.3.2 Balance Theorem

Before turning to the **Balance Theorem** it is worth noticing that the weight of the root in the *splay* tree $\mathcal{T}$, if the weights are fixed is $w(T) = \sum_{i=1}^{n} w(i)$. This means that decrease in potential over a sequence of length m om a n-node *splay* tree, is at most $\sum_{j=1}^{m} \lg(W/w(j))$.

Theorem 3.3. ***(Balance Theorem)*** *Total access time for accessing m nodes on a n-node* splay *tree is $O((m+n)\lg(n))$.*

Proof. Start by assigning weights $\frac{1}{n}$ to each node, then $w(T) = w(x_{root}) = \sum_{i=1}^{n} w(i) = 1$. For any given *splay* operation it holds

that $r(t) \leq \lg n$, and $r(v) \geq 0$. Applying the **Access Theorem** (3.1) we get the amortised cost for an access is no more than,

$$3 \lg n + 1 = O(\lg n) \ .$$

If we set the potential of the *splay* tree equal to total number of elements in the tree, we can bound the cost of a sequence of accesses. We add the cost of a single *splay* operation for each access plus the potential drop over the sequences, which is at most $O(n \lg n)$. The final potential is by definition greater than *zero*. This all adds up to

$$m(3 \lg(n) + 1) + O(n \lg(n)) = O(m \lg(n) + n \lg(n)) = O((m+n) \lg(n)) \ ,$$

which concludes the proof. ■

This means that a series of m accesses to elements in a *splay* tree $\mathcal{T}$, can be done just as fast, in an amortised sense, as other binary search trees.

Lastly we will show why *splay* trees are not on ideal data structure for real time applications, because their worst case access time is much larger than normal search trees.

Worst case access It is easy to see that inserting n sorted elements into a *splay* tree in increasing (respectively decreasing) order results in a tree with height $O(n)$. This is because at each *insert*, the inserted node v is placed just below the root r, as the right (respectively left) child. When *splaying* v to the root, the only necessary step is a normal *zig* step, that place the old root r as the left (respectively right) child of v.

When searching for the smallest (respectively largest) element in the *splay* tree, it is necessary to follow the pointers from the root to the node v_{smallest} which is $O(n)$ pointers, because the height.

Inserts and deletes are bounded by the same bounds as above. Of course when inserting an element, we can use the working set number of the elements parent.

3.4 Implementation

The *splay* trees implementation can be found on the URL: `http://www.cs.au.dk/~henrik/thesis/`, all references are within this directory.

As stated before, the *splay* trees are easily implemented. The actual implementation can be found in */src/SplayTree/* and is used for the experiments in Chapter 6. Compared with the description of the structure there are a couple of small differences in the implementation. First we present a couple of general differences between the implementation and the general description of the *splay* trees. Secondly we are in brief going to describe the *splay* trees nodes, which contain the data needed by the *splay* trees implementation.

3.4.1 The *splay* tree operations

The implemented data structure is only having a public interface for the *insert*, *delete* and *search* operations. The *join* and *split* operation are implemented, but only used as internal functions, used by the delete operation. Both the *insert* and *search* functions are implemented using an iterative top-to-bottom search, succeeding by possibly *splaying* of node v from its original position to the root.

The delete operation is implemented as a search operation followed by a split operation, which works as described in Section 3.2. After v are *splayed* to the root of $\mathcal{T}$, v is deleted and both $\mathcal{T}_{<v}$ and $\mathcal{T}_{>v}$ are remaining as independent *splay* trees. The delete operation is concluded by *splaying* the largest element from the left sub-tree $\mathcal{T}_{<v}$ and attaching $\mathcal{T}_{>v}$ as the right child of $\mathcal{T}_{<v}$'s new root.

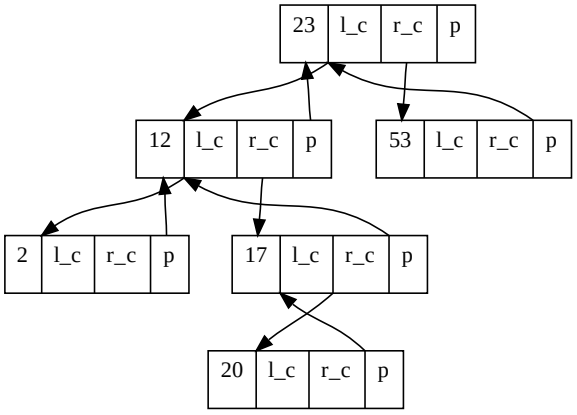


Figure 3.6: Pointers in *splay* tree implementation. The pointers r_c, l_c and p are respectively for right or left child or parent.

The *splaying* of a node v from its current position in a *splay* tree to the root, is done iteratively, apposed to recursively. This means that searching also is iterative. A iterative method normally has some advantages compared to a recursively approach. The stack is growing much slower, but on the contrary the iterative method is normally more difficult to implement. When *splaying* v, it is necessary for the algorithm to know the neighbourhood around v, to choose the right *splay* rotation. The information gathering can be done in a constant number of pointer look ups. Each pointer look up, can of course result in a *page fault*[4]. Even though this does not change the bound of the algorithm in the RAM model, it means a slower running time in practise. Now v has the needed information about its' parent and grandparent and can determine which *splay* step to perform. The *splaying* is done iteratively until v is the root of $\mathcal{T}$.

The implementation is of course only using the *zig* step at the last step.

3.4.2 *Splay* nodes

For a *splay* node v in $\mathcal{T}$ to know about its surroundings, it is necessary to possess parent pointers, when perform the *splaying* iteratively, to avoid storing a list of visited nodes. This means that each node contains 3 pointers, one for each child and one for the parent, see Figure 3.6. If the node is containing integers as data, this means that each node is using three times 32 bits for pointers plus 32 bits for data in memory[5].

[4]Which forces the machine to retrieve data from main memory and fetch this data into *CPU* cache.

[5]This depends on the machine architecture.

4 Probabilistic and deterministic skip lists

Anyone who considers arithmetical
methods of producing random digits is,
of course, in a state of sin.
John von Neumann (1903-1957)

In [BDL08] P. Bose et al. introduce dynamic self adjusting skip lists, which they have shown to have same complexities as *splay* trees [ST85b], described in Chapter 3. We will present the dynamic self adjusting skip lists in details in Chapter 5, before we are giving both the details concerning analysis and implementation we will present some skip list structures. First we will introduce the probabilistic skip lists structure as described by William Pugh [Pug90] in Section 4.1, this was the first skip list structure described. Before we are presenting *biased* skip lists used by dynamic self adjusting skip lists we are also going to present a simpler deterministic skip list, developed by J. Munro et al. [MPS92] in Section 4.2. The *(a,b)* biased skip lists are a fundamental data structure in understanding the dynamic self adjusting skip lists, in Chapter 5 and is presented in Section 4.3.

The skip lists where originally described as a probabilistic data structure, by William Pugh [Pug90] since then, there have been developed a number of deterministic skip lists structures as well.

4.1 Probabilistic skip lists

The main idea of the probabilistic skip list structures are that it is simpler to implement and has an excepted search time, which is equal to that of normal search structures, such as binary search trees, *(a,b)*-trees, red-black trees etc..

The skip list $\mathcal{L}$ consists of n sorted elements and an optional special header element $v_{-\infty}$. We assign a probability p with $\mathcal{L}$, p is a real number in the interval $[0, 1]$. The height of each element in $\mathcal{L}$ is determined on the basis of p. When

inserting an element v into $\mathcal{L}$ a dice[1] is rolled until it shows a number $q > p$. The number of rolls where $q \leq p$ is equal to the height $h_v = h(v)$ of v. Each node w has h_w pointers. At each level i, where $0 < i \leq h_v$, we place a pointer to the next element w', where $h_{w'} \geq i$ in $\mathcal{L}$. The special header element is always the first element in the list and has the same height as the highest element in $\mathcal{L}$. The last element at any given level j is *nil* terminated. An illustration of the skip lists can be seen at Figure 4.1.

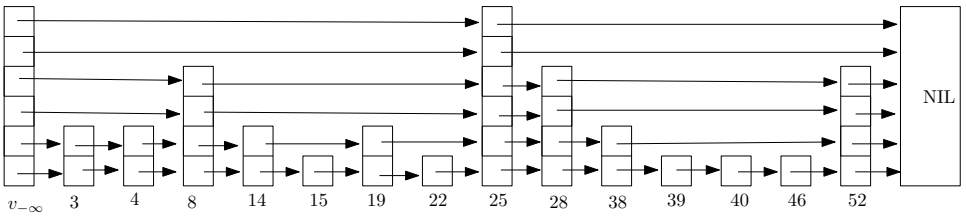


Figure 4.1: Probabilistic skip list, with forward pointers and *nil* terminated.

4.1.1 Dictionary operations in the structure

Searching for v in $\mathcal{L}$ is initiated from the top of $v_{-\infty}$. Then there are two possible ways to navigate at each level. We go right if the next element w has a key $k_v > k_w$ or we move downwards one level if $k_v < k_w$. This guaranties that we find n if the key is present in $\mathcal{L}$, because all elements are placed in sorted order and we never pass an element, which is greater than v.

Deleting is initiated by locating v, if v is present in $\mathcal{L}$, it is found at some level j. Then for each level $0 \leq i \leq j$, we change the pointers between the succeeding and preceding elements. The pointers are changed such that the element preceding v now points at the element succeeding v at each level i. Deleting v is done when no element points to v.

Insertion is done similar, first locating[2] v_-, then using a dice according to Section 4.1 with probability p, choosing the height h_v of the new element v. When the height is found, v is inserted after v_-. Every pointer at level i, where $0 \leq i \leq j$, are changed such that the element v_- preceding v at level i now points to v and v points to the element succeeding v_- before inserting v, at level i.

[1] Special dice which rolls uniform distributed numbers in the interval $[0, 1]$.

[2] Introduced on Page 28.

4.1.2 Time complexities of operations

The expected height h of $\mathcal{L}$ is the height of element v, where $v = \max(h_w)$, for all elements in $\mathcal{L}$. Each element has a probability of p^i of ending at the i^{th} level. This means that the probability P_i of level i having more than one element is $P_i \leq n \cdot p^i$, which again means that the expected height of $\mathcal{L}$, for some constant c is $c \lg n$ with probability at most $1/n^{c-1}$. So with high probability we can say that the expected height of $\mathcal{L}$ is bounded by $O(\lg n)$.

The search time of $\mathcal{L}$ is the dominating factor of the all three dictionary operations *search*, *delete* and *insert*. As stated before, we can only move down and right. The expected number of steps we move down is the height of $\mathcal{L}$, which is expected $O(\lg n)$ level. So we need to bound the number of forward scans. We say that at level i we examine e elements before going to level $i-1$. The keys examined at level i cannot be visited again at level $i-1$, thus we would already have visited them at level i. The probability for an element v' being visited at level i is p. This means that the number of elements e encountered on a level i, is $e - (1/2)e = 2p$. For $p = 1/2$ this is bounded $2 = O(1)$ elements. So the probabilistic cost of searching for v in $\mathcal{L}$ is $O(\lg n)$. Inserting and deleting is bounded the same way, hence the search for an element is the dominating factor of the execution time, namely the same as the maximum expected number of levels, where we need to update pointers when inserting and deleting.

Theorem 4.1. *The probabilistic skip list with n elements, has expected height $r = O(\lg n)$ with high probability.*

Proof. The number of levels $r = 1 + \max_{x \in \mathcal{L}} l(x)$, where levels $l(x)$ are random variables, distributed geometrically with parameter $p = 1/2$. We can view the n elements as independently geometrically distributed random variables $X_1, X_2, X_3, \ldots, X_n$. It is clearly seen that the probability $Pr[X_i > t] \leq (1-p)^t$ and therefore,

$$Pr[\max_i X_i > t] \leq n(1-p)^t = \frac{n}{2^t} \ ,$$

size $p = 1/2$. Using $t = c \cdot \lg n$ and $r = 1 + \max_i X_i$, we obtain the probability for $\mathcal{L}$ having less than $c \cdot \lg n$ elements,

$$Pr[r > c \cdot \lg n] \leq \frac{1}{n^{c-1}} \ ,$$

for all $c > 1$, which concludes the proof. ■

It is of course an expected search time, the worst case is clearly $O(n+h)$. Though this is highly unlikely and therefore not fair to compare probabilistic skip lists and ordinary search trees by their worst case complexity.

4.2 Deterministic skip lists

A deterministic version of the probabilistic skip lists described above, is developed by J. Munro et al. in [MPS92]. In the paper they introduces three different skip lists, each are deterministic and support logarithmic access. The first version, which we will present in this thesis, are using linear space and have $O(\lg^2 n)$ updates cost. The second list promises logarithmic updates at the cost of using more space, the third version is improving on the space requirements, without compromising the logarithmic access and updates bounds.

The structure of this deterministic *1-2* skip list $\mathcal{L}$ is much like the probabilistic skip lists described in Section 4.1; We have an ordered set of n elements, each element v has height $h(v)$. We require that between any two consecutive elements, on level i, there are at most 2 elements on level $i-1$.

When searching for x in $\mathcal{L}$ we can guaranty that we moves at most 2 steps horizontally, before going down one level. Therefore we halves the number of nodes we potentially will look at every time we move down one level. This bounds the access time to $O(\lg n)$ comparisons.

Inserting x into $\mathcal{L}$ means inserting x after x_- at level 1. If there now are 3 elements in a row at level 1 the height of the middle element is increased with 1. In the worst case we need to rise an element on all levels in $\mathcal{L}$. If implementing the horizontal pointers as arrays, we need to copy pointers in and out of an array. This means increasing the height of level h to $h+1$ will change $\Theta(h)$ pointers. Therefore this costs at most $\Theta(\lg^2 n)$ comparisons to insert an element. Deleting an element is done analogous and having the same bound as inserts.

J. Munro et al. presents in [MPS92] an optimisation, where they bounds the updates to $\Theta(\lg n)$, this is done by increasing the height of nodes exponential, thereby guarantying that we visit only one element at each level.

So now that we have seen some examples on deterministic and probabilistic skip lists, we are ready to show the *(a,b)* biased skip lists.

4.3 *(a,b)* biased skip lists

The biased skip lists presented by A. Bagchi et al. [BBG02, pages 31-41], is a skip list structure. In this structure the authors are placing constraints on how to place elements in this structure. All elements are given a weight and they are placed according to this weight. If we associated high weights with a high access probability, this will result in placing frequently accessed elements close to the top. This will give a faster access time if the access probability is right. The *biased* skip list is presented in both a deterministic and a randomised version [BBG02, pages 44-47]. We are only going to present the deterministic skip lists in this thesis.

The *biased* skip list $\mathcal{L}$ consist, like probabilistic skip lists, see Section 4.1, of n sorted elements. Each element has a height h_v and a forward pointer for each level $i < h_v$, to the succeeding element at the same level. Two elements v and v' are called *consecutive* if $h_w < \min(h_v, h_{v'})$ for all w, where $k_v < k_w < k_{v'}$. A *plateau* is the maximal number of *consecutive* elements with same height, which are not interrupted by a higher element. Each element v in biased skip lists holds, as described above, besides the key and forwards pointers also a weight w_v, without loss of generality we say that $w_v \geq 1$, for all nodes. The rank is defined as $r_v = \lfloor \log_a w_v \rfloor$. We can now define an *(a,b)* biased skip list. For integers a and b, such that $1 \leq a \leq \lfloor b/2 \rfloor$, it should apply that for each element v, the height h_v should be $h_v \geq r_v$. Furthermore the following invariants should also be satisfied after each dictionary operation:

I1) At any given height i there is $\leq b$ consecutive elements at any place.

I2) For any element v_i on level i there are $> a$ elements of height $i - 1$ between v_i and any preceding or succeeding element on level i.

This means that the placements of elements in the *biased* skip list is structured by these two invariants and means we can calculate the height of the list. In this structure it is furthermore necessary to remove redundant pointers to use constant space in the number of elements. So for all adjacent elements v and v', where there are no elements between v and v', we remove pointers from v at level $0 \leq i \leq \min(h_v, h_{v'}) - 1$.

4.3.1 Dictionary operations

The operations we are presenting in this thesis are *search*, *insert* and *delete*. We are describing searching in *(a,b)* biased skip lists. With *insert* and *delete* we presents the necessary details to restore the invariants after an update, given the invariants are satisfied before.

Searching for v_x in an *(a,b)* biased skip list $\mathcal{L}$, is done like in the probabilistic skip list, described in Section 4.1. The search is initiated from the top left, and moves either right or downwards, by comparing v_x with the next element's key y. In an *(a,b)* biased skip list, we can deterministically determine the access time, because we can calculate the height. When searching for v_x in $\mathcal{L}$, we can state by **I1** that we are not visiting anymore than $b+1$ elements at each level $i \leq H(\mathcal{L})$, where $H(\mathcal{L}) = \max(h_v)$, for all elements v in $\mathcal{L}$, or the maximum height of any element in $\mathcal{L}$. The depth of an element v is donated $d_v = H(\mathcal{L}) - h_v$.

Lemma 4.2. ***(Depth Lemma)*** *The depth of any element v in $\mathcal{L}$ is $O(\log_a(W/w_v))$.*

Proof. It is given that the number of elements at any given rank, that can appear in a higher level decreases geometrically by level. We define $N_i = |\{v : r_v = i\}|$ and $N'_i = |\{v : r_v \leq i \wedge h_v \geq i\}|$. It holds that,

$$N'_i \leq \sum_{j=0}^{i} \frac{1}{a^i} N_j \ . \tag{4.1}$$

For height 0, $N'_0 \leq N_0$ clearly holds. For $j > 0$, Invariant **I2** states that,

$$\begin{aligned} N'_i &\leq N_{i+1} \left\lfloor \frac{1}{a} N'_i \right\rfloor \\ &\leq N_{i+1} + \frac{1}{a} N'_i \ , \end{aligned}$$

which means Equation (4.1) is correct. We now define $W_i = \sum_{r_v \leq i} w_v$ and states that $W_i \geq a^i N'_i$. This is right by definition:

$$W_i \geq \sum_{j=0}^{i} a^j N_j \geq a^i \sum_{j=0}^{i} N_j \ . \tag{4.2}$$

Donate $R = \max(r_v)$, for all elements in $\mathcal{L}$, then it is clear that any element, with rank greater than R, must have been promoted to maintain the invariants. Invariant **I2** implies that $H(\mathcal{L}) \leq R + \log_a N'_R$ and thus the maximal depth for an element x in $\mathcal{L}$ is,

$$d_v \leq H(\mathcal{L}) - r_v \leq R + \log_a N'_R - r_i \ . \tag{4.3}$$

Equation (4.3) combined with (4.2) shows that $\log_a N'_R \leq \log_a(W - R)$, hence $d_i \leq \log_a(W - r_v)$. Because $\log_a(w_v - 1) < r_v \leq \log_a w_v$ the **Depth Lemma** holds. ■

The cost for searching in biased skip lists for element v with weight w_v and elements v_- or v_+ can be bounded. To search for v, means finding the lowest element of the three i.e. the one with the smallest weight. On each level we visit at most b elements according to Invariant **I1**, therefore the search costs at most,

$$O\left(1 + b \log_a \frac{W + w_v}{\min(w_{v_-}, w_v, w_{v_+})}\right) .$$

Inserting key v_x into $\mathcal{L}$, starts by locating both elements v_- and v_+. After this we create a new node v_x, with height h_v. Element v_x are inserted between v_- and v_+, in the same manner as in probabilistic skip lists. Element v_x which are inserted, can now violate **I2** by breaking up $a \leq v_x < b$ consecutive elements on each level i, where $0 \leq i \leq h_v$. The Invariant **I2** can be corrected by demoting an element w to the left (respectively right) of v_x on levels $h_{v_-} \leq i \leq h_{v_+}$. This demoting of elements can then induce violating Invariant **I1**, thus there can now be $> b$ elements on levels $0 \leq i \leq h_v$. Demoting an element can combine two sets of consecutive elements and result in a *plateau* of at least $b + 1$ elements. Therefore it is necessary to promote element z, where z is the $\lfloor (b + 1)/2 \rfloor$'th element. This promoting should be done at levels $h_v \leq i$, until **I1** is restored on levels $i \leq h_v$.

We can now calculate the time complexity of an insert operation. Finding the element v_-, v_+, and v can according to the **Depth lemma** 4.2 be done in time:

$$O\left(1 + b \log_a \frac{W + w_v}{\min(w_{v_-}, w_v, w_{v_+})}\right) .$$

Restoring **I2** is done for levels i, where $\min(h_{v_-}, h_{v_+}) \leq h_v$ and take $O(b)$ work at each level. Upholding Invariant **I2** is done for each level $> \min(h_{v_-}, h_v, h_{v_+})$, and on each level costing $O(b)$, hence the running time for an insert is at most,

$$O\left(1 + b \log_a \frac{W + w_v}{\min(w_{v_-}, w_v, w_{v_+})}\right) .$$

Deleting an element v in $\mathcal{L}$ starts as in the probabilistic skip list, described in 4.1, by locating elements v_-, v and v_+, if v exists in $\mathcal{L}$. In the case where v does not exists in $\mathcal{L}$, the operation stops here. Otherwise removing the element and connecting v_- and v_+, for all levels $i < \min(h(v_-), h(v_+))$. Removing v can result

in a violation of Invariant **I1**, where two sets of consecutive elements are united, this can happen on levels $\min(h_{v_-}, h_{v_+}) \leq i \leq h(v)$. On each level, we promote an element y, which is the $\lfloor k/2 \rfloor$'th element on the $(b, 2b]$ plateau. Deleting v can of course decrease length of the plateau on level i to less than a, Invariant **I2**. In this case where we demoted the violating node y, to level i, it is done for levels $\min(h_{v_-}, h_{v_+}) \leq i \leq h(v)$. Each demotion can result in a violation of **I1**, which should be corrected in the same manner as described above.

The complexity for finding v, its neighbours v_- and v_+, removing v and relinking takes time equal to:

$$O\left(1 + b \log_a \frac{W + w_v}{\min(w_{v_-}, w_v, w_{v_+})}\right) .$$

Given $\min(h_{v_-}, h_{v_+}) \leq h_v$, then performing the correction of **I1** takes $O(b)$ time at each level, between $\min(h_{v_-}, h_{v_+})$ and h_v. Equally correcting **I2** takes time proportional to $O(b)$ at levels from h_v to $H(\mathcal{L})$. This adds up to an execution time equal to

$$O\left(1 + b \log_a \frac{W + w_v}{\min(w_{v_-}, w_v, w_{v_+})}\right) .$$

This concludes that all operation on the *(a,b)* biased skip lists is bounded by $O(\log W/w_i)$, which is the wanted access time for biased search structures.

5 Dynamic self adjusting skip lists

That which is static and repetitive is boring. That which is dynamic and random is confusing. In between lies art.
John Locke (1632 - 1704)

The *splay* tree described in Chapter 3, has a lot of nice properties, one of these is the dynamic optimality. This means that over a sequence of accesses $\mathbb{X} = x_1, x_2, \ldots, x_m$ the *splay* trees are using only $O(OPT(\mathbb{X}))$ time. Where $OPT(\mathbb{X})$ is the time used by an optimal offline algorithm. In other words accessing a sequence $\mathbb{X}$ in the *splay* tree is bounded by $O(\sum_{i=1}^{m} \lg t_i(x))$, where $t_i(x)$ is the number of distinct accesses since x was last accessed also called the working set property and is described in Section 1.2.1. This means that the amortised complexity for a single accesses are $O(\lg t_i(x))$.

In this chapter we give a complete description and analysis of the dynamic self adjusting skip list data structure by Bagchi et al. [BCDI07]. The analysis is amortised and shows that dynamic self adjusting skip lists also satisfy the working set property and in an amortised time can access an element in $O(b \lg t_i(v))$.

In Chapter 6 we give all details concerning the conducted experiments compared to the theoretical bounds of dynamic self adjusting skip lists and *splay* trees. Tough the constant hidden in the *Big-O* notation is much greater than in *splay* trees as we shall see in this section, we will compare actual running times of these two algorithms in different scenarios.

5.1 Data structure

This data structure is a deterministic skip list structure $\mathcal{L}$, with n elements. Each element have a max height $h = 2^{k+2} - 2$, where $k = 2 \cdot \lceil \lg \lg n \rceil$, so the height of $\mathcal{L}$ is bounded logarithmic by $O(\lg n)$. The elements are furthermore placed in to $2k$ layers, the layers are named,

$$\{l_1, l'_1, l_2, l'_2, \ldots, l_k, l'_k\} \ ,$$

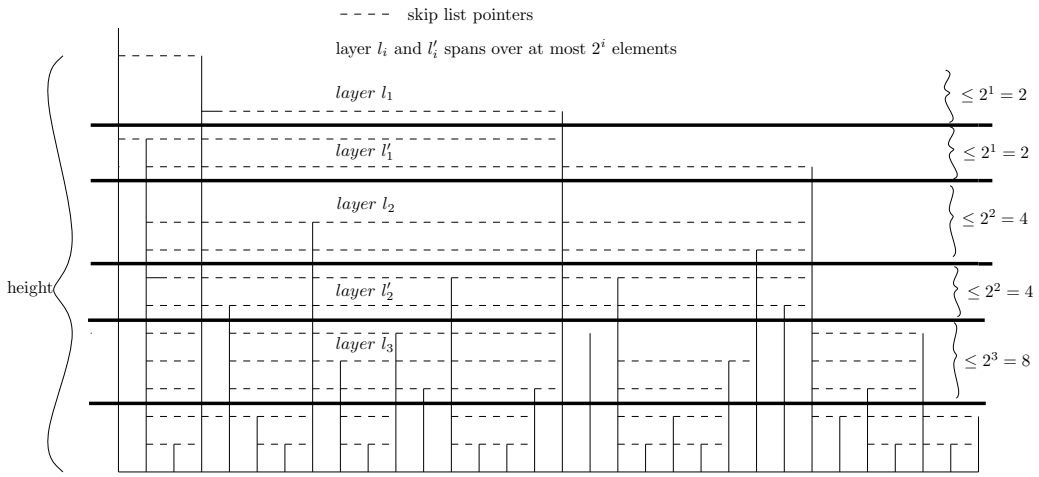


Figure 5.1: Terminology for the dynamic self adjusting skip lists.

where layers l_i and l'_i, each spans over the same maximum number of levels. Layers l_i and l'_i are spanning over 2^i levels. This means elements with a height within the range covered by l_i are associated with layer l_i. Each layer l_i and l'_i may furthermore not contain more elements than the 2^i levels can support.

The layers are ordered in decreasing order, such that the smallest layers are placed highest in $\mathcal{L}$, such that l_1 covers the highest 2^1 levels, l'_1 the next 2^1 levels and so forth. Elements are placed in l_1 and the next in l'_1 and so forth. The elements placed in the highest layers, are the elements accessed most recently i.e. with lowest working set number, since we move elements upwards to layer l_1, when accessing these. We will in Theorem 5.2 show that an element v at time i with working set number $t_i(v)$, where $B^{2^{j-1}} < t_i(v) \leq B^{2^j}$ in worst case reside no further down than layer l_j, v can of course also reside in a higher layer. This along with the access time shows that dynamic self adjusting skip lists satisfy the working set property.

A layer is further divided, by the invariants, **I1** and **I2** given below, into blocks of *consecutive* elements, each block contains between $[a, b]$ elements, where a, b are constants defined as $a = \lfloor b/2 \rfloor$ and b is given as input to the algorithm, either on compile time or before run time.

We are going to maintain the following two invariants, which are also maintained in the biased skip lists by Bagchi et al. [BBG02]:

I1) At a given layer i, there are no more than b *consecutive* elements with equal height, after an delete or overflow.

I2) For each elements x on layer i there are at least a elements of layer $i-1$ between x and any consecutive element on either side with the same layer i, after an delete or overflow.

These invariants have some consequences, which are explained in Section 4.3 for the structure of the dynamic self adjusting skip list and the movement of elements in this data structure.

In this data structure elements are being moved up and down according to their working set number and we will be keeping elements recently accessed high in the skip list to speed up access times to these elements. These two invariants sets some restraints to the way elements can be moved, when moving elements to layer l_1. An example is two elements placed next to each other, each accessed in order, then these two nodes can not both be at the top layer without breaking **I2**.

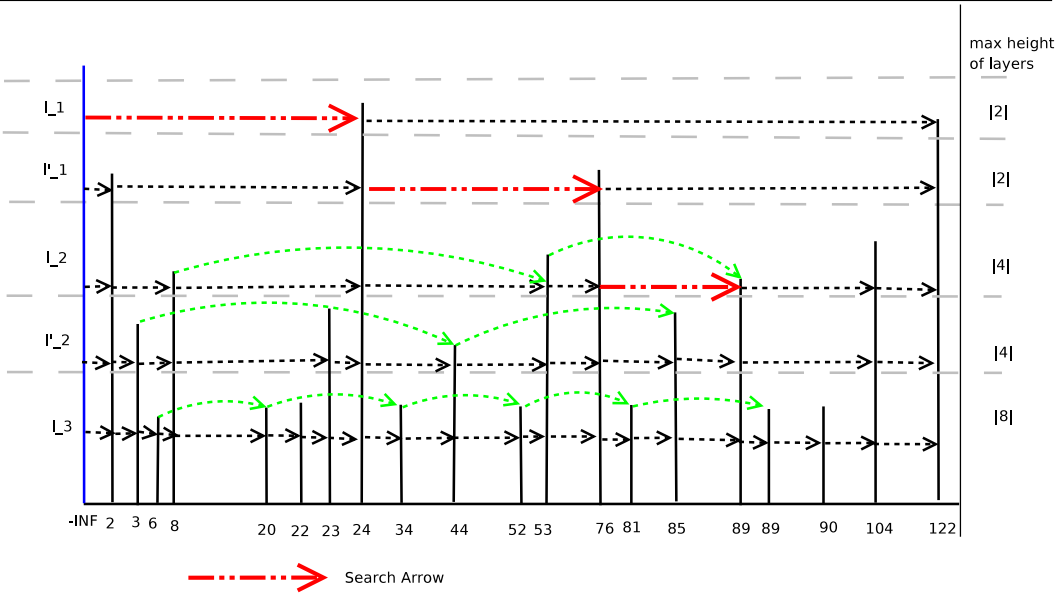


Figure 5.2: Search in dynamic self adjusting skip list for element 89.

An example of the dynamic self adjusting skip lists before and after a search, which result in a cascade of overflows, is seen in Figures 5.2 and 5.3.

5.1.1 Searching

Searching for key x in the dynamic self adjusting skip list $\mathcal{L}$, is done as in an ordinary skip list, we start from the top of the leftmost dummy element. We move either rightward to element v if for key k_v, $x < k_v$ holds, if instead $x > k_v$ we traverse the current element one step, see Figure 5.2, where we search for 89. This

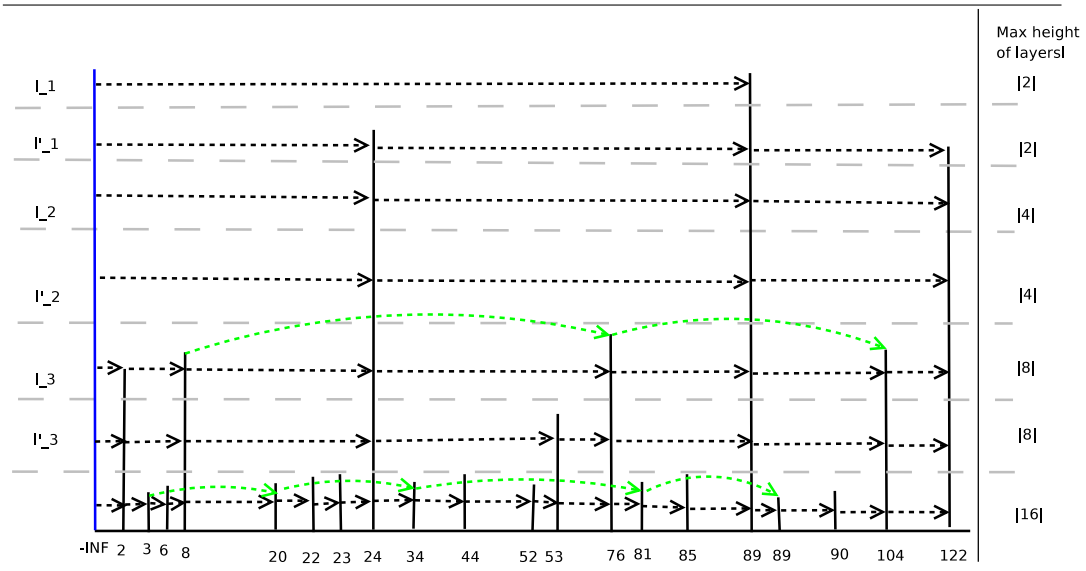


Figure 5.3: Moving *89* to top after search, results in overflow.

is done until either x is found in element v_x or we stand at the lowest level between two elements $\{v, w\}$, where $k_v < x$ and $x < k_w$.

If an element v_x is found on layer l_b, we promote that element otherwise we can choose to *promote* the nearest element[1], in key space, of elements the $\{v_-, v_+\}$. It is of course also possible to ignore these two elements, if v_x does not exist. The promotion is done by moving the element from layer l_b to l_1 if $b > 1$, see Section 5.1.2. As a result $\mathcal{L}$ could overflow in layer l_1 and cascade in lower layers, which should be handled described in Section 5.1.5.

5.1.2 Move element to top

Moving element v from its current position at layer l_i to level l_1 simply consist of dividing blocks on every level around v.

5.1.3 Insertion

Insertion of key x is done by searching for v_- in $\mathcal{L}$. After v_- we insert the new element v_x with height such that v_x is associated with layer l_1. This can also result in an overflow, again see section 5.1.5 for details.

[1] Introduced on Page 28.

5.1.4 Deletion

To delete key x from the skip list, we do a normal search for x in $\mathcal{L}$. If the key is located in element v_x, where $l(v_x)$ is the level v_x is placed on, this element is removed from the list. The pointers from the node v_i are updated on levels i, where $i \leq l(v_x)$, such that v_- points to v_+ on each level. When no elements points to v_x, we can safely delete it from $\mathcal{L}$.

After deleting an element, we need to make sure invariants are maintained. Both invariants can be compromised after deletion. If we now have to many consecutive elements, Invariant **I1**, we promote the middle element to break the consecutive elements, as described in Section 4.3.1. If instead Invariant **I2**are violated we can promote an element from a lower level, to increase the number of consecutive levels.

5.1.5 Overflow

An overflow in layer l_i occurs when the number of elements in layer l_i exceeds the number of elements the 2^i levels can support, which equals b^{2^i} elements. If an overflow occurs after inserting an element into layer l_i, we need to move elements from layers l_i and l'_i into the lower and larger layers.

The overflow procedure is initiated by merging elements in layer l'_i into layer l_{i+1}, which includes decreasing the height of each elements in l'_i with 2^i, so they now are associated with layer l_{i+1}.

Now that layer l'_i is empty we only need to move elements from layer l_i into the empty layer l'_i. Just decreasing the height by 2^i of each element in l_i, so they now are associated with layer l'_i.

See Figures 5.2 and 5.3 for example, where we after searching for element 89, Figure 5.2 and moves element 89 into layer l_1, this result in overflowing layer l_1, which after a cascade of overflows, where we merge layers, as described above Section 5.1.5 and result in the dynamic self adjusting skip list shown in the second figure.

This overflow will normally start as a result of an insert or search where the element v is inserted in layer l_1, thereby resulting in an overflow in layer l_1 in $\mathcal{L}$.

5.2 Analysis

Running times for the above data structures public interface such as *search*, *insert* and *delete* are all analysed in this section. In this section we are assuming that the size of $\mathcal{L}$ is always n. The constants b and a are given in advance and are defined in Section 5.1.

5.2.1 Moving an element to the top

Moving element v from its position in layer l_j to layer l_1, means increasing v's height so it is associated with layer l_1 instead of l_j. At the same time we need to split each block at level i around element v, where $j > i \geq 1$. It is now fairly simple to analyse the work needed for moving an element to the top of the list. The cost of splitting a block around element v is in worst case equal to finding element w, where key k_w is the first key where $k_w \geq k_v$, in $\mathcal{L}$ and dividing the block into two new blocks, which is equal at cost $O(b2^i)$. The worst case cost of moving an element from its position at layer l_j to layer l_1 is therefore $O(b\sum_{i=0}^{j} 2^i) = O(b2^j)$, this can result in a possibly overflow, which has to be paid by the access, see Section 5.2.3

5.2.2 Overflow

An overflow in layer l_i happens when the size of the layer exceeds b^{2^i} elements. The procedure is described above in Section 5.1.5. The worst case time complexity for merging two layers l'_i and l_{i+1} should be linear in the number of elements in l'_i. This is because each element v, when moved downwards, should use at most $b/h(v)$ of its potential to pay for each overflow operation, here $h(v)$ is the number over times element v can be involved in an overflow operation, i.e the number of layers below element v. The time to lower elements in layer l_i or l'_i are equal to $O(|l_i|) = O(|l'_i|)$. This is also the worst case time for an overflow operation in layer l_i, this can of course cascade into layers $j > i$. The worst case cost is $O(n)$ if every layer and thereby every element are affected.

5.2.3 Access time

To analyse the access time of the dynamic self adjusting skip lists we use potential functions. We give each element in the list a potential $\Phi(\mathcal{L}) = \sum_{\forall v \in \mathcal{L}} b \cdot h(v)$. Now we can analyse the access time.

Lemma 5.1. *Searching for an element v which are located on layer l_i in the dynamic self adjusting skip list $\mathcal{L}$ costs $O(b2^i)$ in the worst case.*

Proof. A search can be viewed as $2i$ sub searches in the layers $l_1, l'_1, l_2, l'_2, \ldots, l_i, l'_i$. A search in layer l_k, where $k \leq i$ only involves element associated with layer l_k, which corresponds to searching in a traditional skip list with 2^k levels, The search therefore costs $O(b\sum_{k=0}^{i} 2^k) = O(b2^i)$. This also holds when searching for an element associated with layer l'_i, because

$$O(b\sum_{k=0}^{i}(2^k) + b2^i) = O(b2^i + b2^i) = O(b2^i) \ .$$

An access operation in dynamic self adjusting skip lists, is followed by elevating the element from its position in layer l_i to layer l_1. The work done for moving an element to layer l_1, is equal to update pointers on every layer from l_i to l_1. In each layer l_k or l'_k, we at most need to update 2^k pointers. Therefore wee need to update at most $O(\sum_{k=0}^{i} 2^k) = O(2^i)$ pointers.

The dominating factor is finding v. This sums to a worst case cost for accessing and rising v at

$$O(2^i) + O(b2^i) = O(2^i) \ ,$$

for each access. In the above analysis we need do not count the cost of overflows, therefore we need the potential function defined above. ■

When accessing a sequence of elements in a dynamic self adjusting skip lists the top layer will eventually grow to big, which results in an overflow. In a long sequence of accesses, cascades of overflows can occur. The cost of an access to an element must incorporate the access cost. Therefore we assign a potential $\phi(v) = b \cdot h(v)$ as mentioned above. Now each element v has enough potential to pay for maximum $h(v)$ overflow operations, where each overflow operation decreases the elements potential by a constant b. This can be paid for by only adding a constant factor to each access (*insert* or *delete*).

Theorem 5.2. *Accessing a sequence of length m in the dynamic self adjusting skip lists can be performed in $O(\sum_{j=0}^{m} b \cdot \lg t_i(v))$ time.*

Proof. We show that searching for an element with working set number $t_i(v)$, where $b^{2^{k-1}} < t_i(v) \leq b^{2^k}$ is costing at most $O(b \cdot 2^k)$. Overflowing an empty layer l_k needs at least b^{2^k} accesses, this means that an element v, originally associated with layer l_1 cannot be moved further down than layer l_k after b^{2^k} accesses. Therefore an access to an element v with working set number $t_i(v)$ is costing $O(b2^k)$, which are the same as it costs to rise v from layer l_k to layer l_l. The cost of accessing m elements is therefore

$$O\left(\sum_{i=0}^{m} b \cdot \lg t_i(v)\right) ,$$

where $t_i(v)$ is the working set number for v at time i.

This proves the amortised access time for searches. ■

5.2.4 Deletes and inserts

The amortised cost for *searches* is also valid for *inserts*, where searching and possibles overflows are the dominating factors. We insert the element v at the bottom layer after element v_- and move it to the top from there, searching for v_- at the lowest layer costs in the worst case $O(b \lg n)$. This also means that the amortised time for insertions are $O(b \lg n)$, because we need to pay for the potential, which is equal to the number of levels (respectively the height) namely $O(b \lg n)$. The worst case is clearly $O(n)$ because at any one insert we can overflow layer l_1, which can cascade into the lowest layer and then involves all n elements.

A *delete* operation also needs to locate the element v and remove pointers from all elements v_- with height $h(v) \geq i \geq \lg \lg n$. The *delete* operation therefore also pays for removing pointers on h_v levels, this adds up to

$$O\left(b \lg t_i(v) + b \cdot h_v\right) = O\left(b \lg n\right) ,$$

because $O(\lg t_i(v)) + O(h_v) = O(\lg n)$.

5.2.5 Space consumption

The dynamic self adjusting skip lists described above uses besides the space also needed by the biased skip lists in Section 4.3, also memory for keeping track of $\lceil \lg \lg n \rceil$ layers.

The space needed for skip list structures and thereby the pointers between elements are $O(n)$, when removing adjacent pointers just as the biased skip list.

For keeping track of $2\lceil \lg \lg n \rceil$ layers, the dynamic self adjusting skip lists uses space $O(\lg \lg n)$. Layers l_i and l'_i contains pointers to $2 \cdot 2^i$ elements. This adds up to $O(n)$, which means that the dynamic self adjusting skip lists are space efficient.

5.3 Implementation

In this section we describe the implementation we are using in Chapter 6, when comparing the dynamic self adjusting skip lists with *splay* trees. Like in the *splay* trees implementation the dynamic self adjusting skip lists implementation can be found on the URL: `http://www.cs.au.dk/~henrik/thesis/`, all references are with this directory.

The implementation of the dynamic self adjusting skip lists can be found in `/src/DSkipList/`. The details explained in this section concerns deviations from the above description of the data structure. We also explain implementation details, which have been neglected above, cause they do not have any influence on the analysis.

The main implementation is lying in `DSkipList.cpp` where the following functions ***search_in(integer k)***, ***moveToTop(element v)***, ***overflow(layer*** l_a***)*** and ***mergeLayer(layer*** l_a***)*** can be found. These are also described in details in the following sections.

The main functionality of the dynamic self adjusting skip lists lies in these functions.

5.3.1 *search_in(integer* k*)*

The main function of this method is to locate the element v_k. This is done as described in Section 5.1.1. The search starts at the top of the leftmost element, which is always as high as the dynamic self adjusting skip lists. From here we move right and down according to the next elements' key at the given layer.

When searching for k, we also save the closest element v_- on all layers $a \geq i \geq 1$, so these are easily accessible later, when we move the element to the top.

5.3.2 *moveToTop(element* v*)*

Moving an element v from its placement at layer k to layer 1. Consists of three things:

1. Moving v to the top layer and insert it into the top layer l_1.
2. Inserting it into the skip list structure on level $k \geq i \geq 1$, so that v_- for each level i points to v and v points to v_+.

3. Correcting invariants **L1** and **L2**, by adjusting the neighbourhood around v on layers $a \geq i \geq 1$.

Correction invariants are only done when it make sense, i.e. not when two succeeding elements are lifted two l_1.

When locating the element v we save each pointer as described in Section 5.3.1. This means we in constant time for each level can update pointers from levels $k \geq i \geq 1$ can update pointers around v. There are as described above maximum $O(\log_a n)$ level, so a *moveToTop* operation costs $O(b \cdot k)$.

5.3.3 *overflow(layer l_i)*

When moving elements to layer l_i, this can trigger an overflow, if layer l_i now contains more than 2^i elements. Detecting whether or not layer i is overflowing is in this implementation done in constant time. This comes with a price namely $2 \cdot \lceil \lg \lg n \rceil$ units of extra space consumption. We maintain the size of each layer in a array, each time we either delete (respectively insert) into this layer we decrease (respectively increase) the number of elements at that layer.

An overflow at layer l_i is otherwise done according to the description in Section 5.1.5.

5.3.4 *mergeLayer(layer l_i)*

Merging layer l_i into l'_i is done after we have emptied layer l'_i. If l'_i is empty, moving l_i into l'_i is empty and is done time $O(|l_i|)$. The time comes from updating pointers on layer l_i so they correspond the height of l'_i. Before doing this, we make sure no element on layer $l_1, l'_1, \ldots, l_{i-1}$ are pointing to a element on layer l_i. The size of layers $l_1, l'_1, \ldots, l_{i-1} \leq O(2^i)$, therefore we have time to make sure no pointers at a level above layer l_i points to a wrong element.

In the case where l'_i is full we merge layer l'_i into layers l_{i+1} by making a sweep over both layers in parallel. Every time we move an element from layers l'_i into l_{i+1}, we change the height of that element. This gives a time complexity corresponding to $O(|l_i|)$, since the maximum difference in size of these two layers are,

$$2 \cdot |l_i| = 2 \cdot |l'_i| = O(|l_{i+1}|) \ .$$

These two cases each have an upper bound on the time complexity of $O(|l_i|)$.

5.3.5 *DBlock* class

In this implementation the dynamic self adjusting skip list, we use a class called *DBlock*. This class is used as a container class for the elements inserted into our structure. Each *DBlock* contains no more than b elements.

The function of this container class is to make sure, that keeping invariants **I1** and **I2** are easy. This of course comes with a price and in this case the price is the consumption of extra space. Each *DBlock* maintains a list of size $b + c = O(b)$. All functions of the *DBlock* has a complexity there are bounded by the constant b.

The implementation of the *DBlock* class can be found at `/src/DSkipList/DLayer.cpp` and the header file in `/src/DSkipList/DLayer.hpp`.

5.3.6 *DSkipListNode* class

Lastly we are describing the *DSkipListNode* class. This class is responsible for containing the data and for keeping pointers to the next elements, furthermore it holds a pointer to the *DBlock* where it is residing.

The size of a *DSkipListNode* is bounded by the height of the element v, which is equal $h(v)$. The implementation can be found at `/src/DSkipList/DSkipListNode.cpp` and the header `/src/DSkipList/DSkipListNode.hpp`.

mergeLayer DBlock cpp moveToTop DSkipListNode

6 Testing dynamic dictionaries

God does not care about our mathematical difficulties. He integrates empirically.
Albert Einstein (1879 - 1955)

In this chapter we presents the test setup and the results of the conducted tests. We compare the *splay* trees, described in Chapter 3, against the dynamic self adjusting skip lists, described in Chapter 5, in multiple scenarios. Section 6.1 describes the test procedure as well as the different scenarios we are using and in Section 6.2 we are presenting both results of testing dynamic self adjusting skip lists and *splay* trees alone and compared to each other.

Since we have performed many experiments, we present some result in Appendix A. In Appendix B we describes the physical test environment and the software used for these tests.

6.1 Test scenario

In this section we are presenting the empirical test scenarios, where we are going to test *splay* trees against dynamic self adjusting skip lists.

We are in some of the scenarios below testing using random material. We are using random numbers, which are put at disposal by www.random.org[1], the numbers are a collection truly random numbers collected from atmospheric noise. Furthermore these numbers are uniformly distributed.

We are going to test these two implementations against each other, by using different access sequences. These sequences have been mentioned throughout this thesis, as sequences where adaptive algorithms should behaviour better than normal search trees.

[1] http://www.random.org

The Sequences we are testing are:

$$1, 2, 1, 2, 1, 2, \ldots, 1, 2 \tag{6.1}$$
$$1, 2, \ldots, n, 1, 2, \ldots, n, 1, 2, \ldots \tag{6.2}$$
$$1, n, 1, n, 1, n, \ldots \ . \tag{6.3}$$

Besides testing Sequences (6.1), (6.2), (6.3), we are also testing how dynamic self adjusting skip lists and *splay* trees behaves at random accesses.

All test are conducted by first inserting n elements

$$1, 2, 3, \ldots, n-1, n \ . \tag{6.4}$$

After this we start the m length search sequences. In the random cases, we are also inserting the n numbers in a random order to get more balanced structures, which behaves more like real use situations.

The inserts are in the random case succeeded by searching for the m elements in uniformly distributed order. We have two different random access scenarios, 1) with unique random numbers and 2) with repetitions of numbers in the start structure.

We also testing both algorithm with the unified sequence from Section 1.2.5. It is not shown that either *splay* trees of dynamic self adjusting skip lists satisfy this property, but we expect the dynamic finger and sequential access properties should give *splay* trees an advantages over the dynamic self adjusting skip lists when accessing Sequence (6.5)

$$1, \frac{n}{2}, 2, \frac{n}{2}+1, 3, \frac{n}{2}+2, 4, \ldots \ . \tag{6.5}$$

Now we have scenarios where we test accesses to elements with very low working set numbers. These access sequences may not be very natural, therefore we will also test access to elements with increasing large working set numbers. In these test we insert the numbers from 1 to n and searching for m numbers with increasing working set numbers x. We have chosen two different ways to pick out the number we search for.

Search for the elements in the following ways:

1. $\{1, 2, 3, \ldots, x-1, x, 1, 2, 3, \ldots, x-1, x, 1, \ldots\}$.

2. Pick a random number in the interval $[1, 2x]$.

We chose to pick a single random number within a range twice as big as in the sequential case, we are with probability $Pr[X \leq (x/2)] = 1/2$, getting at number with lower working set number than we wanted and $Pr[X > (x/2)] = 1/2$, getting an element with larger working set number. In average we will pick a number with working set number equal to $2x/2 = x$.

By inserting elements in increasing order Sequence (6.4) we get a worst case structure in both *splay* trees and the dynamic self adjusting skip lists, where elements are places in long chains and access to the smallest elements is slow.

Every graph below is a result of an average of three runs, this should remove peaks, which comes as result of other services or kernel processes getting *CPU* time.

Before presenting any comparisons between the two data structures. We presents tests of dynamic self adjusting skip lists and *splay* trees. First we are showing that the number of overflows or *mergeLayer* are linear in the number of accesses. We are also testing different values of b, to observe under which conditions the dynamic self adjusting skip lists acts best. These tests are presented in Section 6.2.2.

6.1.1 Expected results

As mentioned before the dynamic self adjusting skip lists, should be better at keeping locality than *splay* trees. On the other hand we have a very simple *splay* tree structure, which furthermore could be optimised in different ways, like for example only *splaying* half the distance from a node v's placement to the root. This would better work because elements, which are accessed often slowly will ooze towards the root, while elements, which are only accessed once will be faster by a factor two if accessed again, but not slow down access times for elements, which we have seen have a larger probability to get accessed soon. In this implementation of *splay* trees, we have not optimised the structure in any such way. The simpler *splay* trees has the same expected number of comparisons as dynamic self adjusting skip lists over an sequence of length m namely,

$$O\left(\sum_{i=0}^{m} \lg t_i(v)\right) .$$

The *splay* trees furthermore satisfies the dynamic finger and sequential access properties, described in Section 1.2.4. This makes *splay* trees performed better at such sequences as (6.2) and (6.5), where the next element always are close, in key space, to an element which we have recently searched for.

In the *Big-O* notation the dynamic self adjusting skip lists are hiding a great constant. The dynamic self adjusting skip lists are also using more memory than *splay* trees, where both contains n elements, thus this is only a constant factor worse, the time used to allocate this memory is part of the constant that is hidden in the *Big-O* notation. Another example where the constant not noticeable, because of the *Big-O* notation, are when merging two layers. When merging these two layers, we are sweeping the layer three times to keep the invariants, described in Section 5.1, this is also hidden in the *Big-O* notation.

When increasing the working set number gradually, we expect that, when accessing increasing numbers, the *splay* trees is faster than dynamic self adjusting skip lists, because of the dynamic finger property. When accessing elements with a expected working set number equal to x randomly, *splay* trees, do not gain from the dynamic finger or sequential properties. This means that the difference we see in access time, is only due differences in preserving locality and constant factors hidden in the *Big-O* notation. When accessing the numbers in random order, the only difference between the two algorithms are the hidden constant in the *Big-O* notation and the locality preserving advantages by the dynamic self adjusting skip lists.

From this discussion our expectations are that *splay* trees are just as good or better by a constant factor than dynamic self adjusting skip lists, this is also what we are going to see in the next section, where we present the results.

6.2 Results

In this section we are describing the results of the empirical tests when comparing the dynamic self adjusting skip lists and *splay* trees. In the graphs shown in this section and Appendix A, we are using *DSL* as acronym for dynamic self adjusting skip lists.

6.2.1 Correctness of dynamic self adjusting skip lists implementation

Before presenting the results of the actual test, we will argument that the dynamic self adjusting skip lists and the *splay* tree are implemented correct.

As we will see in the Figure 6.1 the number of merges is what we will expect, namely linear in the number of accesses. So the heavy operation, merging of two levels, l_i and l_{i+1}, is executed the correct number of times, within a constant factor.

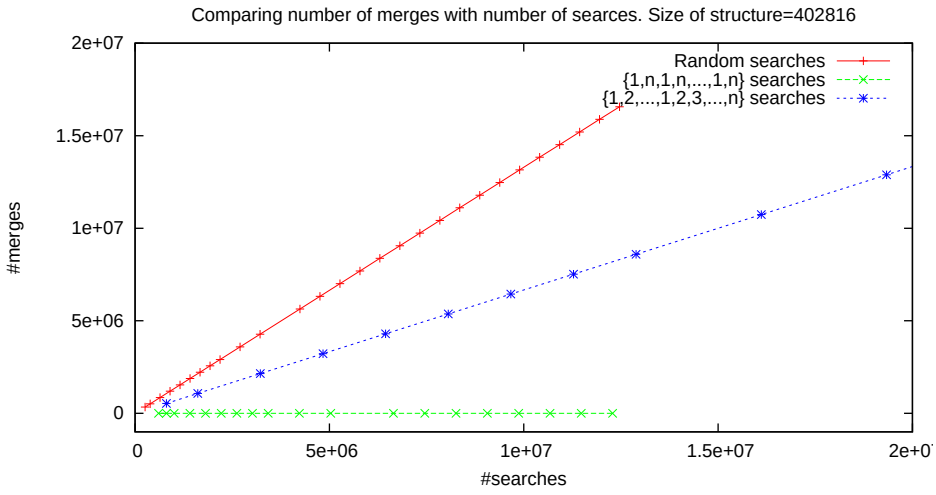


Figure 6.1: Comparing number of merges in stucutures of size 402816. The access sequence is random.

The time needed to merge two layers, is linear in the number of elements in layer l_i. We make tree sweeps over this layer when merging layer l_i into l_{i+1}. The execution time for this operation is always $O(|l_i|)$.

Searching for an element is conducted like in any other skip list structure and is fairly simple. Our structures has height bounded by $O(\lg n)$ levels, so searching compares at most $O(b \lg n)$ elements, this is the same bounds as for *splay* trees and as seen in Theorem 5.2 accesses can pay for overflows. Inserting or deleting an element v is done by removing v and changing the pointers on both elements (v_- and v_+) on either site of v on every level. This is constant amount of work on each level. Again at most $O(b \lg n)$ pointers.

Furthermore we have conducted a number of tests on dynamic self adjusting skip lists, to make sure each operation is returning the expected result.

We have run the following automated test scenarios, where we insert 5000 random numbers from the interval $[0, 2^{25}]$, searched for every element twice in random order, followed by a removal of all 5000 elements. An error in this scenario is when a *search* or *delete* operation did not return true, consequently when we did not find the expected element in the structure. This test was conducted 10.000 with different sequences of inserts, searches, and deletes without errors before proclaim that the structure was without errors.

Besides these automated tests for correctness, we have made it possible to visualise

smaller instances of a dynamic self adjusting skip lists and *splay* trees. This functionality have been used mainly to debug the data structures, but also to validate correctness of pointers in dynamic self adjusting skip lists and *splay* trees.

6.2.2 Testing the dynamic self adjusting skip list

We are here testing two things, first we are testing how many merges we are making as a function of the number of accesses, seen in Figure 6.1. We see number of merges as function of number of access in different scenarios like Sequences (6.1) to (6.3) and random accesses. We would expect a constant number of merges per access, for sequences of size m. This result in Figure 6.1 are as we would expect.

When comparing Sequence (6.2) and random access sequence, we see that the number of merges in the random case is greater, than when accessing the numbers in increasing order. This is because we are making a greater number of smaller merges in the top layers when accessing the numbers randomly, because we as described in the previous section, more often accesses elements with low working set number. When accessing the numbers in increasing order, we are moving the bottom element to the top, thereby merging all levels from levels $l_1 \geq i > l_{\text{max}}$ in one merge, because all levels are filled, due to the worst case insert sequence. After this cascade of overflows, we only see small merges for a long time. This makes searching slower, because all nodes are on the lower levels, which can be observed in Figure 6.3.

Now that we have shown that the implementation is merging the correct number of times, we tests different values of b. The reason for testing this, is to determine what values of b, we should use when testing dynamic self adjusting skip lists against *splay* trees in Section 6.2.3.

The intuition says we will see a difference, between the different values of b. When changing b we will allow more elements on any level i, between two consecutive elements on level $i + 1$. When using larger b we can end up comparing more elements at each level.

So from the above discussion and result on Figure 6.2, we choose to use smaller b's namely 2 and 4, when testing against *splay* trees, cause we in this way uses less memory and thereby hopefully gets more accurate tests.

6.2.3 Comparing self adjusting search trees

First we are going to look at some of the sequences we have talked about earlier namely Sequences (6.1) and (6.3).

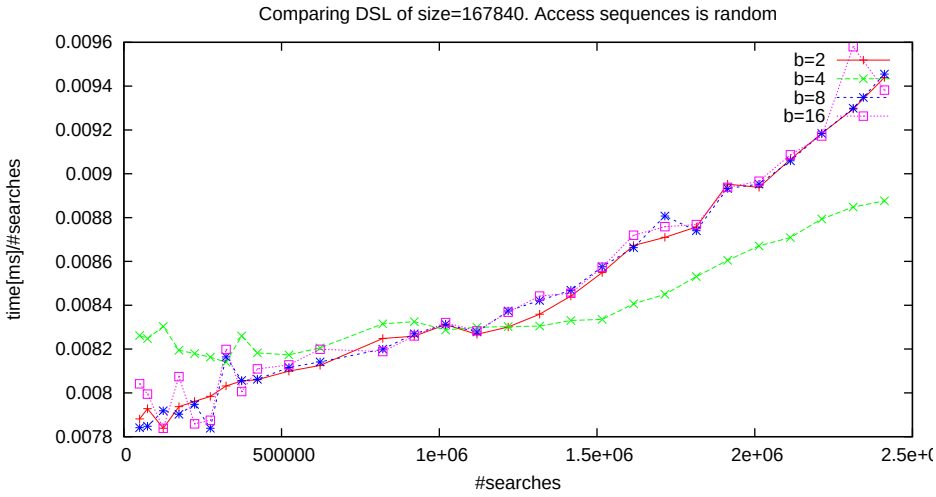


Figure 6.2: Testing different values for b.

6.2.4 Low working set numbers

There is not much to say about the results of accessing elements with low working set number, as seen in Figures 6.3 and 6.4 as well as in Figures A.3 to A.8 in Appendix A. We expect in both data structures few comparisons, because both the elements we look for are close to the top after we have accessed both elements once, furthermore in the dynamic self adjusting skip lists we only see a constant number of merges, see Figure 6.1. The structures is approximately $3-15$ times faster per access, as in the random case, see Figure 6.13.

6.2.5 Increasing working set numbers

Accessing elements with increasing working set number is shown on Figures 6.5 and 6.6 and also in the appendix in Figures A.9 to A.11.

These tests are conducted as described above by inserting all elements and afterwards searching for elements in increasing order. For example searching for elements with working set number 4096 the access sequence will look like in sequence:

$$\{1, 2, 3, \ldots, 4096, 1, 2, 3, \ldots, 4096, 1, 2, 3 \ldots\} .$$

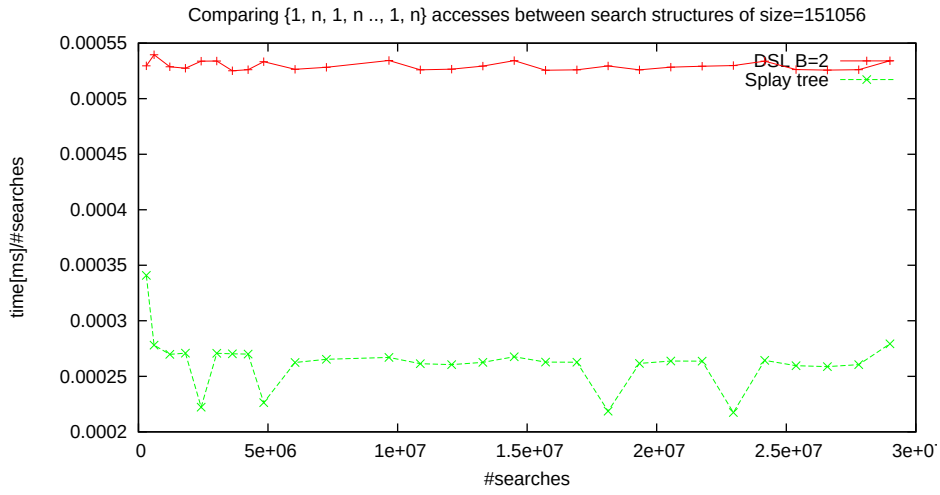


Figure 6.3: Searching for $\{1, 151056, 1, 151056, 1, 1, 151056, \ldots, 151056\}$ in structures containing elements $\{1, 2, 3, \ldots, 151056\}$.

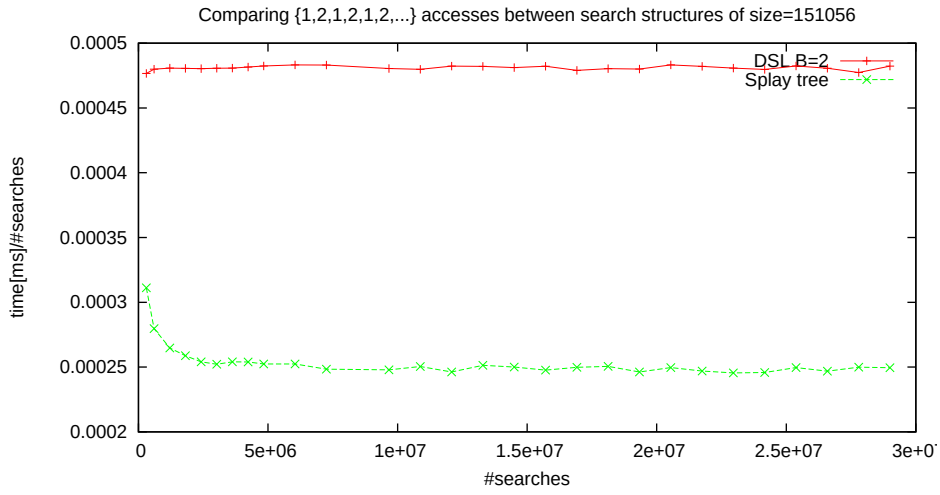


Figure 6.4: Searching for $\{1, 2, 1, 2, 1, 2, \ldots, 1, 2\}$ in structures containing elements $\{1, 2, 3, \ldots, 151056\}$.

Here we forces the working set number within a constant x, for exampled 4096, for each access, because we access x unique elements before again accessing the same element.

The result of these tests, where we gradually increases the working set number, we are searching for, are that for very low number of accesses the dynamic self adjusting skip lists are faster than *splay* trees. This is because of the first $O(n \lg n)$ drop in potential by the *splay* tree, when accessing a m length sequence. This drop is becoming less and less dominating the more accesses we search for. The faster *splay* trees are caused mostly by the sequential access property, which effects the execution time when the first x elements are found, and thereby making searching faster.

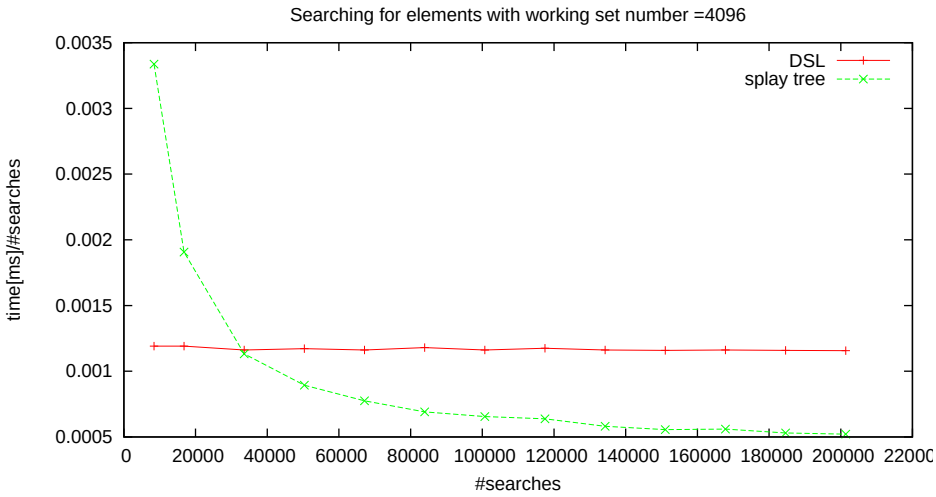


Figure 6.5: Searching for $\{1, 2, 3, \ldots, 4096, 1, 2, 3, \ldots, 4096, \ldots\}$ in structures.

When searching for larger working set numbers in random order, as it is the case in Figure 6.7, we see that the dynamic self adjusting skip lists and the *splay* tree have almost equal execution times. When accessing elements in a random order the sequential access property in the *splay* tree does not make accesses faster. Therefore the locality preserving in the dynamic self adjusting skip lists are equalising the execution time, which are seen in the almost equal access times.

6.2.6 Worst case access times

The worst case sequence for dynamic self adjusting skip lists is Sequence (6.2), where we at each access will go all the way to the bottom of the list to find the next elements. We are therefore seeing the longest access times. Empirical tests can be found on Figure 6.8 and in the appendix in Figures A.14 and A.15. The *splay* trees has a theoretical better bound for this sequence, which is equal to $O(m)$, because of the sequential access property.

Figure 6.6: Searching for $\{1, 2, 3, \ldots, 73728, 1, 2, 3, \ldots, 73728, \ldots\}$ in stuctures.

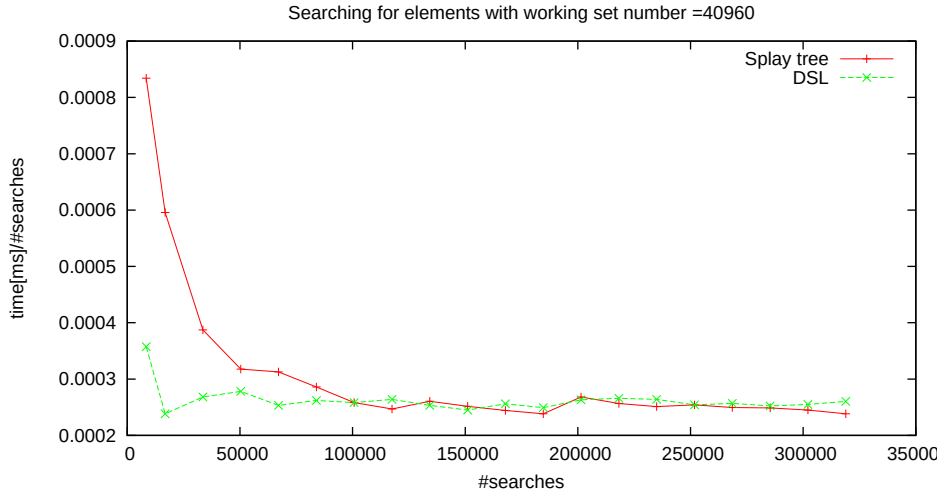


Figure 6.7: Searching for random elements within a working set number 40960.

In this sequence type the *splay* trees are starting with a long chain of elements, so when accessing the first element in Sequence (6.2) we find it at the bottom. When accessing the smallest element and splaying the element to the root, by a sequence of *zig-zig* steps, this halves the height of the tree as described in Section 3.1. This

indicates that accessing elements from here on out, is done a constant factor faster than dynamic self adjusting skip lists, which are not in the same way halving the next elements depth. Therefore we see a great difference between the *splay* trees and the dynamic self adjusting skip lists in this access sequence.

The dynamic self adjusting skip lists structure has an access time $O(\sum_{i=0}^{m} \lg t_i(v))$, which in this case is equal to $O(m \cdot n)$ cause $t_i(v) = n$, for each access. Therefore we see a more expensive average search costs, than for example in Figure 6.3.

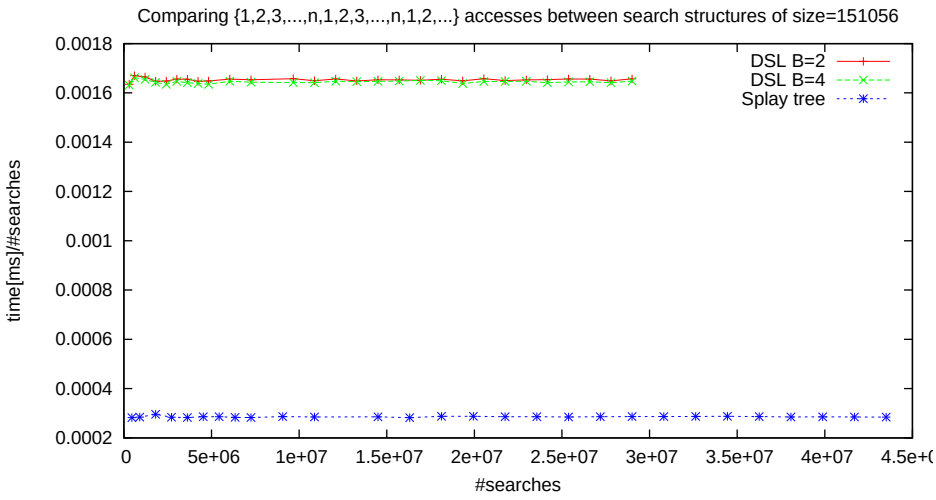


Figure 6.8: Searching for $\{1, 2, 3, \ldots, 151056, 1, 2, \ldots, 151056\}$ in structures containing elements $\{1, 2, 3, \ldots, 151056\}$.

6.2.7 Unified access sequences

Non of the two structures are satisfying the unified access property, described in Section 1.2.5. We see in both Figures 6.9 and 6.10 and also in the appendix on Figures A.16 and A.17, that *splay* tress are paying for the $O(n \lg n)$ drop in potential, which are distinct at lower number of access, but gain the overall advantages because of the dynamic finger search property.

The drop in potential is more expensive when increasing the size of the structures, a gradual comparison can be seen by comparing all four test, with increasing structure size. This can is seen in Figure 6.11.

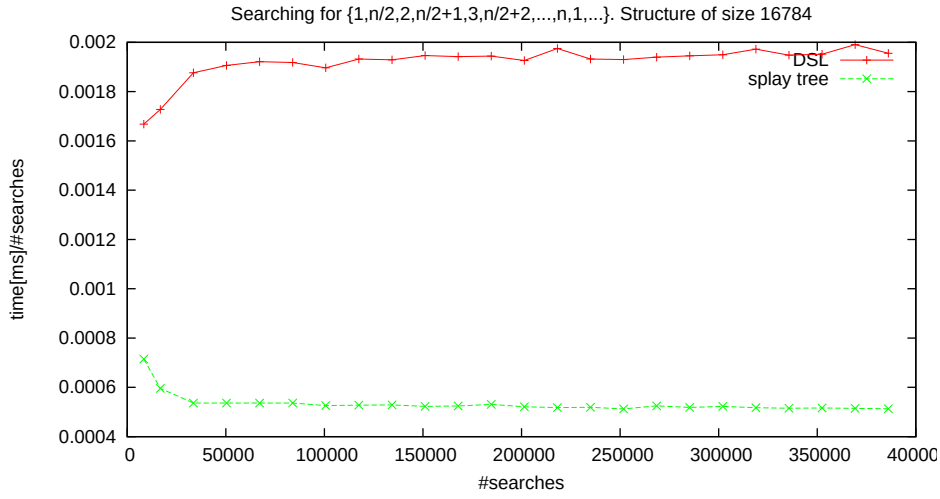


Figure 6.9: Searching for $\{1, 16784/2, 2, 16784/2 + 1, \ldots, 16784/2 - 1, 16784, 1, \ldots\}$ in structures containing elements $\{1, 2, 3, \ldots, 16784\}$.

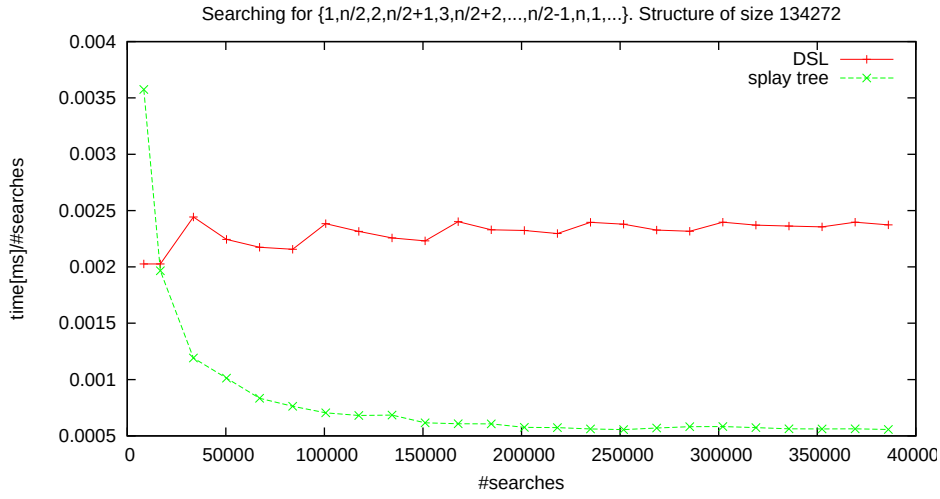


Figure 6.10: Searching for $\{1, 134272/2, 2, 134272/2 + 1, 3, 134272/2 + 2, \ldots, 134272/2 - 1, 134272, 1, \ldots\}$ in structures containing elements $\{1, 2, 3, \ldots, 134272\}$.

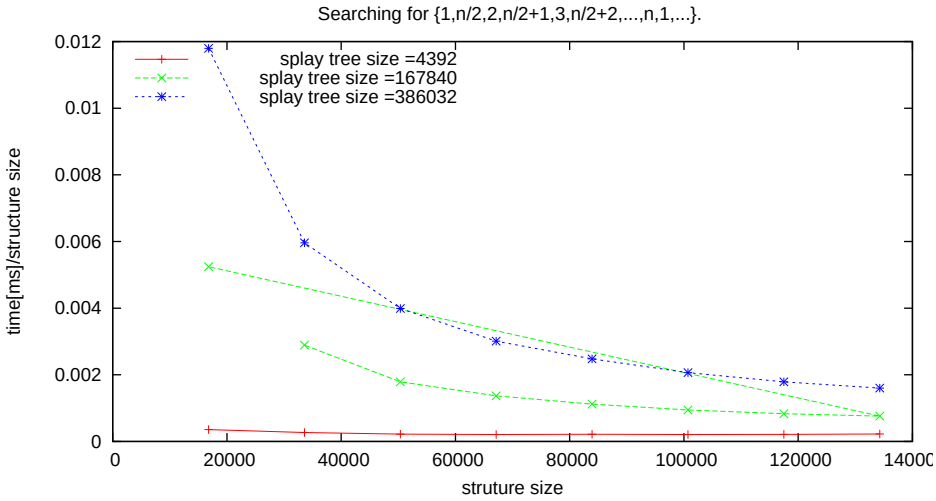


Figure 6.11: Variation of structure size, when searching for $\{1, n/2, 2, n/2 + 1, 3, n/2 + 2, \ldots, n/2 - 1, n, 1, \ldots\}$.

6.2.8 Random accesses

The last two access sequences are the obvious ones, where we search for m random numbers in a structure containing n elements. In Figure 6.12 the n elements are unique and in Figure 6.13 there can be repetitions of elements.

When accessing random elements, we expect that the average depth by which we find elements is $O(1)$. This indicates that accessing elements should be linear, because over a sequence of elements the average working set number is constant at about $n/2$, because of the probability explained above. This is seen because we in for example Figures 6.13 and 6.12 access twice as many elements as in Figure 6.8, where $t_i(v) = n$.

This is clearly seen in the figures, again it is clear that dynamic self adjusting skip lists is a constant factor slower than the *splay* trees, which corresponds to the other tests we have been running.

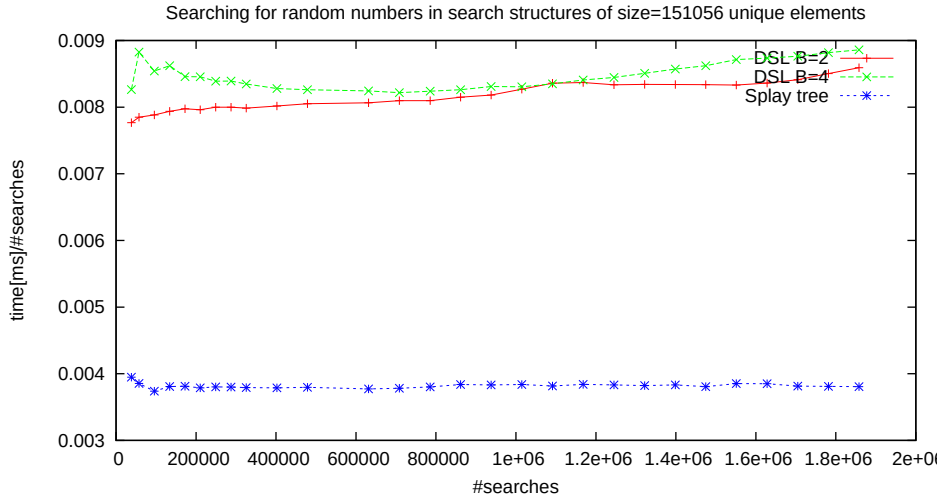


Figure 6.12: Searching for random numbers in stucutures containing 151056 elements unique elements.

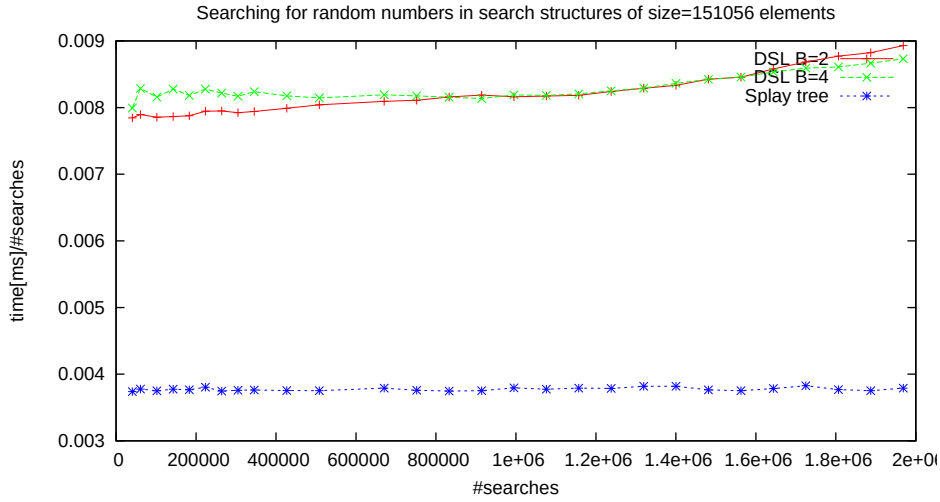


Figure 6.13: Searching for random numbers in stucutures containing 151056 elements.

7 The *I/O*-model

The sciences do not try to explain, they hardly even try to interpret, they mainly make models.
Johann Von Neumann (1903 - 1957)

Before we present internal and external *B*-trees in Chapter 8 and *I/O* effective self adjusting *B*-trees in Chapter 10, we will in this chapter presenting the *I/O*-model. We will here also describe how the analysis differs in from the *RAM* model and the *I/O*-model, when analysing an *I/O* algorithm.

The *I/O*-model, presented by A. Aggarwal et al. [AV88], consists of a two layered system, with a much slower "infinite" disk, *D* and faster main memory *M*, see Figure 7.1. The *I/O*-model models the vast difference in transferring speed between the main memory and the hard disk, which is found in modern computers. Transferring data from *D* to *M* is only possible in chunks of B units of consecutive data, note that *M* can only hold $|M|/B$ data chunks at any time. The way to bound an operations complexity in an algorithm, in the *I/O*-model is different from the *RAM*-model. We do not count the number of comparisons like in the *RAM*-model, but instead counts the number of block transfers back and forth, between disk and main memory.

7.1 Analysing in the *I/O*-model

Analysing an external algorithm are in the *I/O*-model, as mentioned above, different from the *RAM*-model. When analysing an *I/O* algorithm all work done internally in main memory and processor are considered free. The only work we count, when analysing an external algorithm, is the number of memory transfers back and forth, between the two levels.

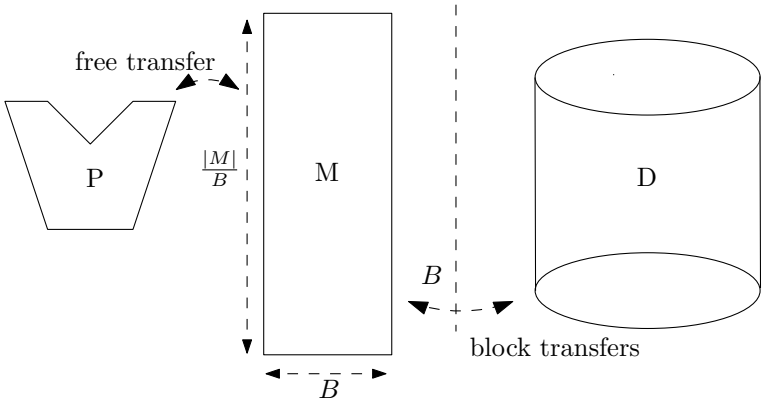


Figure 7.1: In the *I/O*-model it is only possible to transfer B consecutive elements from the "infinite" disk to the main memory and vice versa. The main memory can contain $|M|/B$ subsequent blocks. *P* is here the processor.

7.1.1 Mergesort

As an example we introduces and analysis the *mergesort* algorithm also presented in [AV88, p. 1123] paper, which is an *I/O* algorithm that sorts N unsorted elements in the optimal sorting bound in the *I/O*-model,

$$\text{sort}_{\text{external}}(N) = O\left(\frac{N}{B}\log_{\frac{M}{B}}\frac{N}{B}\right) .$$

We are in this thesis not proving the lower bound of sorting in the *I/O*-model, but simply describing and analysing the *mergesort* algorithm.

The *mergesort* algorithm starts by dividing the N input elements into $O(N/M)$ chunks, each of size M. In internal memory we sort each of these chunks by an internal algorithm like for example *Quicksort* [GTG01]. After sorting the $O(N/M)$ chunks we merge $O(M/B)$ blocks at a time until only one sorted chunk of size N is left.

An analysis of the external *mergesort* could progress like this; Dividing input data into the $O(N/M)$ blocks are done in *scan* time, namely $O(N/B)$ *I/O*'s. Sorting the chunks happens internally in memory and is therefore free. We can do this as we split the input data, so the cost is embedded in the above scan. Repeatedly merging the $\Theta(M/B)$ blocks, can be done $O(\log_{(M/B)}(N/M))$ phases, each costing $O(N/B)$ *I/O*'s, which adds up to $O((N/B)\log_{(M/B)}(N/B))$. This is clearly seen in Figure 7.2. Writing the sorted data back to disk costs $O(N/B)$ *I/O*'s. All this adds up to;

$$\begin{aligned}\text{sort}_{\text{external}}(N) &= O\left(\frac{N}{B}\right) \textit{I/O's} + O\left(\frac{N}{B}\log_{\frac{M}{B}}\frac{N}{B}\right) \textit{I/O's} + O\left(\frac{N}{B}\right) \textit{I/O's} \\ &= O\left(2\cdot\frac{N}{B} + \frac{N}{B}\log_{\frac{M}{B}}\frac{N}{B}\right) \textit{I/O's} \\ &= O\left(\frac{N}{B}\log_{\frac{M}{B}}\frac{N}{B}\right) \textit{I/O's}\ ,\end{aligned}$$

here dominating factor is the merging. This means that the complexity of sorting using the mergesort algorithm is $O(\frac{N}{B}\log_{\frac{M}{B}}\frac{N}{B})$ *I/O*'s in the *I/O*-model.

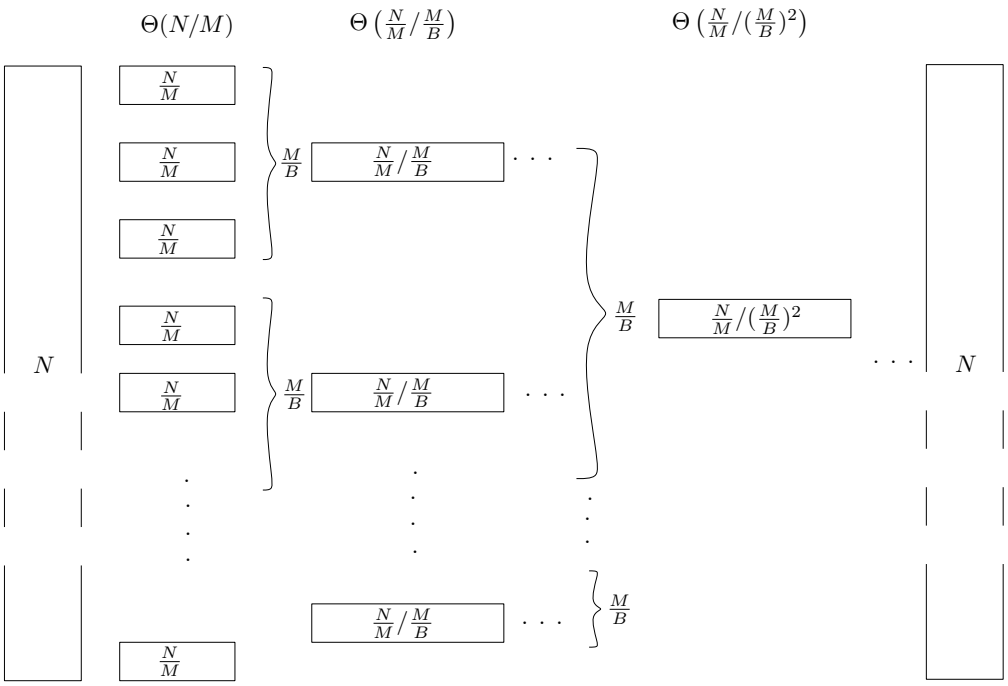


Figure 7.2: Mergesort algorithm visualised. Merge $O(N/M)$ sorted chunks in $O\left((N/B)\log_{(M/B)}(N/B)\right)$ merges.

In the above analysis it is clearly seen how an analysis of an external algorithm goes and how work done in internal memory is free.

7.2 Cache oblivious memory mode

When designing algorithms we normally call algorithms designed for the *I/O*-model *cache aware*, because these algorithms can be optimised for the specific computer based on parameter given on compile or run time. Another memory model namely the *cache oblivious model* was introduced by M. Frigo et al. [FLPR99] in

1999. A processor can in this model only directly fetch data residing in the closest cache. We define *cache-hits* as processor request to data which already resides in cache, and *cache-faults* as request to data which are not already in cache.

The *I/O*-model only models a two layered system without cache, but only memory and disk, whereas the *cache oblivious model* is a multi-layered system with r caches, named $1, 2, 3, \ldots, r$. The performance of a *cache oblivious model* algorithm is measured by its work complexity like in the *RAM*-model and also by the number of *cache-misses*

Historically good performance can obtained in the *I/O*-model, but the disadvantages are that external algorithms should be tuned for the machine they are executed on. Algorithm designed for *cache oblivious model* are normally divide and conquer algorithms, which works good on all machines.

We are here not giving any examples on *cache oblivious model* algorithms, since this is not the focus of this thesis.

8 *B*-trees

Memory feeds imagination.
Amy Tan (1952 -)

Before we start describing the dynamic *I/O* effecient *B*-trees in Chapter 10, which is deduced from the dynamic skip list, described in Chapter 5, we will describe a general method to transform skip lists to *B*-trees and back again in Chapter 9. This method was first described in [MPS92] by J. Munro et al. and later in [DJ07] by Brian C. Dean and Zachary H. Jones.

Before going into this dualism between *B*-trees and skip lists, we are introducing the internal and external *B*-trees structure. This structure is almost equal in both the internal and external memory model, the major difference is the analysis, we present the two analysis' in different sections; Internal *B*-trees are described in Section 8.2 and
external *B*-trees in Section 8.3. The differences in the data structure itself is pointed out, when we described the *B*-trees, throughout Section 8.1.

8.1 Internal and external *B*-trees

The *B*-trees are introduced in 1972 by R. Bayer and E. McCreight [BM72] and promise efficient search, even on large collections such as files.

B-trees with order k has five invariants it must satisfy;

1) Every node has at most k children.
2) Every node except for the root has at least $a = k/2$ nodes.
3) The root has at least 2 children, unless it is a leaf.
4) All leaves reside at the same level.
5) A node with i children contains $i - 1$ keys.

An example of a *B*-tree is shown in Figure 8.1. In this thesis we only consider *B*-trees where $k \geq 5$. In the case where $k = 4$ we have *(2-4)*-trees as described in Section 2.3.1. R. Bayer and E. McCreight [BM72] only considered trees, where k was odd.

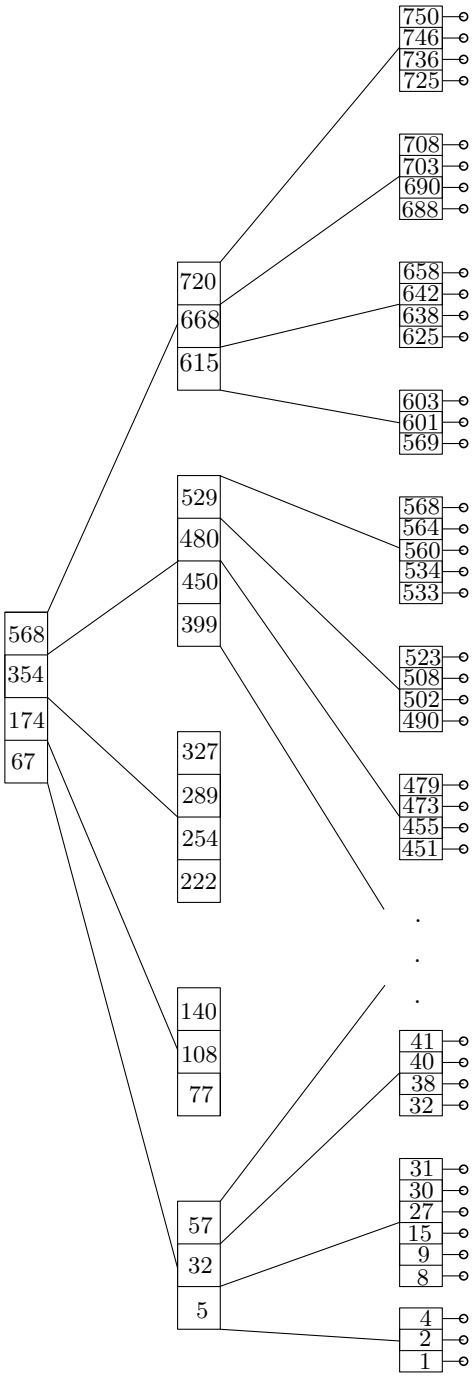


Figure 8.1: A *B*-tree with $k = 6$, which means each node has at least $a = 6/2 = 3$ children and maximum 6 children.

B-trees consist of nodes each containing $i = [(k/2) - 1, k - 1]$ keys and $i + 1$ children. These are arranged such that keys are placed in increasing order

$v_1 \leq v_2 \leq \ldots \leq v_i$. The pointers $p_1, p_2, \ldots, p_{i+1}$ are arranged such that p_i is placed between keys v_{i-1} and v_i. In the *I/O*-model each node are normally a file on disk and $k = B$, which is the block size.

8.1.1 Dictionary operations

When searching for value x in *B*-tree $\mathcal{T}$ of size n, we start at the root and in each node finding the first key k_i, at position i, where $x < k_i$. We then follow pointer p_i. The search terminates when we reach a leaf l and either find x or determine that x exists l and therefore in $\mathcal{T}$. Searching for x in a node v can be done linear, but if k becomes large this is not feasible in practise and searching in a node should instead be executed with a search scheme like for example binary searching, which is bounded by at most $O(\lg k)$ comparisons.

Inserting key x into $\mathcal{T}$ is done by first locating leaf l, which should contain x. In leaf l we find index i, where $k_i < x \leq k_{i+1}$ and insert x between these two keys. After inserting x into leaf l, the size of l can exceed size k and therefore create an overflow.

An overflow operation occurs when node v exceed size k and is solved by dividing v into two new nodes v' and v''. Node v are divided around some key $i = \lceil (k+1)/2 \rceil$. The nodes v' and v'' are inserted into parent p_v instead of v. Each overflow operation can potentially cascade in p_v, this should be handled recursively in $\mathcal{T}$. If root r is split, we insert a new root q instead of r. This increases the height of $\mathcal{T}$. In the *I/O*-model, dividing a node normally consists of writing two new files v' and v'' to disk. It is obvious that the total number of overflows in a single insert cannot exceed the height of $\mathcal{T}$, which is $O(\log_k n)$, because we only split nodes on path from v to r.

Deleting key x in $\mathcal{T}$ is initiated by locating leaf l, which contain x. If x exists we delete x from l. Now l is possibly smaller than $k/2$ keys, which means we should handle underflows.

In this case we should merge it with the succeeding or preceding leaf w into a new leaf q. The total size of these two nodes can exceed size k since,

$$|q| = |v| + |w| \geq \frac{k}{2} - 1 + \frac{k}{2} + 1 \geq k \ ,$$

where $|w|$ are bigger than $k/2$. Therefore we may split q in two nodes. This guaranties that each node are smaller than k. If dividing is not necessary, the size of parent of v is decreased by one.

Underflow in v can therefore cascade in the parent, in this case we recursively handle an underflow in each node from v to r.

This coarse description of *B*-trees gives an overview of the data structure, but leaves out how to optimise the data structure. Therefore we will present some obvious ideas to optimise the general *B*-trees structure.

8.1.2 Optimising *B*-trees structures

When implementing *B*-trees in practise there are some optimisation possibilities, not introduced above.

The first obvious optimisation is to store more than just a search key in the internal nodes. Actually we could store the keys in internals node, instead of placing these into a special key leaf. This would save space in an actual implementation, though only a constant amount.

In internal *B*-trees as well as in external, changing Invariant 2 [Knu98, page 488], such that a node cannot contain less than $(2k)/3$ keys, will give better access times, but slower inserts, because we will end up demanding more attention when inserting or deleting. Likewise we can let a node contain less than $k/2$ elements to make it split less often.

For external *B*-trees it is expensive with a root containing only 2 pointers, this is expensive because we need to read the root into memory at each access. Therefore we should keep the root node in memory at all time and give it more children. In the analysis in Section 8.3 this does not result in a tighter bound, but works better in practise. In internal and external *B*-trees we should always try to keep as many top level nodes in main memory or in the *level one* or *two* cache, to make accesses to these nodes faster.

8.2 Internal analysis

The analysis differs on key points in the internal and external memory model, therefore are we describing these analysis in two different sections. We are going to start with the internal analysis in this section. *B*-trees are promising fast accesses because we only touch few internal nodes, even on large collections. For example with $n = 1.999.998$ and $k = 199$ we only need to touch 3 nodes before hitting a leaf. Furthermore the analysis does not depend on amortisation, which makes *B*-trees ideal for *real time* applications.

We start the analysis by determining the height of a *B*-tree $\mathcal{T}$ with n keys and a branching factor k. All keys are placed in leafs. Each node v_i has $[k/2-1, k-1]$ keys (respectively $[k/2, k]$ children). The number of keys on levels $0, 1, 2, 3 \ldots$ is at least

$$2, 2\left\lceil \frac{k}{2} \right\rceil, 2 \cdot \left\lceil \frac{k+1}{2}^2 \right\rceil, \ldots$$

The $(n+1)^{th}$ key in tree $\mathcal{T}$ of size n appears on level l, hence

$$n+1 \geq 2\left\lceil \frac{k}{2} \right\rceil^{l-1} .$$

This means that the number of levels $h = h(\mathcal{T})$ is given as;

$$\begin{aligned} h(\mathcal{T}) &\leq 1 + \log_{\lceil \frac{k+1}{2} \rceil}\left(\frac{n+1}{2}\right) && (8.1)\\ &= O(\log_k n) && (8.2)\\ &= O(\log n) \ . && (8.3) \end{aligned}$$

Accessing a leaf is therefore equal to accessing $O(\lg n)$ internal nodes. In each node find the right child or key. If using binary searching at each internal node, this equals maximum $O(\lg k)$ comparisons. The dominating factor is the height of the tree and therefore an access cost bounded by,

$$O(\lg k \cdot \lg n) = O(\lg n) \ ,$$

comparisons.

An overflow (respectively underflow) operation can cascade on the path from a leaf to the root, which in worst case is equal to $O(\log n)$ overflows (respectively underflows). An overflow operation, divides a node into two new nodes, this costs at most $2k = O(k)$. The same is valid for underflow operations, where we are merging two nodes and maybe dividing one node of size $k < l < 2k$, this costs no more than $2k+2k = 4 \cdot k = O(k)$. This means a sequence of overflows (respectively underflows) costs at most $O(k \lg n)$.

Adding all this together means that a deletion or insertion composes searching plus possibly succeeded by sequences of overflows respectively underflows. The bound for *delete* or *insert* operations is therefore:

$$O\left(\lg k \cdot \log_k n\right) + 2 \cdot O\left(k \log_k n\right) \leq 3 \cdot O\left(k \lg n\right) \\ = O\left(k \lg n\right) \ .$$

This concludes the analysis internal *B*-trees in the *RAM*-model. The analysis is worst case, which makes *B*-trees ideal for usages in *real time* applications, where there is a demand of no single time consuming operations.

8.3 External analysis

The analysis in the *I/O*-model differs from the internal analysis, presented above. The analysis in *I/O*-models are different from the analysis in *RAM*-models, this has been described in Chapter 7.

The height argument given above in Equation (8.2) is the same for external *B*-trees. Let B be the size of a block, then each node contains at most B keys. As described in Chapter 7, we can transfer B consecutive keys back and forth, between disk and main memory. This implies that transferring a single node from disk to main memory costs a single *I/O* operation.

To search in external *B*-tree $\mathcal{T}$ of size n, we need to fetch $O(\log_B n)$ nodes from disk. Searching for key x in a node v, is done in internal memory and is therefore free. The total cost for searching is therefore bounded by $O(\log_B n)$ *I/O* transfers.

Merging and splitting two nodes residing on disk, corresponds to doing a constant number of reads and writes from disk. In the case where we split a node in two, each newly created node contains approximately $B/2$ keys. Moving two nodes into main memory costs two *I/O*'s. Merging the two nodes can be done in internal memory[1]. Writing the new node back cost one *I/O* and finally changing the pointers in the parent corresponds to another two *I/O*'s. The total cost is five *I/O*'s and the expensive part is therefore cascading. The total cost is therefore:

$$h(\mathcal{T}) \cdot 5 \textit{ I/O}'s = O(\log_B n) \cdot 5 \textit{ I/O}'s = O(\log_B n) \textit{ I/O}'s \ .$$

Equally an underflow takes a constant number of *I/O*'s. To read the two nodes into main memory costs two *I/O*'s. Merging and splitting is done in main memory, writing one or two nodes cost at most two *I/O*'s. Changing the parents pointers costs another two *I/O*. This bounds an underflow to

[1] We here assumes that $B^2 < |M|$.

$$h(\mathcal{T}) \cdot 6 \textit{ I/O's} = O(\log_B n) \textit{ I/O's} \cdot 6 \textit{ I/O's} = O(\log_B n) \textit{ I/O's} \ .$$

This means that the number of *I/O*'s for updates (*insert* and *delete*) in the *I/O*-model are equal to:

$$2 \cdot O(\log_B n) = O(\log_B n) \ .$$

As in the internal analysis, we have shown that searches and updates are bounded logarithmic in the number of keys in the *B*-tree just before the operation.

All three operations are worst case, which makes the *B*-tree data structure ideal for real time applications. In practise the external *B*-tree is furthermore superior to the *RAM* version, when the space consumed by the input data exceeds the space available in main memory. This property makes this data structure more attractive when data size is growing as we see in a many real life scenarios like physics experiments etc..

9 Translation between skip lists and *B*-trees

Now that we in Chapter 8 have introduced the *B*-trees data structure, which is fundamental data structure in many external algorithms and also in dynamic *I/O* effecient *B*-trees, which we introduce in Chapter 10. As the last thing before we present this structure we are presenting a method to translate skip lists to *B*-trees, this method was introduced by B. Dean and Z. Jones [DJ07]. The paper introduces a dualism between the two data structures and method to translated back and forth between these structures.

The general idea is that each level of a skip lists is translated into a level of nodes in *B*-trees. Thereby the height of the skip lists is also the height of the final *B*-trees. In the skip list all elements n_i, where $x_i \leq n_i \leq y_i$, lies between elements x_i and y_i on level $i = \min(h(x), h(y))$, are placed in *B*-trees as children to the node $\min(h(x), h(y))$

As an example we translate the skip list on Figure 4.1 [p. 38] to a *B*-tree, using the method described below. But before we presents the concrete example we explains the translation scheme.

As mentioned above, the translation from skip list $\mathcal{L}$ to *B*-tree $\mathcal{T}$ starts by creating a $n_{-\infty}$ element, this is the root of $\mathcal{T}$. The child of $n_{-\infty}$ is a element containing all elements from the highest level i_{top} in $\mathcal{L}$. Now recursively divide the existing elements on level $i-1$, between any two elements x and y on level i, where $0 \leq i \leq i_{\text{top}}$. This is implemented by taking all elements from level $i-1$, which lies between two elements on level i and make these into a tree node n_{i-1}. Now we need to make a pointer from node n_i, between the keys x and y, to node n_{i-1}.

Beside placing the nodes in the hierarchy described above we attach a weight to each edge in $\mathcal{T}$. The weight on the edge from $n_{-\infty}$ to the next node is equal the height of $\mathcal{L}$ and is the only edge having a positive weight. All other edges have a weight, which is equal to the difference in levels between the elements in $\mathcal{L}$ and is always negative.

An translation from the skip list visualised in Figure 4.1 [p. 38] to a *B*-tree can be seen at Figure 9.1.

The height of this newly created *B*-tree is most the same height as the original skip

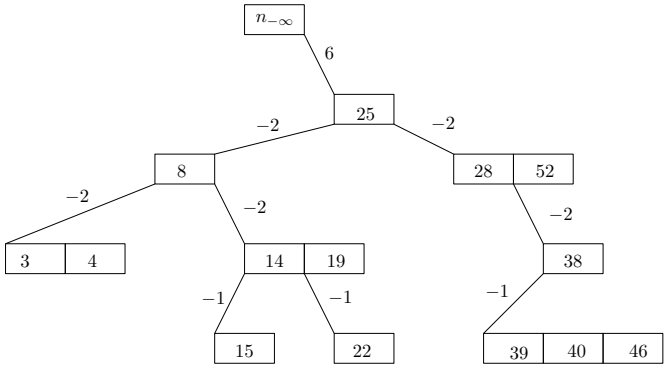


Figure 9.1: A translation from the skip list Figure 4.1 [p. 38] to a *B*-tree $\mathcal{T}$.

list plus one. As seen in Figure 9.1 and 4.1, this is an upper bound on the height. The expected height of the probabilistic skip list in Section 4.1 is both bounded by $O(\lg n)$. In general the height of a *B*-tree $\mathcal{T}$ generated from a skip list $\mathcal{L}$ is maximum $h(\mathcal{L}) + 1$.

Inserting key x into $\mathcal{T}$ is done almost as in an ordinary *B*-trees, see Section 8.1. When inserting x we find, by going downwards in $\mathcal{T}$, the right leaf v, where x is inserted. As an example we insert 27 into $\mathcal{T}$, which are showed in Figure 9.1. We insert 27 as leaf with 28 as parent. The node should have a total weight on the path from the root to v, which is equal to $w_v = h(\mathcal{T}) + \sum_i w_i - 1$, for all i on path from $root$ to v. In our concrete case this equals $6 + (-2) - 1 = -3$.

So node 27 get weight on the edge from 27 to 28, which equals -3. To determine v's final placement in $\mathcal{T}$ we roll a dice[1], just as in we would when inserting into a probabilistic skip list, see Section 4.1. As long as this dice returns a number smaller or equal to p, we subtracts one from the total weight of edge. When inserting 27 we roll our special dice until it rolls a number greater than $1/2$. Here we move v upwards in $\mathcal{T}$, rearranging the nodes on the path from the root to v. The concrete example results for example in two positive rolls, before the dice returns a number greater than $p = 1/2$. So we move the element 27 up two levels, which means we add 2 to the weight and therefore the edge between 28 and 27 equals -1.

Deleting an element x from $\mathcal{T}$, is done by removing the key from the node v and merging two nodes below if v is an internal node. If x is the only key in v we delete node v and merge the two children w_1 and w_2 into node w. The pointer from v's parent to v is changed so it points to w and we subtracts 1 from the weight on this edge. This corresponds to removing an higher element between to lower sets of consecutive elements in skip lists.

In the concrete example both of these operations is bounded in the height $O(\lg n)$. Searching costs at most $O(\lg n)$, the same as in the skip list described in Section 4.1.

[1] A dice which rolls a uniform distributed number in the interval $[0, 1]$.

Promoting an element is expected $O(\lg n)$ positives rolls, by the same argument as in Section 4.1, each step can be done at cost $O(1)$.

In general the height of generated *B*-trees are equal the height of skip lists used.

10 Dynamic I/O efficient *B*-trees

Collecting data is only the first step
toward wisdom, but sharing data is the
first step toward community.
Henry Louis Gates Jr (1950 -)

The dynamic *I/O* effecient *B*-trees data structures is described by M. Bŏdoiu et al. in [BCDI07], is a dynamic *I/O* effective data structure, which is adaptive, like the *splay* tress, described in Chapter 3 and dynamic self adjusting skip lists, described in Chapter 5. We are in this chapter going to describe the data structure, where all details concerning the data structure and the analysis will be presented and to some extent also details which deals with implementing it. In Section 10.2 we are going through the analysis, where we are presenting the working set property for this *I/O* algorithm.

Before going into the analysis of this data structure we will in Section 10.1 described how the operation on the dynamic self adjusting skip lists works.

10.1 Data structure

In this section we will describe what the dynamic *I/O* effecient *B*-trees data structure looks like and how operations are performed. We will also present some of the implementation details whenever these are not following directly from the description.

We recall from Section 8.3 that in the *I/O* effective *B*-trees data structures we define B, where B is equal the block size. Now we can define the two constants we are going to use in the dynamic *I/O* effecient *B*-tree namely $b = B$ and $a = \lceil B/2 \rceil$. The *B*-trees structure, described in Section 8.1, keeps every dictionary key in the leaves, however we are in here storing keys in the internal nodes, as suggested in Section 8.1.2.

Furthermore we maintain the following invariants:

1) Every node has at most B children.

2) Every node v_i except for the root has at least a nodes after an overflow or delete operation, which involves v_i.

3) All leaves reside at the same level.

4) A node with $k+1$ children contains k keys.

The dynamic *I/O* effecient *B*-trees therefore keeps between a and B keys in each internal node. A node with k keys is having $k+1$ children and of course a pointer to each of these. We have loosened Invariant 2) compared to the general *B*-trees in Section 8.1, so that node v can contain less than a keys, until an overflow or delete operation involving node v is performed. This means that $\mathcal{T}$ can contain nodes, which have fewer than a nodes.

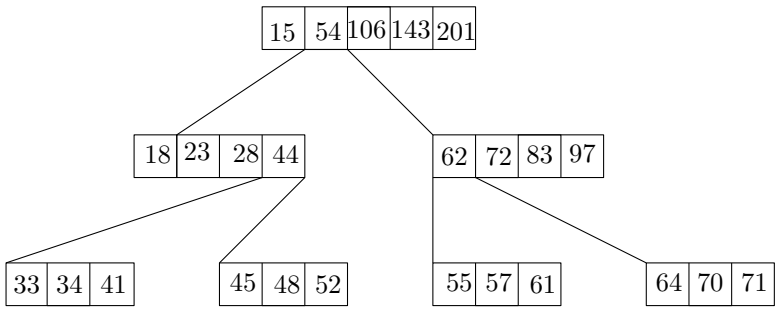


Figure 10.1: Dynamic *I/O* effective *B*-tree. Only necessary pointers are shown.

An illustration of a dynamic *I/O* effecient *B*-trees can be found in Figure 10.1, where the constant $b = B = 6$ and $a = 6/2 = 3$.

10.1.1 Searching

Searching for key x in a dynamic *I/O* effecient *B*-tree $\mathcal{T}$, is done like in ordinary *B*-trees, moving down into a nodes i^{th} child, if the keys $k_i < x < k_{i+1}$. If we reach a leaf and the key x does not exist in this then the search stops, because then x does not exist in $\mathcal{T}$.

If we find x in a node or leaf, we need to promote x to the root r. We are going to describe this operation in Section 10.1.4

10.1.2 Insertions

When inserting a key x in $\mathcal{T}$ we need to insert this into root node r, so that x afterwards is easily accessible. Inserting x into root node r is easy, but requires splitting children nodes, such that searching in $\mathcal{T}$ is still possible. The value x divides each node on a path from r to a leaf l in two around the value x. This means that in worst case we need $h(\mathcal{T})$ splits, which is bounded by $O(\log_B n)$ *I/O*'s.

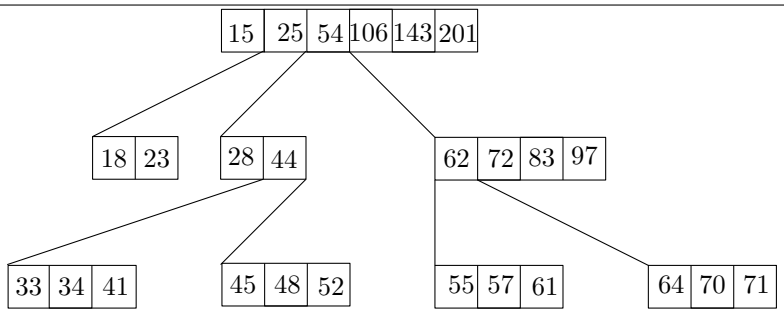


Figure 10.2: Dynamic *I/O* effective *B*-tree after inserting key 25. Only necessary pointers are shown.

As an example we have inserted 25 into the dynamic *I/O* effecient *B*-tree, shown in Figure 10.1, the resulting tree is shown in Figure 10.2. Here we can see that some of the nodes contains fewer than a keys after an insertion.

Inserting a key into r can of course also result in r becoming larger than B, which is handled by an *overflow* operation, see Section 10.1.5 for description.

10.1.3 Deletions

Deleting a key x can only be executed if x is present in the structure. So we start by searching for x. If x is in a leaf l, two things can happen:

1. The size of l becomes smaller than a, in this case we merge l with a succeeding or preceding leaf.
2. The size of l is still containing between a and B keys and deletion stops here.

Otherwise x is located in an internal node v, we first remove the key from v and then merge the children along the path from v to some leaf. All nodes merged, can of course now exceed the size b, in which case they should be slitted again to maintain Invariant 1).

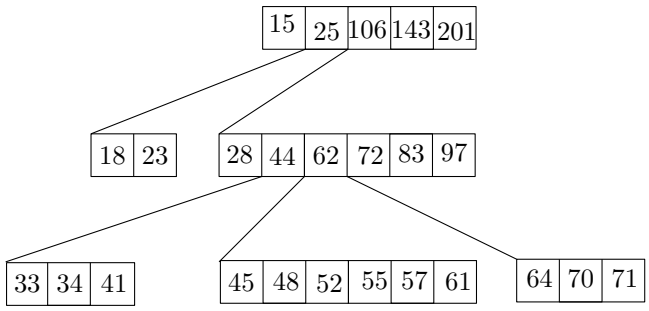


Figure 10.3: Dynamic *I/O* effective *B*-tree after deleting key 54. Only necessary pointers are shown.

The result of a *delete* operation is shown in Figure 10.3, where we have removed key 54 in the dynamic *I/O* effecient *B*-tree.

10.1.4 Promote

Promoting a key from its original position at node v to root node r in $\mathcal{T}$, is what makes this algorithm adaptive and satisfying the working set property, described in Section 1.2.1. Promotion is simply done as a deletion of x from v, here we assume that we already have a pointer to x, so we do not need to locate x first. After this we insert x back into $\mathcal{T}$ as described in Section 10.1.1. This means that a promotion can result in a *overflow*, if the size of r exceeds B keys after inserting x.

10.1.5 Overflow

The expensive operation in the dynamic *I/O* effecient *B*-trees is the *overflow* operation. The idea with this operation is to move keys with a high working set number[1] downwards in $\mathcal{T}$. The higher working set number, the "older" a key is.

If a node v is exceeding size B we need to let some of the keys seep downwards in to the children. We choose to move the $\lceil B/2 \rceil$ oldest keys downwards into the children. Moving a single key x down from node v to w, consist of deleting x from v and merging w and it predecessor (successor respectively). No we can insert x into w, which can result in w becoming to large, here we need to split w to maintain Invariant 1.

The overflow operation moves the $\lceil B/2 \rceil$ oldest keys in v downwards in $\mathcal{T}$. An overflow operation in node v can of course cascade into lower nodes, in which case this operation is becoming expensive in the worst case, the trick to make accesses

[1] A key with a high working set number, has not been accessed recently.

pay a part of this expensive operation, as we shall see in Section 10.2.5, when analysing this operation.

10.2 Analysis

In this section we are going to present an analysis of the dynamic *I/O* effecient *B*-tree $\mathcal{T}$, with n keys. Here we will weight the amortised analysis of accessing a sequence $\mathbb{X} = \{x_1, x_2, \ldots, x_m\}$ highest, but also include amortised analysis where it makes sense.

To analyse the dynamic *I/O* effecient *B*-trees, it is necessary to use potential functions, since we have at least one operation, namely *overflow*, described in Section 10.1.5, which can cascade and in worst case use very much time equal to $O(n)$. So let the height of an key x, donated $h(x)$, be the number of nodes we need to visit when searching for x in $\mathcal{T}$, this means that the depth of x equals $d(x) = h(\mathcal{T}) - h(x)$. Then we can define the potential function

$$\Phi(\mathcal{T}) = \sum_{i=0}^{n} c \cdot d(x_i) \ ,$$

for some constant c, if x is placed in r, then $d(x) = h(\mathcal{T})$. This means that each keys potential, can decrease a constant c for each overflow operation. Therefore it cannot cost more than a constant c when moving x downwards one level in $\mathcal{T}$.

The height of a dynamic *I/O* effecient *B*-tree $\mathcal{T}$ can be bounded to the size of $\mathcal{T}$. The constants a and B are defined above. On each level i we have at least $O(i \cdot a^i)$ keys. This gives an upper bound on the number of levels $O(\log_B n)$, as we have seen in previous analysis'.

But before going into further details, about how to pay for the *overflow* operations and showing the working set analysis, we are going to calculate what each operation on $\mathcal{T}$ of size n costs.

10.2.1 Search

Locating a key x in $\mathcal{T}$, is like searching in a normal external *B*-tree and the complexity is the same, namely $O(\log_B n)$ *I/O*'s. When we know where x is in $\mathcal{T}$, we need to promote this key to the root, which can result in an overflow. It is shown in Sections 10.2.4 and 10.2.5, that we can charge extra $O(\log_B n)$ *I/O*'s to

the search cost to pay for these operations. The total amortised cost is therefore bounded by $O(\log_B n)$ *I/O*'s.

This is an amortised analysis, which assume that x is in a leaf, we are in Section 10.2.6 showing that if we are accessing a sequence $\mathbb{X}$ of size m, then the cost is bounded by the working set property, described in Section 1.2.1, because keys not always resides in a leaf.

10.2.2 Insert

When inserting a key x in $\mathcal{T}$, we recall that we insert x in the root r and split nodes around the key x from r on a single path to a leaf.

We have bounded the height of $\mathcal{T}$, so the work needed on each level are splitting a node v_x around x, where $|v_x|$ is maximum B keys. This means we do $O(1)$ *I/O*'s on each level. This means we have an upper bound on the number of *I/O*'s for an insertion in $\mathcal{T}$, namely,

$$c \cdot \log_B n = O(\log_B n) \ ,$$

where c is some unspecified constant, which mainly depends on the implementation. An *insert* can, as described above, result in an *overflow*, this cost are incorporated into this cost. Each overflow could potential move x down one level, so to incorporate this cost, we charge extra $O(\log_B n)$ *I/O*'s to each *insert*, to pay for the $c \cdot h(\mathcal{T}) = c \cdot d(x)$ potential needed by x.

10.2.3 Deletion

Locating x in $\mathcal{T}$ costs, as shown in Section 10.2.1, at most $O(\log_B n)$ *I/O*'s. Deleting a key, means merging two and possibly splitting two nodes at each level below x. The maximum number of nodes we need to read from disk and write back is therefore,

$$O\left((2\ \textit{I/O}'s + 2\ \textit{I/O}'s) \cdot h(x)\right) \leq O\left(4\ \textit{I/O}'s\ \cdot \log_B n\right) = O\left(\log_B n\right)\ \textit{I/O}'s \ .$$

Then we can conclude that deleting key x is bounded by $O(\log_B n)$ *I/O*'s.

10.2.4 Promote

Suppose we have found key x in a node v_x in $\mathcal{T}$. Then we need to delete x. This costs according to Section 10.2.3 at most $O(\log_B n)$ *I/O*'s to merge and split nodes below v_x. Inserting key x afterwards in r cost according to Section 10.2.2 also $O(\log_B n)$ *I/O*'s.

Now we only need to charge x for the increase in potential when moving x up into r, which is equal to:

$$c \cdot (h(\mathcal{T}) - h(x)) \leq c \cdot h(\mathcal{T}) = O(\log_B n) \ .$$

So promoting a key x cost at most $O(\log_B n)$ *I/O*'s.

10.2.5 Overflow

When a node in $\mathcal{T}$ exceeds the size B, we need to move keys downwards in $\mathcal{T}$. When moving key x one level down, from node v to w, we already know, we do not have to divide nodes further down, because this is already done. Removing x from v can result in v becoming to small, in which case we should merge v with one of its neighbours. This cost at most 2 *I/O*$'s = O(1)$ *I/O*$'s$. Inserting x into w can result in an overflow in w, which can trigger a cascade of overflow operations.

The drop in potential corresponds to $O(1)$ for each level, key x is moved downwards. Therefore paying for an overflow operation can be done by adding a constant $O(1)$ *I/O*'s to each access operation.

10.2.6 Working set analysis

In this section we will use the above result to show that a sequence of m accesses on a dynamic *I/O* effecient *B*-tree $\mathcal{T}$ of size n is bounded by the working set property.

Theorem 10.1. *On a sequence $\mathbb{X}$ of m accesses, where $\mathbb{X} = \{x_1, x_2, x_3, \ldots, x_m\}$, the working set property bounds the number of* I/O*'s used to:*

$$O\left(\sum_{i=0}^{m} \log_b t_i(x_i)\right) \ ,$$

where $t_i(x_i)$ is the working set number for x at the i^{th} access.

Proof. Accessing a key on level j costs at most $O(j)$ *I/O*'s, because we visit j nodes. Visiting a node costs $O(1)$ *I/O*'s.

To fill all nodes in an empty level j in $\mathcal{T}$, needs at least B^j accesses. When accessing an key x with working set number $t_i(x)$, where $B^{j-1} < t_i(x) \leq B^j$, the search cost for this key is less than $O(j)$ *I/O*'s, which is equal to x residing in a node at level j.

This means the number of *I/O*'s we should use to find x on level j is,

$$O(\log_b B^j) = O(j) = O(\log_B t_i(x)) \textit{ I/O}'s \ .$$

Over a m length sequence these access costs

$$O\left(\sum_{i=0}^{m} \log_b t_i(x)\right) \textit{ I/O}'s \ ,$$

which concludes the proof.

■

This concludes the working set analysis of the dynamic *I/O* effecient *B*-tree.

11 Future work

Prediction is very difficult, especially
about the future.
Niels Bohr (1885 - 1962)

We have in this thesis covered the working set property in internal memory, with empirical tests as well as description and analysis of both *splay* trees and dynamic self adjusting skip lists.

We have in the external memory model also described and analysed the dynamic *I/O* effecient *B*-trees structure. We need to implement this structure to test it against internal memory algorithms below and around the memory level. Testing it above the memory level with for example external *B*-trees would also be interesting, to see how much of a speed up, we gain when dynamically optimise for skew access sequences.

12 Conclusion

Finally, in conclusion, let me say just this.
Peter Sellers (1925 - 1980)

We have in this thesis covered different adaptive data structure designed for faster accesses to skew access sequences. In particular have we described and test dynamic self adjusting skip lists and *splay* trees. We have shown that both these data structures have an equal amortised logarithmic bound for accessing elements by their working set number. The implementation and conducted tests shows that *splay* trees performs faster on sequences, where *splay* trees gains an advantages from either the sequential access property or the dynamic finger property. But when testing the two structures by a purely working set sequence we see an equal access time.

We have lastly shown how to transform the dynamic self adjusting skip lists into an external data structure dynamic *I/O* effecient *B*-trees, which only uses an logarithmic number of *I/O*'s for accessing element according to their working set number.

Bibliography

[AV88] Alok Aggarwal and Jeffrey S. Vitter. The input/output complexity of sorting and related problems. *Commun. ACM*, 31(9):**1116–1127**, 1988.

[BBG02] Amitabha Bagchi, Adam L. Buchsbaum, and Michael T. Goodrich. Biased skip lists. In *ISAAC '02: Proceedings of the 13th International Symposium on Algorithms and Computation*, pages **31–43**,44–48. Springer-Verlag, London, UK, 2002.

[BCDI07] Mihai Bŏdoiu, Richard Cole, Erik D. Demaine, and John Iacono. A unified access bound on comparison-based dynamic dictionaries. *Theor. Comput. Sci.*, 382(2):**86–96**, 2007.

[BDL08] Prosenjit Bose, Karim Douïeb, and Stefan Langerman. Dynamic optimality for skip lists and *B*-trees. In *SODA '08: Proceedings of the nineteenth annual ACM-SIAM symposium on Discrete algorithms*, pages **1106–1114**. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2008.

[BM72] Rudolf Bayer and Edward M. McCreight. Organization and maintenance of large ordered indexes. *Acta Informatica*, 1(2):**173–189**, 1972.

[BST85] Samuel W. Bent, Daniel D. Sleator, and Robert E. Tarjan. Biased search trees. *SIAM J. Comput.*, 14(3):**545–568**, 1985.

[BT78] Mark R. Brown and Robert E. Tarjan. Design and analysis of a data structure for representing sorted lists. Technical report, Stanford, CA, USA, 1978.

[Col00] Richard Cole. On the dynamic finger conjecture for splay trees. part ii: The proof. *SIAM J. Comput.*, 30(1):44–85, 2000.

[DJ07] Brian C. Dean and Zachary H. Jones. Exploring the duality between skip lists and binary search trees. In *ACM-SE 45: Proceedings of the 45th annual southeast regional conference*, pages **395–399**. ACM, New York, NY, USA, 2007.

[FLPR99] Matteo Frigo, Charles E. Leiserson, Harald Prokop, and Sridhar Ramachandran. Cache-oblivious algorithms. In *FOCS '99: Proceedings*

of the 40th Annual Symposium on Foundations of Computer Science, pages **285–397**. IEEE Computer Society, Washington, DC, USA, 1999.

[GTG01] Michael T. Goodrich, Roberto Tamassia, and Michael Goodrich. *Algorithm Design: Foundations, Analysis, and Internet Examples*. Wiley, September 2001. **235-240** pp.

[HM82] Scott Huddleston and Kurt Mehlhorn. A new data structure for representing sorted lists. *Acta Informatica*, 17(2):157–184, 1982.

[Hoc97] Dorit S. Hochbaum. *Approximation Algorithms for NP-hard Problems*, chapter 13, pages **521–564**. PWS Publishing Company, first edition, 1997.

[IL02] John Iacono and Stefan Langerman. Queaps. In *ISAAC '02: Proceedings of the 13th International Symposium on Algorithms and Computation*, pages **211–218**. Springer-Verlag, London, UK, 2002.

[Knu98] Donald E. Knuth. *Art of Computer Programming, Volume 3: Sorting and Searching*. Addison-Wesley Professional, second edition, April 1998. **409-417, 482-491** pp.

[MPS92] J. Ian Munro, Thomas Papadakis, and Robert Sedgewick. Deterministic skip lists. In *SODA '92: Proceedings of the third annual ACM-SIAM symposium on Discrete algorithms*, pages **367–375**. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1992.

[Pug90] William Pugh. Skip lists: a probabilistic alternative to balanced trees. *Commun. ACM*, 33(6):**668–676**, 1990.

[ST85a] Daniel D. Sleator and Robert E. Tarjan. Amortized efficiency of list update and paging rules. *Commun. ACM*, 28(2):**202–208**, 1985.

[ST85b] Daniel Dominic Sleator and Robert Endre Tarjan. Self-adjusting binary search trees. *J. ACM*, 32(3):**652–686**, 1985.

[Tar85a] Robert E. Tarjan. Amortized computational complexity. *SIAM*, 6(2): 306–318, 1985.

[Tar85b] Robert E. Tarjan. Sequential access in splay trees takes linear time. *Combinatorica*, 5(4):367–378, 1985.

[TG98] Andrew S. Tanenbaum and James R. Goodman. *Structured Computer Organization*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1998. **16-19** pp.

Note. **Bold faced** page numbering means these pages have been read carefully.

A Test result

Below are all the graphs from the empirical tests, conducted in Chapter 6. The data used for these results can be found on the URL:
http://www.cs.au.dk/~henrik/thesis/test/.

A.1 Comparing dynamic self adjusting skip list(continued)

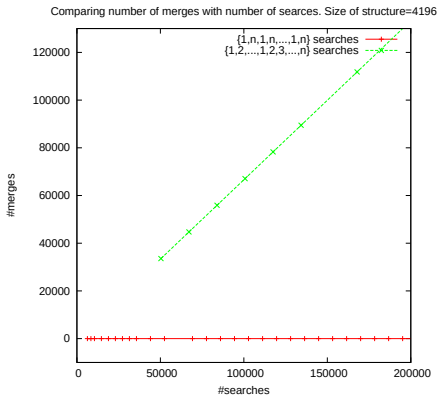


Figure A.1: Comparing number of merges in stucutures of size 4196.

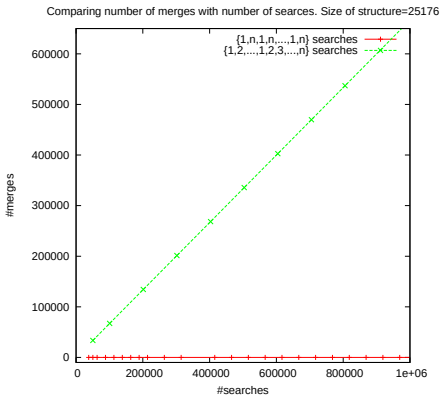


Figure A.2: Comparing number of merges in stucutures of size 25176.

A.2 Comparing self adjusting search trees (continued)

A.2.1 Sequences $\{1, n, 1, n, 1, n, \ldots, 1, n\}$

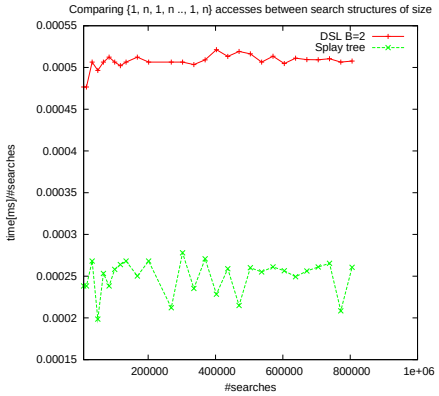


Figure A.3: Searching for $\{1, 4196, 1, 4196, 1, 4196, \ldots, 4196\}$ in stucutures containing elements $\{1, 2, 3, \ldots, 4196\}$.

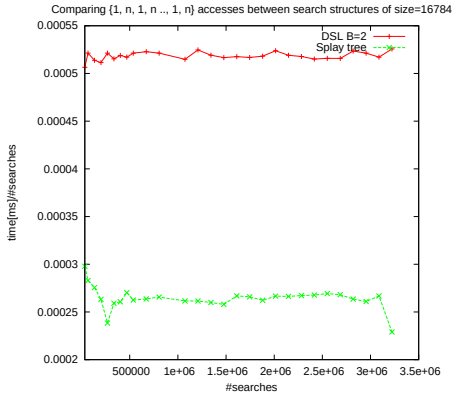


Figure A.4: Searching for $\{1, 16784, 1, 16784, 1, 16784, \ldots, 16784\}$ in stucutures containing elements $\{1, 2, 3, \ldots, 16784\}$.

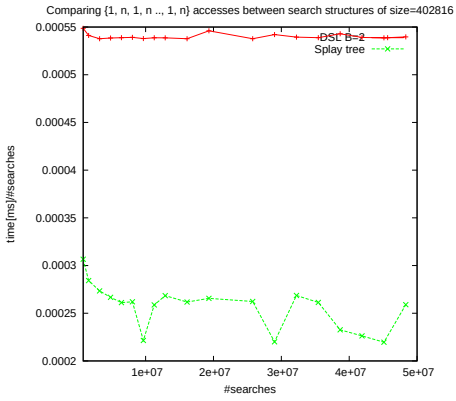


Figure A.5: Searching for $\{1, 402816, 1, 402816, 1, 402816, \ldots, 402816\}$ in stucutures containing elements $\{1, 2, 3, \ldots, 402816\}$.

A.2.2 Sequences $\{1, 2, 1, 2, \ldots, 1, 2\}$

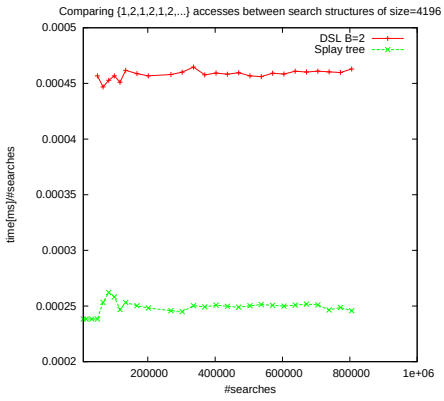


Figure A.6: Searching for $\{1, 2, 1, 2, 1, 2, \ldots, 1, 2\}$ in stucutures containing elements $\{1, 2, 3, \ldots, 4196\}$.

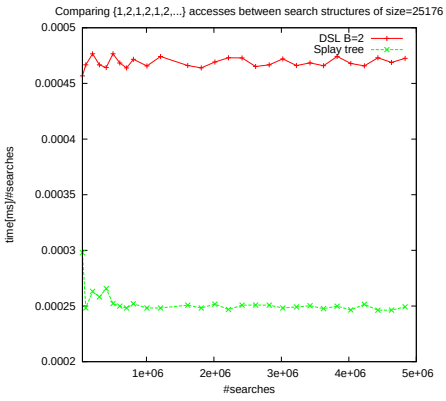


Figure A.7: Searching for $\{1, 2, 1, 2, 1, 2, \ldots, 1, 2\}$ in stucutures containing elements $\{1, 2, 3, \ldots, 25176\}$.

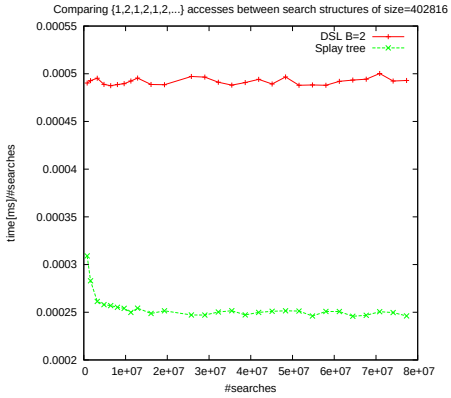


Figure A.8: Searching for $\{1, 2, 1, 2, 1, 2, \ldots, 2\}$ in stucutures containing elements $\{1, 2, 3, \ldots, 402816\}$.

A.2.3 Increasing working set numbers

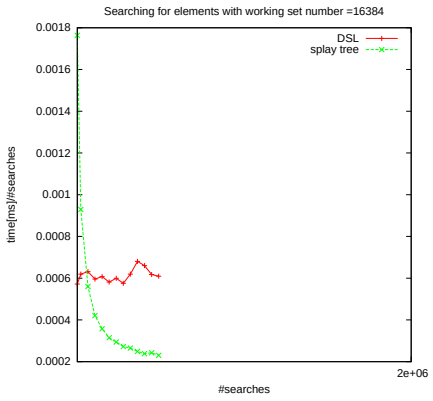


Figure A.9: Searching for $\{1, 2, 3, \ldots, 16384, 1, \ldots\}$ in stuctures.

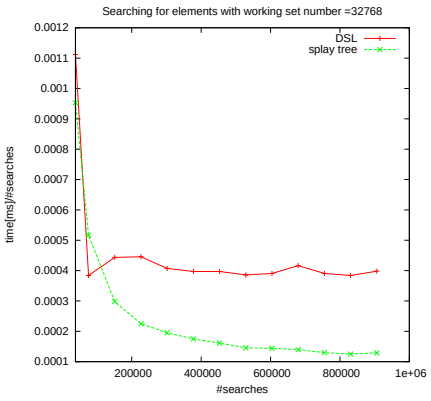


Figure A.10: Searching for $\{1, 2, 3, \ldots, 32768, 1, \ldots\}$ in stuctures.

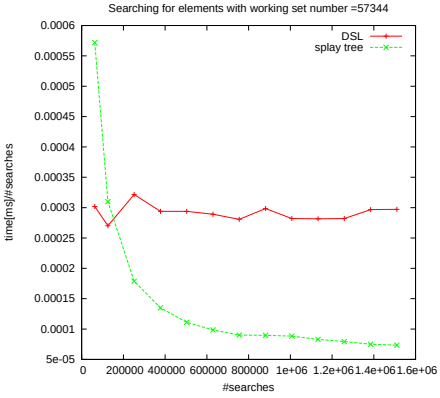


Figure A.11: Searching for $\{1, 2, 3, 57344, 1, \ldots\}$ in stuctures.

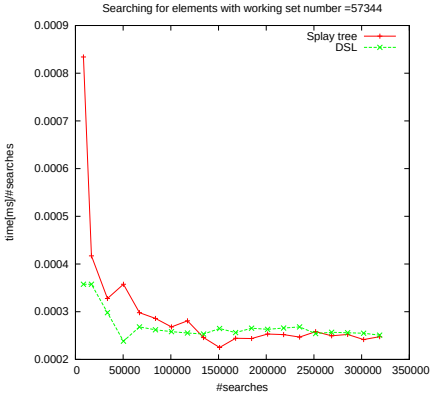


Figure A.12: Searching for random elements within a working set number 57344.

Figure A.13: Searching for random elements within a working set number 73728.

A.2.4 Sequences $\{1, 2, 3, \ldots, n, 1, 2 \ldots\}$

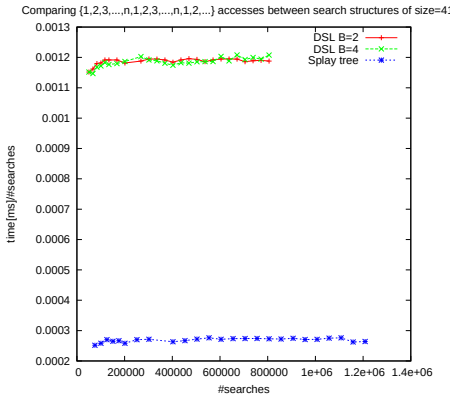


Figure A.14: Searching for $\{1, 2, 3, \ldots, 4196, 1, 2, \ldots, 4196\}$ in stucutures containing elements $\{1, 2, 3, \ldots, 4196\}$.

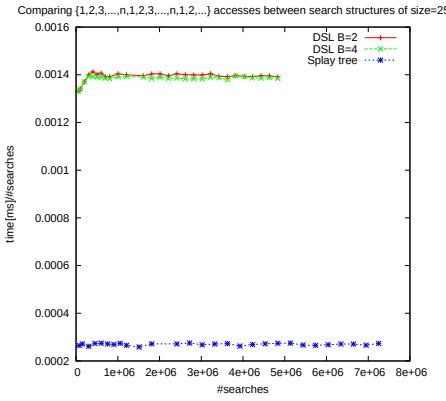


Figure A.15: Searching for $\{1, 2, 3, \ldots, 25176, 1, 2, \ldots, 25176\}$ in stucutures containing elements $\{1, 2, 3, \ldots, 25176\}$.

A.2.5 Unified access sequences

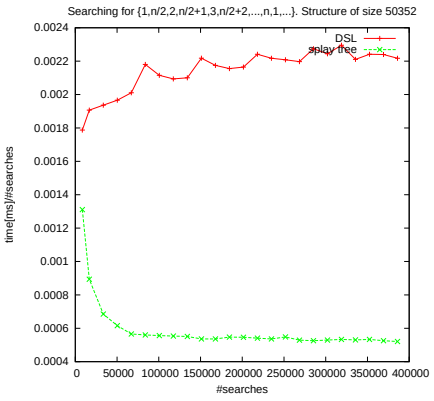


Figure A.16: Searching for $\{1, 50352/2, 2, 50352/2 + 1, 3, 50352/2 + 2, \ldots, 50352/2 - 1, 50352, 1, \ldots\}$ in stucutures containing elements $\{1, 2, 3, \ldots, 50352\}$.

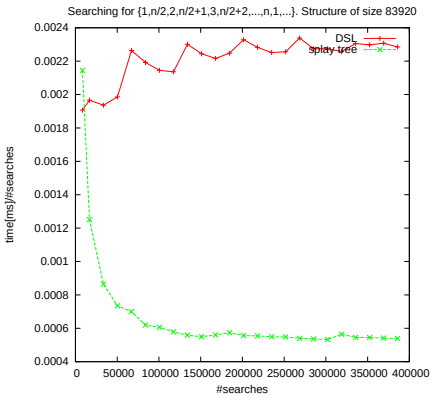


Figure A.17: Searching for $\{1, 83920/2, 2, 83920/2 + 1, 3, 83920/2 + 2, \ldots, 83920/2 - 1, 83920, 1, \ldots\}$ in stucutures containing elements $\{1, 2, 3, \ldots, 83920\}$.

B Test environment

B.1 Equipment

The test described in this chapter, is conducted on a dual core *Intel(R) Xeon(TM) CPU 3.00GHz* computer with 2 *MB* cache. Furthermore this computer has been supplied with 2 *GB DDR2 RAM*.

```
$ cat /proc/cpuinfo
processor       : 0
vendor_id       : GenuineIntel
cpu family      : 15
model           : 4
model name      : Intel(R) Xeon(TM) CPU 3.00GHz
stepping        : 3
cpu MHz         : 3000.284
cache size      : 2048 KB
physical id     : 0
siblings        : 2
core id         : 0
cpu cores       : 1
fdiv_bug        : no
hlt_bug         : no
f00f_bug        : no
coma_bug        : no
fpu             : yes
fpu_exception   : yes
cpuid level     : 5
wp              : yes
flags           : fpu vme de pse tsc msr pae mce cx8
                  apic sep mtrr pge mca cmov pat pse36
                  clflush dts acpi mmx fxsr sse sse2 ss ht
                  tm pbe nx lm constant_tsc pebs bts sync_rdtsc
                  pni monitor ds_cpl cid cx16 xtpr
bogomips        : 6004.61
clflush size    : 64
```

```
processor       : 1
vendor_id       : GenuineIntel
cpu family      : 15
model           : 4
model name      : Intel(R) Xeon(TM) CPU 3.00GHz
stepping        : 3
cpu MHz         : 3000.284
cache size      : 2048 KB
physical id     : 0
siblings        : 2
core id         : 0
cpu cores       : 1
fdiv_bug        : no
hlt_bug         : no
f00f_bug        : no
coma_bug        : no
fpu             : yes
fpu_exception   : yes
cpuid level     : 5
wp              : yes
flags           : fpu vme de pse tsc msr pae mce cx8
                  apic sep mtrr pge mca cmov pat pse36
                  clflush dts acpi mmx fxsr sse sse2 ss
                  ht tm pbe nx lm constant_tsc pebs bts
                  sync_rdtsc pni monitor ds_cpl cid
                  cx16 xtpr
bogomips        : 6000.26
clflush size    : 64
```

```
$ free
             total       used       free     shared    buffers
Mem:       1035908     855388     180520          0       3520
-/+ buffers/cache:      62576     973332
Swap:      2650684       8640    2642044
```

The operating system installed on the test machine is a the linux distribution *Debian Sid*. This distribution is installed with:

```
$ uname -a
Linux camel15 2.6.24-perfctr #1
    SMP Mon Sep 8 21:53:20 CEST 2008 i686 GNU/Linux
```

All test is performed without a running *X*-server and a minimum of services.

We have used *GCC*'s *C++* compiler to compile the source code:

```
$ gcc -v
Using built-in specs.
Target: i486-linux-gnu
Configured with: ../src/configure -v
  --with-pkgversion='Debian 4.3.2-1'
  --with-bugurl=file:///usr/share/doc/gcc-4.3/README.Bugs

  --enable-languages=c,c++,fortran,objc,obj-c++
  --prefix=/usr
  --enable-shared
  --with-system-zlib
  --libexecdir=/usr/lib
  --without-included-gettext
  --enable-threads=posix
  --enable-nls
  --with-gxx-include-dir=/usr/include/c++/4.3
  --program-suffix=-4.3
  --enable-clocale=gnu
  --enable-libstdcxx-debug
  --enable-objc-gc
  --enable-mpfr
  --enable-targets=all
  --enable-cld
  --enable-checking=release
  --build=i486-linux-gnu
  --host=i486-linux-gnu
  --target=i486-linux-gnu
Thread model: posix
gcc version 4.3.2 (Debian 4.3.2-1)
```

The compile options given to *GCC* when compiling for testing are:

```
g++ -o DSkipList/DSkipList.o -c -O2 -Wall -DTEST=true
     testSuite.cpp
```

B.2 To compile and run

To compile and run the code presented in this thesis following thing are needed.

- GCC C++ compiler

- SCons
- gprof

To build the source suite follow the following procedure:

1. $ cd /src
2. *Choose between profiling/test mode.*
   (a) $ scons debug=2 - *to build profiling files.*
   (b) $ scons debug=3 - *to build test files.*

When the code is build, it should be simple to start a test, testing all sequences mentioned in Chapter 6, just execute:

```
$ ./runTest.sh
```

To test a single structure with a given set of parameters run on of the following commands:

```
$ ./DSkipList/dSkipList_{test,profiling}
$ ./SplayTree/splaytree_{test,profiling}
```

The commands should be run with the requested set of parameters and the structure should of course be build before doing this step.