HELSINKI UNIVERSITY OF TECHNOLOGY
Department of Computer Science and Engineering
Degree Programme of Computer Science and Engineering

Satu Virtanen

Group Update and Relaxed Balance in Search Trees

Master's thesis submitted in partial fulfillment of the requirements for the degree of Master of Science in Technology

Supervisor and instructor Professor Eljas Soisalon-Soininen

Espoo, December 4, 2000

Tekijä:	Satu Virtanen		
Työn nimi:	Ryhmäpäivitys ja löyhä tasapaino hakupuissa		
English title:	Group Update and Relaxed Balance in Search Trees		
Päivämäärä:	4. joulukuuta, 2000	**Sivumäärä:**	81
Osasto:	Tietotekniikan osasto		
Professuuri:	Tik-106 Ohjelmistojärjestelmät		
Työn valvoja:	Professori Eljas Soisalon-Soininen		
Työn ohjaaja:	Professori Eljas Soisalon-Soininen		

Tämä työ tarkastelee hakupuihin perustuvan indeksoinnin tehokuutta. Työ sisältää katsauksen perinteiseen puuindeksointiin sekä esittelee kaksi tunnettua tapaa parantaa indeksipuiden toimintaa. Työssä esitetään algoritmi eräälle binääripuuluokalle, jossa nämä kaksi strategiaa yhdistyvät. Algoritmia analysoidaan matemaattisesti ja sen tehokkuutta tutkitaan kokeellisesti yksinkertaistetun Java-toteutuksen avulla.

Tapoja indeksoinnin tehostamiseen tarvitaan käsiteltävien tietomäärien kasvaessa. Esimerkiksi WWW-hakukoneet käsittävät nykyisellään yli miljardi dokumenttia. Lisäksi tietokantojen käyttäjämäärät ovat nopeassa kasvussa; hakutiheys kasvaa käyttäjäkunnan myötä. Perinteiset indeksihaut ja -päivitykset vaativat pitkiä kulkuja indeksipuussa sekä vaarantavat indeksin palvelutehon sulkemalla osia puusta vain omaan käyttöönsä.

Indeksien tehokkaampaan käyttöön pyritään nykyisin kahdella uudistuksella. Ryhmäpäivityksessä päätavoitteena on vähentää tarvittavan puussa kulkemisen määrää indeksisolmujen paikallistamisen yhteydessä sekä helpottaa päivitysten läpivientiä. Toinen lähestymistapa liittyy puun tasapainon ylläpitoon; tämä ns. löyhä tasapainotus erottaa tasapainotusvaiheen päivityksestä pyrkien siten vähentämään operaatioiden poissulkevuutta puussa. Työssä esitetyn analyysin mukaan se myös vähentää tasapainotukseen tarvittavan laskennan määrää.

Työssä kuvattu algoritmi yhdistää ryhmäpäivityksen ja löyhän tasapainotuksen ulkoisille binäärihakupuille. Algoritmianalyysi osoittaa, että verrattuna yksittäisiin tiukasti tasapainotettuihin operaatioihin, kompleksisuus on samaa tai pienempää luokkaa riippuen avainjakaumasta. Kokeelliset tulokset osoittavat ryhmäpäivityksen selvästi tehokkaammaksi, mutta tasapainotusmenettelyjen keskinäisiin suhteisiin ei tehdyillä kokeilla saatu varmistusta. Joskaan löyhän tasapainotuksen ei työssä todisteta tuovan säästöä työmäärässä, voidaan luotettavasti todeta sen olevan kompleksisuudeltaan lähellä tiukan tasapainotuksen suuruusluokkaa. Indeksioperaatioiden skeduloinnin sekä rinnakkaisen käytön vaatiman lukinnan kannalta löyhä tasapainotus on suotuisampi toimintatapa.

Avainsanat: indeksointi, hakupuu, ryhmäpäivitys, löyhä tasapainotus

Ei lainata ennen:	**Työn sijaintipaikka:**

HELSINKI UNIVERSITY OF TECHNOLOGY ABSTRACT OF MASTER'S THESIS

Author:	Satu Virtanen		
Title of thesis:	Group Update and Relaxed Balance in Search Trees		
Finnish title:	Ryhmäpäivitys ja löyhä tasapaino hakupuissa		
Date:	December 4, 2000	**Pages:**	81
Department:	Department of Computer Science and Engineering		
Chair:	Tik-106 Software Technology		
Supervisor:	Professor Eljas Soisalon-Soininen		
Instructor:	Professor Eljas Soisalon-Soininen		

This thesis examines the efficiency of indexing with search trees. It includes an overview of the traditional means of operation on tree-based indices and introductions of two known improvement strategies. The thesis describes an algorithm for a class of binary trees that combines these improvements. The analysis of the algorithm is presented and experiments are conducted with a simplified Java model.

Efficient database indexing is a necessity as the size of multiuser databases increases rapidly, for example in WWW search engines, and the service request frequency also grows together with the number of users. The traditional approach of single insertions with strictly coupled tree-rebalancing risks the service of other requests by blocking the index. It also requires intense traversing of the index tree during the search of a key.

Currently there are two main strategies for improving index performance. The method of group update eases the computational overhead in locating the position nodes of the requested information. This is achieved by reducing the number of links needed to traverse per key by locating the positions of several of keys at once. The second improvement strategy, relaxed balancing, separates the index maintenance from the updates. This lowers the risk of blocking and also has the opportunity to reduce the number of rotations needed to maintain the index in balance.

The algorithm presented in this thesis combines these two methods for external binary search trees. It is analyzed for complexity and experimented on with a model implementation, comparing the possible combinations of the traditional approaches and the suggested improvements. The results of both the analysis and the experiments show that the group update algorithm is never inferior to the single operations and often more efficient. The balance complexity is quite similar for the strict and relaxed approaches, but the analysis shows that significant savings are possible from the relaxed balancing under favorable conditions. The relaxed approach is also more favorable to scheduling index operations and performing concurrent operations.

Keywords: indexing, search tree, group update, relaxed balancing

Not borrowable till: **Library code:**

Preface

Even though the hours spent at the computer working on this thesis were not few, those that I spent complaining about it to others were numerous. Those conversations just might have been the most beneficial time invested in this work, and I am in great gratitude to all those who I bothered during the last few months. Especially I thank my instructor and supervisor Professor Soisalon-Soininen not only for providing me a steady income, but for the ideas, explanations — even though confusing at times — and the literature references. Constant disagreement on what is the best approach to a problem and endless refining of the solutions finally gave me some insight into what algorithm analysis and design is, besides tedious.

On the home front I owe much to Kosti, my partner in life and fellow student, who helped me improve the text by reading it carefully, however boring it may be to non-devoted readers. He gave many helpful comments on the presentation, finding errors already invisible to me.

Also my co-workers have been most helpful. Most of all I thank Esko Nuutila among all for the helpful conversations, LaTeX examples, and criticism. I also thank Vesa Hirvisalo who helped me in using `gnuplot` and Riku Saikkonen for providing for a couple of helpful Makefiles and tips on `xfig`. Instead of attempting to list all those from the Laboratory of Information Processing Science who have been helpful, I thank the laboratory as a whole.

Espoo, December 4, 2000

Satu Virtanen

Contents

1 Introduction **1**

2 Indexing with Search Trees **3**

2.1 Types of Index Trees 4

2.2 External Binary Search Trees 5

2.3 Index Operations 7

2.3.1 Search 7

2.3.2 Insert 8

2.3.3 Delete 9

2.4 Size and Shape of Index Trees 9

2.5 Standard Rotations 13

3 Problematics of Traditional Indexing **17**

3.1 Single Updates with Strictly Coupled Balancing 17

3.2 Concurrency Control 18

4 Known Improvements **21**

4.1 Relaxed Balancing 21

4.1.1 Effects of Uncoupling Update and Balance Phases 22

4.1.2 Trees with Relaxed Balance 23

4.2 Group Update 25

4.2.1 Required Modifications on an Update 26

4.2.2 Concurrency and Group Updates 28

5 The Algorithm 29

5.1 Summary of the EBST 30

5.2 Locate Phase 30

5.3 Update Phase 32

5.3.1 Insertion 32

5.3.2 Deletion 33

5.4 Balance Phase 34

6 Algorithm Analysis 39

6.1 Proof of Correctness 39

6.2 Proof of Complexity 45

6.2.1 Locate Complexity 45

6.2.2 Update Complexity 49

6.2.3 Balance Complexity 49

6.2.4 Strictly-Coupled Single Update Complexity 52

6.2.5 Total Complexity 54

7 Experiments 58

7.1 System Variations 58

7.2 Metrics 62

7.3 Description of Implementation 63

7.3.1 Packages 63

7.3.2 Operating on the Index Tree 64

7.4 Measurements 66

7.5 Test Results 67

7.5.1 Single Operations 68

7.5.2 Group Operations 69

7.6 Evaluation 72

8 Conclusions 76

Bibliography 78

Notational Conventions

The abbreviations and acronyms used in this text are few. There are only two that need to be familiar in order to follow the discussion. **EBST** stands for *external binary search trees*, a the class of trees defined in Section 2.2. Also another acronym referring to a class of trees is necessary; **AVL** trees [1] are named after their inventors Adel'son-Vel'skiĭ and Landis.

$\log x$	Logarithm to the base two rounded to the closes integer $< x$, $\lfloor \log_2 x \rfloor$
$O(x)$	Worst-case complexity proportional to x
$\Omega(x)$	Lower-bound of complexity proportional to x
$\Theta(x)$	Complexity both $O(x)$ and $\Omega(x)$

Trees are generally marked with capital letters, as individual nodes are named with lower-case letters. The size of a tree is signified with an algebraic notation of absolute value, used amongst all in set theory: $|T|$.

In the illustrations of trees, *the root is drawn as the topmost node* when shown, as recommended by Knuth in [25]. Otherwise the direction of the root is signified with a dotted-line parent link from the subtree root. In EBSTs, *leaf nodes are colored gray and routers white* to remind the reader of the distinction. At times the trees are not drawn full at the leaf level, but a subtree is replaced by a triangular shape. If not stated otherwise, that subtree may also be empty.

Chapter 1

Introduction

As the amount and significance of digital information in everyday life has grown substantially over the last decade, improvements in data management are welcome. A central part in database operation is the index, one or often more, that displays the structure of the database. A database is accessed only through the index, as simply scanning the storage files is highly inefficient in most cases [26]. In this thesis, an overview of the problematics of efficient database indexing is provided, with emphasis on update and rebalancing.

Different types of trees are commonly used structures for indexing, especially variations of balanced binary search trees. They provide an efficient implementation for the basic index operations: searching, inserting, or deleting a specific key and associated data [26]. However, the problematics of keeping the tree in balance and controling simultaneous operations downgrade the achieved performance. Indices can reside both in main memory and in other storage medium. B-trees are a common choice for disk-based indices, whereas binary trees are mostly used for main-memory applications.

Two important improvement strategies are *group update* and *relaxed balancing*. In group update operations are batched instead of processing each separately [20, 52]. Relaxed balancing means that the maintenance of the index is delayed as the index is in heavy use, and performed at a more tranquil moment [18, 24, 33, 40]. These two approaches have recently been combined for a few types of tree indices [33, 34, 35], but the resulting solutions have not yet steadily improved the index performance in all circumstances. Even though the algorithms speed up the processing under certain conditions, the performance is below the original level in others [33].

In this thesis a combination approach is suggested for a class of binary trees. The goal is to provide an algorithm whose performance is of at least the original level in all conditions while it still ensures the speed-up for those conditions favorable to group updates. The suggested solution is analyzed in detail and examined in practice with a simplified but fully functional model of the suggested solution. The analysis shows that the cost of the proposed algorithm is of similar or smaller magnitude than the traditional usage of individually balanced single operations. It can be safely concluded from the experiments that the group update brings savings in the total operation cost, but the comparison of strictly coupled and relaxed balancing cost is inconclusive. Further research that could provide answers to the open questions are described with the test results.

The discussion of the improvement strategies together with the analysis and the experiments leads to the conclusion that the proposed algorithm is an improvement to operating the index. It introduces significantly lower search cost and competitive update and rebalancing cost together with a less restrictive operation mode. The balancing may be performed gradually and paused at times, and the amount of locking required for concurrent operations is lower.

The text is organized as follows. First, in Chapter 2 the basic concepts of trees in database indexing are presented. The problems of the traditional approach are discussed in Chapter 3. Furthermore, in Chapter 4 the known improvements of relaxed balancing and group updating are explained and the currently known approaches are presented. A relaxed-balance group update algorithm basing on these improvement strategies is presented in Chapter 5 and analyzed for correctness and computational complexity in Chapter 6. For practical comparison, experiments with the model implementation are documented in Chapter 7. Finally in Chapter 8 the suggested approach and its performance are briefly summarized and the implications of this work discussed.

Chapter 2

Indexing with Search Trees

In this chapter a general outline is given on the practice of information indexing. The nature and importance of indices is explained and the problematics pointed out through some concrete examples. In Section 2.1 some of the different types of tree structures used in indexing are introduced and the choice of trees used in this thesis is explained. The chosen class of external binary search trees (EBST) is defined in Section 2.2. The index operations together with their implementation on an EBST are presented in Section 2.3. The effect of the index operations on the structure of an EBST are discussed in Section 2.4, and a solution for the problems caused by the structural changes is presented in Section 2.5.

Indices are not only used for traditional databases such as library catalogues and employee databases, but only an important part of the functionality of for example keeping track of mobile phones present at a certain base transceiver station. File directories such as Microsoft's Active Directory, storing information of network objects in a hierarchical manner have grown popular in large corporations and communities; they also rely on indices for service speed. Also, as data mining is used to retrieve information, at least the intermediate storage of data needs an index to be accessed fluently.

Every large database requires an *index* or often several indices, through which they locate and retrieve data. An index is practically a listing of the contents of a relation or an entire database, sorted by some chosen information. For example an employee database in a company could have one index by the social security numbers of employees and one by name. When the information of a certain employee is needed, his or her information is retrieved through one of the indices depending on whether the person making the query knows the social security number or just the name. The index contains the information where in the database (i.e. where on the disk storing the data) the employee data is located. With this index information that data may be retrieved.

As an example of databases and indices, a full-text database is both illustrative and concrete. Large document databases have become a popular tool for researchers and students, as digital libraries available on the World Wide Web have grown to be an essential source of scientific information. The demands on information retrieval systems are rapidly increasing both by the growing amount of data and the growing user population. Formerly a database was operated and queried by just a few scientists or engineers; compared to that, the patience and know-how of the current users seems diminutive [55].

The *full-text indexing* also known as *inverted-index technique* is a wide-spread approach for indexing text documents in such a manner that each word in a document may be an index term, instead of using only predefined keywords for the indexing [44, 45]. A drawback of this method is evidently the heavy load on an update — each of the often numerous index terms will require to be updated as a document changes. Just one short document, a few pages in length, contains thousands of search keys [54].

2.1 Types of Index Trees

As the purpose of indexing is familiar, a more abstract approach can be adopted. Essentially, database indexing is a problem of *external searching*, in which a transaction needs to access information stored in a large sequential file or a collection of such files. A typical data structure for such an index is a *tree* [26].

Binary search trees are widely used and fairly efficient data structures for handling ordered data [26, 50]. Hence they are also of high importance in indexing. Appropriately chosen tree representations meet the demands of external searching well, as they can be split up into natural blocks or *subtrees* that can be assigned to a separate file each and still traversed efficiently due to their ordered nature. The idea is best portrayed in Figure 2.1, adopted from Knuth [26].

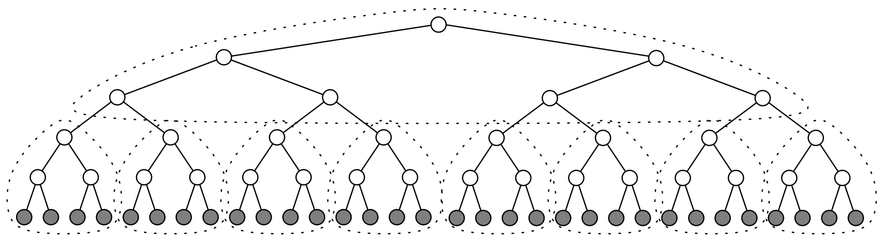


FIGURE 2.1: An example of splitting an index tree on several pages.

A step towards the current trend in tree indices was taken by Landauer [27] in 1963 by introducing a balanced tree kept in proper form by insertion ordering and heavy-cost rotations upon updates. His work was based on the understandable assumption that data updates are quite rare in comparison to information retrieval. After that and the introduction of AVL trees in 1962 [1], a variety of balanced search trees with $O(\log n)$ complexity for basic index operations were developed. One of these advances was the definition of *B–trees*[1] by Bayer and McCreight [9].

Another genre of binary search trees known as *red-black trees* [18] has also been a common choice for index trees, recently researched by Hanke in her PhD thesis [19]. These trees are also searched and updated in $O(\log n)$ time and kept in balance through rotations. The imbalance is located by observing the *colors* of successive nodes, searching for so-called red–black violations.

Ever since the seventies, balanced variations of B-trees and binary search trees have been typical as index structures. In this thesis, I have chosen to concentrate on *external binary search trees* (EBST) for analytical simplicity and fair performance without enhancement.

[1]B-trees are not explained in this thesis even though referred to at times. For a discussion on B-trees, see e.g. Knuth [26]. Knowledge on them is not required for following the text.

It is also an easily generalizable structure. To ensure uniform terminology on the behalf of the discussion and the reader, I shall start with defining the chosen class of trees and their operations.

2.2 External Binary Search Trees

A general definition for a tree is a set or a graph consisting of *nodes* and *edges*. The edges or *links* are usually directed in the sense that the node where an edge originates is considered the *parent* of the node where the edge ends, although in practice, the actual traverse direction of the edge can be either one and often both. A more general and formal discussion on trees is available by Knuth [25]; here I simply summarize those aspects that are essential in further discussion.

In a *binary tree*, each node has one parent (i.e. one incoming edge) and at most two *children* (i.e. targets of outgoing edges). The children are ordered and distinguishable in the sense that one is the *left* child and the other the *right* child. The children of the same node are each other's *siblings*.

An exception to the rule is *a parentless node* known as *root*, which is considered the beginning of the tree, through which the tree is conventionally accessed. Any node is the root of a *subtree* beginning in that node. The subtree includes the node itself and any node that is reachable from there (its children, their children and so forth). The *left subtree* of a node is hence the subtree rooted at the node's left child. Any node in the subtree of a certain node is a *descendant* of that node, and any node on the path from a node to the root is an *ancestor* of that node.

In order to have a finite tree, some nodes must not have any more children. A childless node is called a *leaf* or more generally an *external node*. In some classes of trees, leaves and routers have different purposes for the tree operation.

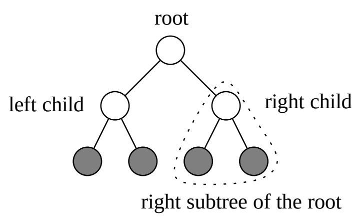


FIGURE 2.2: An example of a binary tree.

A *search tree* is a tree used to store and locate certain information. The only requirement on the information is that it can be ordered; e.g. words are ordered alphabetically. A piece of information in a search tree is called a *key*. The naming convention follows the fact that usually indices contain no data, only information on where a certain piece of data may be found. Thus the index information acts as a key to the actual data in the database. Commonly the key is paired with an address of the data [9]. In this thesis, the information associated in a certain key is of no further interest.

In an *external tree*, all keys are stored in the leaves and the internal nodes act merely as *routers* that present the order of the keys within the tree. The value of a router states the order of the keys stored in the subtrees of that particular router; those that come before the router value when ordering the keys are located in the left subtree, and the others in the right subtree. Thus the nodes of an external tree are all either routers if internal and leaves if external. This is illustrated in Figure 2.2 by coloring the leaves gray and leaving the routers white. The same practice will be used to display EBSTs throughout this thesis.

External binary trees are *full* trees, meaning that each node may have either two or no children. This definition together gives that all routers must have exactly two children and no leaf naturally has any children. This class of trees is also called $0-2$-trees [25].

Each leaf contains one key value, and the routers also have values that are members of the possible key space but not necessarily values of existing leaves.[2] This loose definition of router values is common, as the overhead of updating them as the leaf keys change is both tremendous and unnecessary.

To return to the suitability of trees into indexing, it is now possible to explain the division of a tree into several pages in memory as in Figure 2.1. As a search process for a particular key begins at the root, the first page needed should contain as deep a portion from the index tree as fits on a single page, cut from a certain level (after three levels in Figure 2.1). The subtrees that were cut out from the primary file can each be assigned an own page, if they are appropriately small, or divided further in the same manner that the original index tree.

In this thesis, positive integers are chosen as the keys, as their order is easily perceived and they are easily generalized to anything that can be reduced to positive integers of any length. For example, words of the English language are reducible to integers e.g. as a sequence of the ASCII-values of the letters.

The keys of a tree are stored in an orderly fashion so they can be located without proceeding through the entire tree. A usual criterion for the location of a certain key is the following: beginning from the root, if the value of the key stored in the root is smaller than the key being located, continue at the left subtree. Otherwise, continue at the right subtree. In literature, the choice on whether equal keys go to the left or the right subtree varies.

Definition 2.1. An EBST tree T is in *order* if for all nodes t in T the following applies:
For all nodes s in the left subtree of t, *key*$(s) <$ *key*(t) and for all nodes r in the right subtree of t, *key*$(r) \geq$ *key*(t).

As the location procedure arrives to a leaf node, the correct location for the key in question has been found. The leaf where a query of a specific key ended is called the *position leaf* of that key.

The recursive procedure can be expressed as the following pseudo-code. The all nodes have a key value, represented by `node.key`, and respectively the child nodes of a router are `router.left` and `router.right`. The approach is object-oriented for fluency in reading.

[2]A router gets its value as it is created in a leaf split discussed in Section 2.3 from one of the leaf key values involved in the split. The leaf that 'gave' the value to the router may however be deleted at a later time, without necessarily removing the router. Routers must nevertheless contain values that are possible key values, as otherwise they would block access to a part of the index, as the application uses key values to navigate in the structure.

```
procedure locate(int key, node current) : leaf {
  if (current is leaf) {
    return leaf ;
  } else {
    if (current.key < key) {
      locate(key, current.right);
    } else {
      locate(key, current.left);
    }
  }
}
```

The locating of the position leaf of a key in a tree always begins at the root of that tree. With the procedure for locate defined above, it is possible to give a definition for the position leaf:

Definition 2.2. The *position leaf* of a key k in a tree rooted at node t is the leaf that the procedure locate(k, t) returns.

The position leaf is thus the leaf where an index operation would take place for a given key. The set of operations on an EBST is defined in the following section.

2.3 Index Operations

The possible set of operations on an index is not a fixed set; each application developer may choose his or her own operations. The most common are *searching* for a given key or a key position in the index, *fetching the next key* to the previously accessed, *inserting* a new key into the index, and *deleting* an existing key from the index. Search, insert and delete operation are examined in more detail. EBSTs do not provide special mechanisms for fetching the next key; for example a special case of B-trees called B$^{\text{link}}$-trees support traversing directly from one leaf to the next.

In order to modify a database entry, one must first search for the corresponding key in the index, retrieve the data pointed to by the key information, store the data locally, delete the entry from the database, make the necessary modifications locally and insert it as new data. This is why some have chosen to include a fourth, separate modification operation, but for formal treatment, researchers have usually limited the examination to search, insert and delete.

In the following discussion it is expected that the key of the data operated upon is known. The details on how the key is determined are application-specific. It may be a hash value computed from some part of the data, e.g. the name of a person or otherwise obtained.

2.3.1 Search

All databases need to be accessed and a vast majority modified with varying frequency. The access operation is called a *search* operation. It consists of locating the key in question from the index and using the key to retrieve data from the storage. This is the most common database operation in most practical databases, such as search engines and library catalogs.

```
procedure search(int key, node root) : data {
   leaf location ← locate(key, root);

   if (location is null ) {
      return null // KEY NOT IN TREE
   } else {
      return location.retrieveData();
   }
}
```

As seen from the above outline, a search operation does not modify the index tree in any way. Hence, if a search operation is done on an EBST, the tree is still a valid EBST after completing the operation.

Lemma 2.1. If a search operation is performed on an EBST T, the resulting tree T' is also an EBST, and $T = T'$.

2.3.2 Insert

If new keys need to be added to the index, the necessary operation is an *insert*. If data with the same key already exists in the tree, it depends on the application area what needs to be done. In a library catalog, for example, it must be possible for one keyword to point to several different publications, but in a population register, one social security number may only point to one person. For simplicity, the latter option is used here as in fundamental literature [26].

```
procedure insert(int key, node root) : router {
   leaf location ← locate(key, root);

   if (location.key is key) {
      return null // KEY ALREADY IN TREE
   } else { // KEY INSERTION
      return location.split(key); // OPERATION SUCCESSFUL
   }
}
```

Inserting a new leaf where one leaf already exists requires the creation of a new router that will hold these two leaves. The new router is placed on the location original position leaf, and the two leaves as its children. Here the splitting of the original leaf into a router and two leaves is denoted as the *split* operation, which returns the newly created router.

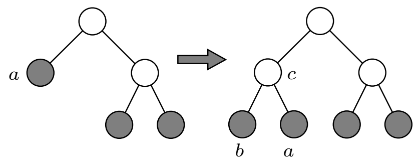


FIGURE 2.3: Inserting a new key $b < a$ results in the split of leaf a into a router c with the key value of a and a new leaf b.

Lemma 2.2. If a successful insert operation is performed on an EBST T, the resulting tree T' is also an EBST, and $|T| < |T'|$.

2.3.3 Delete

As a piece of information becomes outdated, it needs to be removed from the index. The removal operation is called *delete*. Upon deletion, the corresponding key needs to be located, the data pointed to by the key destroyed and the key removed from the index. The procedure that removes the key returns the parent of the location nodes parent, best explained by Figure 2.4.

```
procedure delete(int info, node root) : router {
  leaf location ← locate(key, root);

  if (location.key is not key) {
    return null // KEY NOT IN TREE
  } else { // KEY DELETION
    return location.remove();
  }
}
```

As can be seen from the procedure definition, the simple deletion of the requested leaf does not leave the index tree in a well-defined state. As each router is required to have exactly two children, the parent of the removed leaf needs to be merged with his parent. This is included in the definition of the `remove` procedure call above.

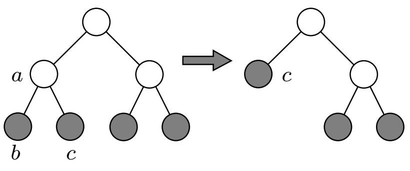


FIGURE 2.4: Deleting a single leaf b also removes the parent router a.

Lemma 2.3. If a successful delete operation is performed on an EBST T, the resulting tree T' is also an EBST, and $|T| > |T'|$.

2.4 Size and Shape of Index Trees

As seen from the operation definitions, using trees as indices is not completely straightforward. The insert and delete operations cause structural changes in the tree, which result in changes in the shape of the tree. A symmetrical tree with all leaves on the same level and all routers full can guarantee access in $O(\log n)$ time. Such a tree is said to be *completely balanced* or *optimal* with respect to the length of the path traversed going from the root to a leaf [5].

Definition 2.3. A full tree with all leaves on the same level is called an *optimal tree*. Each path from the root to a leaf has the same length in an optimal tree.

If the tree is not reshaped after updates but changes its structure randomly with incoming update operations, the outcome may be quite list-like. The very worst case is a tree reduced similar to a linked list having $n-1$ levels and linear-time worst-case complexity $O(n)$ [3]. A badly out-of-balance tree (called an *imbalanced* tree) can require large structural modifications to bring it in balance and take much time to optimize and thus block the index from use [24].

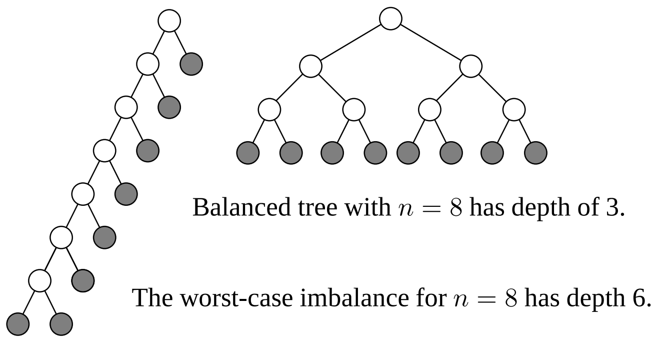

Balanced tree with $n = 8$ has depth of 3.
The worst-case imbalance for $n = 8$ has depth 6.

FIGURE 2.5: An imbalanced versus a balanced EBST for $n = 8$.

To avoid situations as in Figure 2.5, algorithms have been constructed to keep trees in perfect or close to optimal balance while being operated on. The use of binary search trees as database indices is therefore based not only on the ordered access, but also the possibility of *controlling the worst-case access time*.

Commonly, a balanced tree is defined such that for no node, the *height* of it subtrees differs by more than one. This is known as the AVL-balance condition, as it originates from AVL trees [1, 26]. Also other balance conditions are being used, using e.g. path lengths or assigning *weight* values to nodes [26].

To understand the definition of balance, the height of a subtree must first be defined: The height of a subtree is the height of the root node of the subtree. The height of a router is the length of the longest path from that router to a descendant leaf. For all leaves the height is one.

Definition 2.4.
The *height* of a node t, denoted by *height*(t) is

$$\textit{height}(t) = \begin{cases} 1, & \text{if t is a leaf,} \\ \max\{\textit{height}(\textit{left}(t)), \textit{height}(\textit{right}(t))\} + 1, & \text{if t is a router.} \end{cases}$$

Here *left*(t) and *right*(t) are the corresponding children of t.

The height of a tree is the height of the root of that tree; the same applies for subtrees:

Definition 2.5. The *height* of a tree T, denoted by *height*(T) is

$$\textit{height}(T) = \textit{height}(t), t \text{ is the root of } T.$$

Thus it is possible to write an expression for the balance of a tree:

Definition 2.6.
A tree T is in balance if *height*$(T) = O(\log n)$.

The possible conditions by which to ensure this balance are various. For the EBSTs I have chosen the AVL balance condition:

Definition 2.7. An EBST is in balance if for all nodes t in T applies

$$|\mathit{height}(\mathit{left}(t)) - \mathit{height}(\mathit{right}(t))| \leq 1.$$

To explain the usefulness of tree balance, the number of nodes in an external binary search tree needs to be considered. In the following discussion, the total number of nodes the a tree is denoted by N and the number of leaves by n. The depth of the tree is denoted by d.

In order to define tree depth, the depth of a node must first be defined. The depth of the tree is consequently the depth of the deepest node.

Definition 2.8.
The *depth* of a node t, denoted by $\mathit{depth}(t)$ is

$$\mathit{depth}(t) = \begin{cases} 0, & \text{if t is the root,} \\ \mathit{depth}(t.\mathit{parent}) + 1, & \text{otherwise.} \end{cases}$$

Here $t.\mathit{parent}$ is the parent node of t.

Instead of searching for the deepest leaf, the tree depth can also be written simply as a relation to the height of the tree as follows:

Definition 2.9.
The *depth* of a tree T, denoted by $\mathit{depth}(T)$ is

$$\mathit{depth}(T) = \mathit{height}(T) - 1.$$

For a tree consisting of only the root, $N = n = 1$ and $d = 0$. For a tree to hold two keys, it must have one router, and $N = 3$ for $n = 2$, with $d = 1$. The tree has now two *levels*: the root is on the top level and the leaves on the next level. To include three keys, we must split one of the existing leaves into a router containing the previous leaf and the new leaf, as in the insert operation described in Section 2.3.

When comparing the values of N, n and d in Table 2.1 two interesting observations can be made: $N = 2n - 1$ and $d = \log_2 n$ for any $n = 2^k$, $(k = 0, 1, 2, \ldots)$.

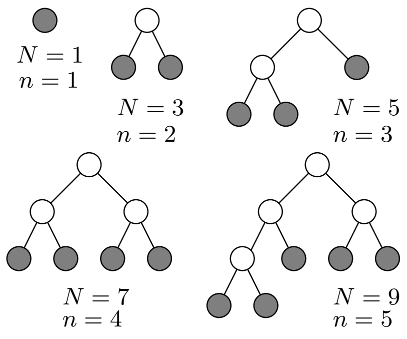
Locating any key in a node of depth t requires exactly t steps, that is $O(t)$ time. Thus for an optimal tree, with the preceding observations valid, locating any node in a tree with n keys can be done in $O(d) = O(\log_2 n)$ time. This is generalizable for all balanced EBSTs, instead of limiting to the optimal trees.

When $n = 3$, the tree has one router and two subtrees, one of them a single leaf and one a two-leaf subtree. The height of the one-leaf branch is one, and the height of its sibling 2. Generally, by Definition 2.4, the subtrees of a router t, where $\mathit{height}(t) = h$ need to be either both of the height $h-1$ or one of that height and the other $h-2$. The latter possibility is analyzed here, as the fully balanced tree has $\log n + 1$ height.

The size n_h of the subtree rooted at t can be expressed with the sizes of its subtrees:

TABLE 2.1: Number of nodes N, number of leaves n and tree depth d in an EBST.

N	1	3	5	7	9	11	...	15	...	31	...
n	1	2	3	4	5	6	...	8	...	16	...
d	0	1	2	2	3	3	...	3	...	4	...



$$n_h = n_{h-1} + n_{h-2}.$$

From this observation it is possible to derive an upper bound for the height of the tree. By adding one to the both sides of this equation and reorganizing, it takes the following form:

$$1 + n_h = (1 + n_{h-1}) + (1 + n_{h-2}).$$

Let $F_h = 1 + n_h$. The above equation then becomes $F_h = F_{h-1} + F_{h-2}$. This is the Fibonacci sequence, for which

$$F_h > \frac{\phi^h}{\sqrt{5}} - 1, \text{ where } \phi = \frac{1+\sqrt{5}}{2}.$$

By substituting this into $n_h = F_h - 1$, the tree size takes the form

$$G_h > \frac{\phi^{h+2}}{\sqrt{5}} - 2. \tag{2.1}$$

As the beginning setup was the case in which imbalance already existed, this can now be written for any tree size and solved for the tree height.

$$\begin{aligned}
n &\geq \frac{\phi^{h+2}}{\sqrt{5}} - 2 \\
n-2 &\geq \frac{\phi^{h+2}}{\sqrt{5}} \\
\log_\phi(n+2) &\geq \log_\phi\left(\frac{\phi^{h+2}}{\sqrt{5}}\right) \\
\log_\phi(n+2) &\geq h+2-\log_\phi(\sqrt{5}) \\
h &\leq 1.440\ \log(n+2) - 0.328
\end{aligned}$$

The latter analysis is the same that Adel'son-Vel'skiĭ and Landis [1] originally used in their article first presenting the AVL trees.

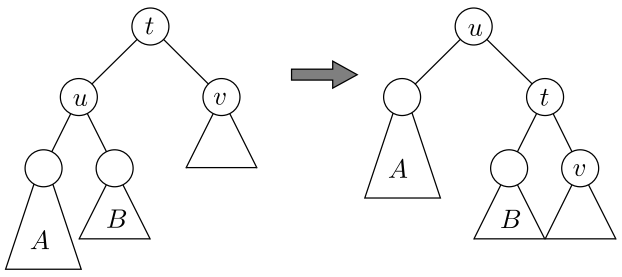
Theorem 2.1. For any balanced EBST, the height is $O(\log n)$. The upper bound of the tree height is approximately $1.44\ \log(n+2) - 0.328$

Due to the original setup, this limit is valid for any balanced EBST. The process of maintaining the tree in balance in order to guarantee this logarithmic height is *optimization* of the structure, and is discussed further in the following section.

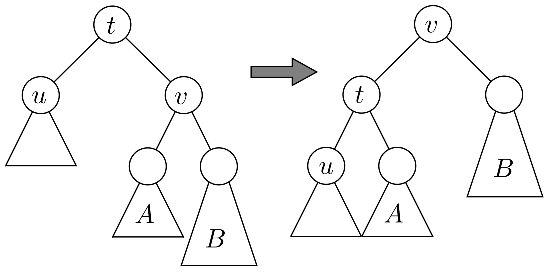
2.5 Standard Rotations

In binary search trees the balance conditions (like the one defined for EBSTs in Definition 2.6) are reached by simple transformations: single and double rotations on the index tree [26, 5]. These rotations aim to lift the largest subtree a bit higher, thus reducing the total height of the tree.

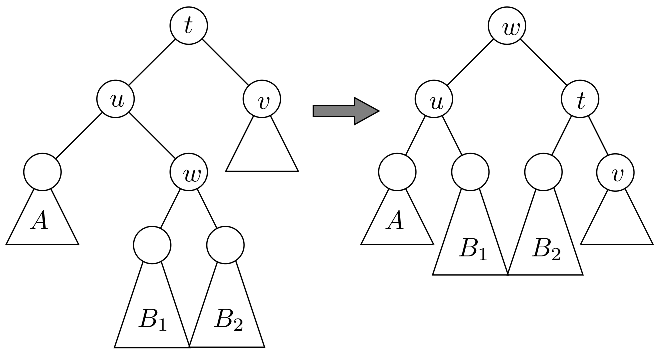
Single right rotation:



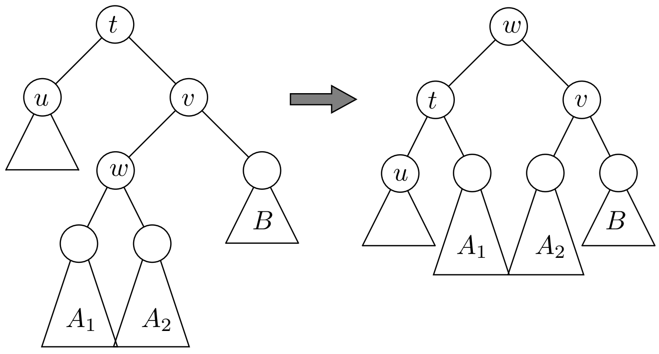
Single left rotation:



Double left-to-right rotation:



Double right-to-left rotation:



As can be seen in the illustrations [26, 33], all of these rotations preserve an EBST as another EBST. Each router still has exactly two children, as the subtrees remain the same. The proper rotation is chosen according to the heights of the subtrees causing the imbalance (i.e. having a height difference larger than one). The criteria is defined below for a node t with a left child u and right child v, nodes and subtrees named according to the above figures.

(1) $|\textit{height}(u) - \textit{height}(v)| \leq 1$:
Balance condition fulfilled

(2) $(\textit{height}(u) \geq \textit{height}(v) + 2) \wedge (\textit{height}(A) \geq \textit{height}(B))$:
A single right rotation at t

(3) $(\textit{height}(u) \geq \textit{height}(v) + 2) \wedge (\textit{height}(A) < \textit{height}(w))$:
A double left-to-right rotation at t

(4) $(\textit{height}(u) \leq \textit{height}(u) - 2) \wedge (\textit{height}(A) \geq \textit{height}(B))$:
A single left rotation at t

(5) $(\textit{height}(u) \leq \textit{height}(v) - 2) \wedge ((\textit{height}(B) < \textit{height}(w))$:
A double right-to-left rotation at t

It can also be observed from the selection criteria and the corresponding figures that no rotation will increase the height of a tree, but a smaller height is possible. These observation are recorded as a lemma for further reference:

Lemma 2.4. Performing any single standard rotation on an EBST T will result in another EBST T'. No nodes are added or removed in the rotation procedure: $|T| = |T'|$. Also *height*$(T') \leq$ *height*(T). Each rotation may reduce the height of the entire tree by at most one.

Nodes needing rotation is traditionally found by back-tracking from an updated node to the root and rebalancing as necessary (once or more in one or more nodes). Thus the complexity for a single update to a balanced tree consists of three phases:

(i) traversing to the target node in $O(\log n)$ time,

(ii) the update operation itself in constant time $O(1)$, and

(iii) back-tracking towards the root in $O(\log n)$ time.

In this thesis these phases in are called respectively *locate*, *update*, and *balance* phase. The straightforward implementation for single-update rebalancing is provided below, with the parent of the updated leaf given as a parameter. The balancing may stop as soon as one rotation has been performed or such a node is met whose height was not affected by the update. Until then, the procedure continues towards the root of the tree, whose parent is null.

```
procedure balance(node position) {
  while (position is not null ) {
    if (not in balance) {
      perform the proper rotation
      break ;
    }
    position.setHeight();
    if (height of the position did not change) {
      break ;
    }
    position ← position.parent;
  }
}
```

Besides the back-tracking, generally known as *bottom-up rebalancing*, also top-down strategies are used, where the balancing starts from the root, examining the balance criteria and proceeding downwards. Both approaches can create a *congestion problem*, meaning that service requests cannot be met as they arrive, but queuing is needed. This tends to happen particularly at the root, but also elsewhere in the tree [30].

Using such rotations is by no means the only strategy for eliminating imbalance in search trees. In past decades when databases and indices were smaller and the number of users and frequency of requests were not as high as today, also complete reconstruction was an applicable solution. Chang and Iyengar have compared the performance of three algorithms for reshaping the tree by adjusting the child and parent links of nodes in linear time $O(n)$ [16]. This would be an option in case if the index is not regularly reshaped and eventually the height differences grew so large that the balancing would also require near-linear time. It would be practical if there were sufficiently long pauses in the normal operation of the database, but little time for reshaping during service.

Andersson and Lai [4] use a balancing technique relying on partial reconstruction of the entire tree as the number of insertions since the latest reconstruction equals one half of the tree size. They amortize the cost of this rebuilding over the upcoming $\Theta(n)$ updates, giving an amortized cost $O(1)$ per each insertion, proved with a potential function. Methods of partial reconstruction or merging are also used in place of rotations [26].

Chapter 3

Problematics of Traditional Indexing

This chapter draws together the traditional approach to database indexing presented in the previous chapter to discuss the inherent performance problems. The practice of single key updates and the involved balancing is discussed in Section 3.1. The problems found are closely related to the concurrency of an index structure, generally explained in Section 3.2.

3.1 Single Updates with Strictly Coupled Balancing

As seen from the previous definitions of the index operations, they deal with one key at a time. For each keyword from a new document to be added into an electric library, the tree is traversed from the root to the proper position leaf, where the keyword is inserted or the existing leaf of that keyword appended to point to the new document as well. This gives $\Theta(\log n)$ link traverses per key in a balanced EBST, and if the document contains m keywords, $\Theta(m \log n)$ link traverses for inserting the entire list of keywords for the document. Also, as each key is added, $\Theta(m)$ leaf splits are required to include the new leaves in the index. The tree is rebalanced $\Theta(m)$ times, once per each inserted keyword.

In a simple application each of these key inserts is assigned its own process. In a concurrent database, as most practical databases nowadays are, several processes traverse and operate on an index simultaneously. Moving the nodes around upon insertions and deletions as well as rebalancing rotations will interfere with making queries and updates around the area in which the modification takes place.

Insert and delete operations add or remove nodes respectively, which causes the tree to change shape. After an update the tree is potentially out of balance. A major question in database indexing has been when and how to balance the tree. A traditional choice is making the rebalancing operations immediately after an insert or delete operation, binding the update and balance closely together:

```
procedure insertEntry(data data, node tree) {
  router position ← insert(data, tree);

  if (position is not null ) {
    // THE POSITION ITSELF IS BY DEFINITION IN BALANCE
    balance(position.getParent());
  }
}
```

```
procedure deleteEntry(int key, node tree) : boolean {
  router position ← delete(key, tree);

  if (position is not null ) {
    // THE RETURNED NODE MAY ALREADY BE IMBALANCED
    balance(position);
  }
  return success;
}
```

This method of operation is called *strict balancing*. Indexing systems that use strict balancing are easier to analyze for correctness and complexity than solutions that do not wrap the update and consequent balancing together as the area influenced by the update is not subject to further modification. It does however cause problems for the processes running simultaneously with it. The practice of dealing with such problems is called *concurrency control* and is explained on a general level in the next section.

3.2 Concurrency Control

For developing a better understanding of the application areas of indices, especially with a large key space, a WWW search engine is a comprehensive example. The amount of data available through the World Wide Web is increasing rapidly, as well as the number of Web users. The number of documents to index on e.g. a search engine is gigantic; On September 15, 2000, the Google search engine[1] reported to have 1060 million web pages indexed, 560 million of them with full-text indexing. Back in 1997 when search engine databases covered about a 100 million documents, another search engine Altavista[2] already claimed to serve approximately 20 million search requests per day. The estimate for the number of queries per day the year 2000 is as high as hundreds of millions [13].

Due to the dynamic nature of the World Wide Web, new documents need to be inserted constantly while search requests come as a never-ending flow [45]. The necessary rate for a search engine to handle queries is from hundreds to thousands of requests per second. With indices of this magnitude and such frequency of service requests, it is clear that the highest possible degree of concurrency is not merely an option, but a strong, unavoidable demand. The research dealing with such large amounts of data is known as *data warehousing*, necessary for example in the Human Genome Project and space research besides

[1]*http://www.google.com*
[2]*http://www.altavista.com*

WWW searching [55].

Normally an index is being used by several independent processes, some of them *readers* and some *writers*. The reader processes do not make any modifications to the index; these are the search requests on a search engine, only reading the database. Writers have the purpose of modifying the index by deleting a single node or inserting one.

In a search engine each new web page found need to be inserted into the index, and outdated links (pointing to nonexistent web pages) need to be deleted. The search engine checks the current state of the web by 'crawling', i.e. automatic following of links and recording page information, which is done as a constant background process to serving requests. An early Google prototype used three crawlers each using about 300 connections [13].

Discussing database updates, a brief overview into concurrency control and locking in necessary. As the exact implementation and scope of locking operations is not a focus area in this thesis, the discussion will be solely introductory.

A delete operation deals with the linkage and contents of the node to be deleted and at least the parent of the deleted node. To prevent other processes from intervening with the deletions, it must *write-lock* the leaf and its parent router under operation. No other process may then access those nodes. Before write-locking a node, the process must make sure that no other process is currently operating there. A reader notifies of its whereabouts with *read-locks*, which do not prevent other readers from access, but force writers to wait until they have finished with the node.

Also other types of locks have been presented in the literature to avoid some of the difficulties of straightforward locking, as deadlocks and starvation, preventing the system from fair operation. Such are e.g. the *might-write* locks and *lock-coupling* strategies. Details of the waiting and lock-handling is not an issue here, but a discussion on a similar situation is given by Nurmi and Soisalon-Soininen for another class of binary search trees [39].

As a delete operation removes a node from the tree, it must also lock the parent of the removed leaf, as otherwise another process might follow the now broken child-link from the parent. And as the index tree is chosen to be an external binary search tree, the parent router with one child is invalid and must be merged with its own parent as in Figure 2.4. Thus each delete operation must lock the leaf to delete, its parent and its grandparent, thus blocking access to two to five other leaves if the tree was initially in balance. This is again a naive implementation, quite easily improved by more sophisticated locking.

Similarly an insert operation needs to lock the location leaf and its parent, as the location leaf is about to be replaced by a new router and moved under the router together with the leaf to be inserted as shown in Figure 2.3. This blocks access to one or two other leaves in case the tree was initially in balance.

When a rotation is about to be performed in a certain node, the nodes involved in the rotation are all subject to change of location. Thus it is not safe for other processes to concurrently operate on those nodes that are being re-linked. In simple implementations of concurrent search trees this usually implies that all nodes that an update visits during the locate phase need to be locked, as all of them might require rotations if the balance conditions are broken [29].

As a consequence, in order to perform a rotation on a node that is not in balance, the balance process must wait until the subtree is clear of other writers and take a write-lock on the node. As imbalance may occur as high as the root, it may be inevitable that the entire index is momentarily out of use due to the rebalance. Thus for each update operation, not only the directly updated part is locked, but also larger parts of the index are in danger of being out of use. The locate phase is the only one not requiring locking.

Chapter 4

Known Improvements

The aim of this chapter is to provide an overview on two current improvement strategies of index performance. Section 4.1 covers the genre of relaxed balancing, as Section 4.2 introduces the concept of group updates. The discussion introduces the research done on these areas, the principals employed, and the results achieved. The algorithm of Chapter 5 is also outlined as similar approaches are presented.

As the amount of digital data is larger than ever and quickly growing, much of the former database management has turned into data warehousing due to the massive amount of tuples and the increased size of a single database entry (considering e.g. video streams and other types of multimedia). Also file management in corporations has been turning into data mining with index-based, often distributed, directories. Dealing with such amounts of data with the traditional means of indexing is unpleasantly inefficient, especially in applications where the users of the database are numerous.

It has become popular to provide customers access to company product databases over the Internet, which creates a potential user population of a size never even anticipated during the early times of computer-operated data management. In e-commerce business it is unacceptable to keep customers waiting for database queries to be served. Earlier this year the web-store *Boo.com* went bankrupt as their online product catalog was too slow for the customers to browse fluently. It is evident that the methods and structures of database indexing needs development to avoid the difficulties and computational complexity discussed in the previous chapters.

4.1 Relaxed Balancing

As already pointed out, commonly more than one client is using an index structure concurrently. If the service requests (searches and updates) occur in bursts, the danger of congestion increases and the response time drops below the desirable level [30]. Thus it is not feasible that by serving one, the database manager would risk the service of others. By performing balance operations it significantly increases the risk of delay. Normally a balance operation proceeds from an updated leaf towards the root, checking for the balance conditions and performing the standard rotations defined earlier in Section 2.5 as necessary. The balancing stops after a rotation has been performed or such a node is found whose height has not changed due to the update. Ultimately the balancing will end at the root.

This is a tedious operation as a whole, locking off subtrees often at quite a high level, even at the root. Performing the balancing procedure after each update operation as traditional, the total workload and interference with other processes grows to be significant. It would clearly be proper to put off the balancing until a conveniently quiet moment [24]. Thus it was suggested by Guibas and Sedgewick [18] that the rebalancing phase should be separate from the update. Originally they had worked with a scheme that performed both update and balance modifications on the way down from the root to the position leaf. They however found that a separate *balancer* could work locally and take care of the inherent imbalance.

It is essential that the newly inserted data is available for readers to use as soon as possible and that deleted data is no longer available in the index[1], but the harm done in delaying the rebalance is usually not significant. Also, if the rebalance is done at some suitable point of time, there is no need to complete the rebalancing at once. It may be worked on once in a while, progressing slowly towards the optimal tree shape, allowing search operations to interleave. Some sophisticated solutions are also capable of serving updates while the balancing is waiting on the background. The balancing of the algorithm in this thesis is generalizable into such a solution by changing the structure holding the nodes awaiting balancing.

As seen in Section 2.5, each balancing rotation only affects a small number of nodes at a time. Hence the rebalancing operation is well *localized* and can easily be performed while concurrently query processes work in other parts of the tree. If the balancing process could stop after one rotation (or some other constant time task) and allow queries to progress to the previously locked area, it would not need to block any portion of a tree for a significant period of time [32]. This would spread the balancing overhead more evenly in time and on the tree [24].

4.1.1 Effects of Uncoupling Update and Balance Phases

The evident approach is that the first two phases of an index operation, locate and update, are tightly coupled, but the balance phase will be delayed until the service requests become sparser. The reason for binding locate and update phases closely together is to provide the updated information in the index as quickly as possible for future transactions to use [34].

If it is not likely that the index application will have free time at any point to perform the balancing, the tree may get so significantly out of balance that it effects the performance. For these cases, some ultimate limit on how much imbalance will be tolerated needs to be agreed upon. The magnitude of such a limit is quite context-sensitive.

To examine the effect of imbalance in an EBST, Figure 4.1 is constructed from the following observations: In an optimal tree the height of the root, defined in Definition 2.5 is $1 + \log_2 n$, where n is the number of leaves. In the worst linear case the root height is n. Flajolet and Odlyzko showed in 1980 that the average height of binary trees of all shapes is $2\sqrt{\pi r}$, where r is the number of internal nodes, i.e. routers [17]. The result applies as is for random trees and sufficiently well for a random tree with random insertions for a general discussion [7].

For both a balanced tree and a worst-case tree, the correspondence between routers and leaves is simply $r = n - 1$ [14] (see e.g. Figure 2.5 for illustration). It can be safely deduced that $r = n - 1$ by observing any EBST; there certainly cannot be more routers than

[1]The minimum requirement is that the leaves that were requested to be removed are marked as invalid.

leaves in a full EBST.

Lemma 4.5. In an EBST T with n leaves the number of routers is $r = n - 1$ and thus the total number of nodes in T is $N = n + r = n + n - 1 = 2n - 1$.

The the values of Figure 4.1 are calculated with Lemma 4.5 and the preceding height observations. For the average height $2\lfloor\sqrt{\pi(n-1)}\rfloor$, an integer value is obtained with the floor function; note that the estimates are unrealistically high with small values of n. Also the AVL height defined in Theorem 2.1 as $1.44\log(n+2) - 0.328$ is drawn, as a perfectly balanced EBST is quite rare amongst all EBSTs.

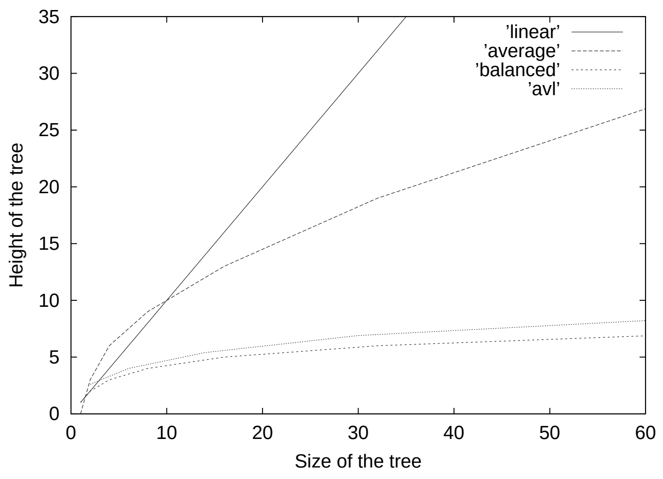


FIGURE 4.1: The tree height for perfectly balanced, AVL, average, and worst-case linear trees with n leaves.

As the imbalance caused by an update operation is local, affecting negatively only the search time of the leaf just inserted, the performance is unhurt elsewhere in the tree. What comes to deletions, the tree depth elsewhere in the tree is unchanged, even though no longer necessarily optimal. The access time of those leaves that changed place due to the deletion improved, as they rose closer to the root. Thus, as the average height grows moderately in comparison to the worst case, and as all shapes are as likely outcomes for a random sequence of insert operations with reasonable accuracy, the temporary imbalance will not cause significant damage in access efficiency. Also, the worst-case trees are very rare, even in comparison to the likelihood of a perfectly balanced tree [26].

Larsen [31] argues that any relaxed balancing will be beneficial due to the locality of the operations, almost always taking place near the leaf-level and the fact that not very many operations are needed. This in addition with evidence of not harming the access times of unmodified nodes (although delaying their optimization), relaxed balance certainly seems worthwhile to implement for any index.

4.1.2 Trees with Relaxed Balance

This practice of uncoupling the balancing from the update is commonly known as *relaxed balancing*. Kessels [24] uses the name *on-the-fly optimization* as in on-the-fly garbage collection, which is also done when the time seems 'right' as a background process. The basic idea of relaxed balancing is to speed up the processing of requests risking the logarithmic access time of the structure [30].

Research on the performance effects of relaxed balancing has been conducted by many during the past few decades.[2] The initial discussion concentrated on a class of balanced binary trees called *red-black trees* by Guibas and Sedgewick in [18]. Further work was done by Tarjan [51] in developing bottom-up methods with $O(1)$ structural changes per update, as the original propositions required $O(\log n)$ such modifications.

Next, the focus was relaxing AVL trees and binary search trees in general. This was speculated already by Kessels [24] during the early eighties and a relaxed version was developed a few years later by Nurmi, Soisalon-Soininen and Wood [40]. The development was later taken further by the same authors [41], augmented with discussion on concurrency control.

Their initial solution of [40] was criticized for restrictivity and difficulty of proof by Larsen [29] and Boyar [12], who suggested improvements on it. Larsen [29] showed that update operations of a tree require $O(\log N)$ rebalancing operations per update, no matter how they are interleaved. Here N must naturally be the maximum number of nodes in the tree during the runtime of the operation sequence. Nurmi, Soisalon-Soininen and Wood [40] also extended B-trees introduced by Bayer and McCreight [9] for relaxed balancing. That solution was modified again by Larsen [31] in order to decrease complexity.

Kessels [24] was the first to use tag values stored in the nodes to contain information on whether the node is subject to imbalance and should be examined during the rebalancing process. Soisalon-Soininen and Widmayer [47] defined these tags as *height values*, therefore defining *height-valued* binary search trees. They showed that for a height-valued tree of size n with balance conflicts, $O(n \log n)$ balancing steps (simple calculations of $O(1)$ time or the standard rotations presented in Section 2.5) suffice to eliminate all imbalance.

Larsen [30] has shown it to be possible to perform relaxed balancing with the standard rotations with constant amortized complexity by using weight tags instead of height values in the nodes. He has also shown that for B-trees, assuming size being at least twice the degree, amortized constant rebalancing can also be defined [31]. In the same research he proved that the amount of rebalancing necessary in a relaxed B-tree decreases exponentially when progressing towards the root.

Another view in balancing binary trees was taken by Andersson *et al.* [3], where a class of SBB-trees[3] is defined through B-trees with maintenance operations based on splitting, joining, and single rotations were defined. They showed the necessary restructuring to be constant even in the worst case.

A relaxed version of red-black trees known as *chromatic trees* was defined by Nurmi and Soisalon-Soininen [38, 39] as a relaxed extension to the update operations requiring $O(1)$ rotations defined by Tarjan [51]. The balancing of a chromatic tree is based on locating red-black violations as in the original trees, but is not coupled with the update and hence has a looser definition for acceptable tree states. The less strict balance conditions are generally called *weak* conditions. Thus the information on which nodes to balance is left by the update operation into the tree, as in Kessels' binary search trees. The benefit of such a marking procedure is the possibility of running the rebalancing as a shadow process on

[2]For a regularly updated list of researchers, see a web site maintained by Kim S. Larsen at the University of Southern Denmark at *http://www.imada.sdu.dk/˜kslarsen/RelBal*.

[3]The SBB-trees in fact define the exact the same class of trees than EBSTs.

the background as garbage collection without needing to synchronize any other data but the index tree.

The approach selected in this thesis follows the outline drawn by Pollari-Malmi, Soisalon-Soininen and Ylönen [44]: the nodes that are candidates for balancing rotations are recorded into a control structure during the locate phase of an update. The selection of these candidates is discussed in detail in Chapter 5. The shadow process schema given for the rebalancing by Nurmi and Soisalon-Soininen [38] should be generalizable to the trees of this thesis, given that the locking procedures were defined. This however would require more attention than is reasonable to include in a single thesis.

One important question is of course, how to allow relaxed balancing and still maintain control over the tree size [29]. No definite answer is given in this algorithm; the balancer may be triggered at an instance found proper for the application using the index. The trigger for the balance process may be some of the following scenarios:

- The database is not serving any clients, which frees the index tree to the rebalancer. The rebalancer may stop as soon as a client needs to access the index, and continue later according to the control structure.
- The database manager keeps track of the largest imbalance and triggers the rebalancer ’manually’ as a predefined limit is exceeded, thus preserving a near-logarithmic locate-complexity at all times.
- A predefined number of update operations has been performed or the time since the last rebalancing exceeds a given bound.

Also many other balancing criteria are possible. For the analytical discussion of an algorithm, no specific criterion needs to be chosen.

4.2 Group Update

In addition to relaxed balancing, also another approach has been taken to improve the performance of index trees. Upon insertion of a new article on a publication database with n being some millions, one generally must insert multiple keywords for the new article. Instead of inserting these some thousands m of keys one by one, using $O(m \log n)$ time, a *batch* or *group* is created from these keys and sorted in $O(m \log m)$ time. The resulting group is inserted as an entity. As the keys are sorted, duplicates may also be merged together to save time.

Database operations tend to commute on many application areas. This means that several requests handle the same keys, thus allowing to fetch these keys only once. This behavior has also been taken advantage of in distributed indexing [23]. The same idea is behind the design of level-linked tree structures and finger systems linking consequent leaves to each other, trying to benefit from correlation between tree accesses, in addition to often easing the locking of the tree [15, 21].

There is not reason for a group of keys to consist of only keys to insert; also keys that need to be deleted or simply searched for may form a batch together with the keys to insert [34].

This way we only need to traverse the tree once per the entire set of operations. This form of operation is commonly known as *group update*, even though also group searches are normally included in the method.

The methods of *group update* are various. An approach chosen by Malmi [33] was to locate the position for the first key in the sorted sequence, and instead of continuing from the root for the next key, he continued from the current location, thus binding the cost of the locate step to the separation distances of the position leaves of subgroups. One of the earliest discussions of this approach was provided by Tsakalidis in 1985 [52]. This principle has also been used in B-trees [49].

In this thesis I have selected a *divide and conquer* -approach, where the entire list of m keys is initially taken to the root and forwarded recursively according to the minimum value in the key list and the routing value of the root. The procedure will be described in detail in Chapter 5. A similar procedure with B-trees is used by Pollari-Malmi, Soisalon-Soininen, and Ylönen [44]. They have chosen to collect the batch of updates into a differential, main-memory index which will at a proper point of time be merged with the main index residing on disk, to create large enough batches. Allowing access to the differential index alongside the main index brings the updates into use as quickly as possible.

This is of particular interest upon index reconstruction. In case of a serious database failure the entire index must be rebuilt from the original data, e.g. text documents. In systems that use a main-memory index that is only occasionally stored on disk these reconstructions are not conveniently rare. The reconstruction procedure naturally reduces into a long series of insert operations, during which it is desirable to allow search operations to carry on simultaneously, using as big of a portion of the index as possible.

As my goal is not to provide a ready-to-use industrial strength implementation for a database index, I use a more simplified model here as well: the batch of keys waiting to be inserted as a group is not accessible until merged with the main index, but the insertion is to be done as soon as a batch arrives. One possible batch in document indexing would be the key words of one document. This practice with B-trees is available in [49].

The practice of group operations has been found useful also outside indexing, in searching sequential and tree-structured files, researched for example by Shneiderman and Goodman [46], and importantly Lang, Driscoll and Lou [28]. The latter have shown that using a batch search, significant savings in the number of disk accesses can be achieved, and they strongly recommend the method to be employed whenever possible.

4.2.1 Required Modifications on an Update

As the keys of an index tree are located at existing leaves, the leaves in which a locate procedure ends at are subject to modification.

Evidently the locate phase causes no structural changes on the tree. Figure 4.2 shows the path traversed for keys that have the same position leaf 47. Normally, there are more than one position leaf, which will be visited in from left to right by the recursive procedure.

In deletion all the keys that located in a leaf with the same key value as theirs, need to be removed. This may result in the removal of entire subtrees.

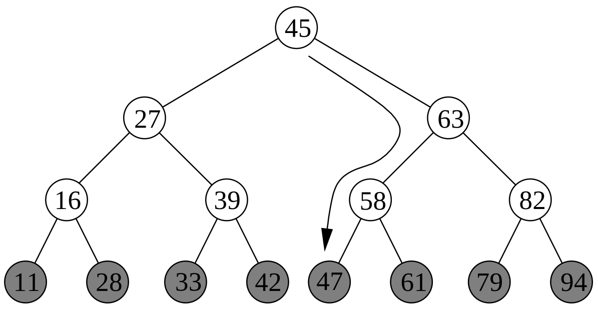


FIGURE 4.2: Performing *locate* with keys in the interval $(46, 57)$.

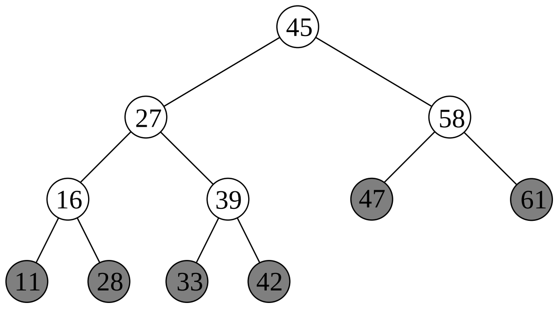


FIGURE 4.3: The tree of Figure 4.2 after deleting the group of keys $\{79, 94\}$.

Only one if any of the located keys will be a match to the key stores in the position leaf. The subtree to remove can be limited by locating the lowest common ancestors of the leaves to remove. A usable strategy is also not to remove the leaves, but merely to mark them as invalid. Most information systems that require an index have significantly more insertions than deletions, and these invalid leaves would be sooner or later replaced by new, valid leaves. Either way, performing a group deletion on an EBST as the tree in Figures 4.2 and 4.3 will result in another EBST, with every router having exactly two children. In the marking scenario some of the leaves would simple be invalid, replaced upon insertion, failing all searches and deletions positioning there.

In an insert operation more than one key may require insertion into the same position. If there is only one new key, the same kind of a split operation than used earlier in single insert is valid. If there are more new keys, the common approach is to merge the key list with the position leaf, generate a balanced tree from these keys and replace the position leaf with the root of the newly constructed subtree.

Tree generation from a sorted list is possible in linear time[4], and the insertion of the position leaf key into the key list can be done in logarithmic time.

Again, the group insert evidently preserves an EBST as a EBST, properly replacing a leaf with a valid, newly generated subtree. The resulting tree from the insertions or deletions is not necessarily in balance and thus needs either strict rebalancing directly after the update or relaxed balancing at a later point of time.

[4]A discussion on linear-time tree generation is given by Alonso, Rémy and Schott [2], but turning a n-item sorted list into an EBST of size $2n$ in $O(n)$ time is quite simple e.g. by choosing the median as the subtree root when the list length is known.

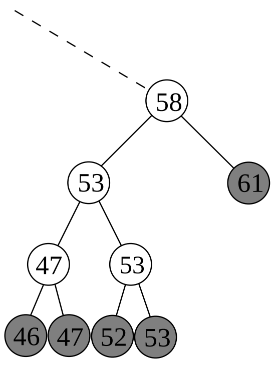


FIGURE 4.4: The subtree rooted at 58 from the tree in Figure 4.3 after inserting the group of keys $\{46, 52, 52\}$ at position leaf 47.

In order to balance a group updates tree with the standard rotations, it must be noted that unlike in the single strict balancing, one rotation may not remove all imbalance, but more may be needed at one single location. Thus it is necessary to balance in each router until it is known to be in balance, instead of rotating only once.

4.2.2 Concurrency and Group Updates

Although the details concurrency control are outside the scope of this thesis, a few words on the concurrency of group update algorithms are nevertheless needed. Basically, as the explanation of the previous section showed, a group insert does not need any more locks on a tree than a single insert in the same position leaf. Instead of the split, the position leaf is directly replaced by a new subtree. When considering the amount of locking per key, a series of m single inserts in one position leaf will lock its parent router m times. The group insert of m keys in one node will only lock the parent once, avoiding significant overhead in locking.

For the deletion, instead of locking the modified router (the grandparent of the position node) m times as in single deletions, the group update algorithm only needs to lock the deleted branch once, locking off a larger portion of the tree, but for a shorter period of time, reducing computational overhead even though enlarging the blocked area momentarily.

By using an AVL-tree variant, Soisalon-Soininen and Widmayer [48] have constructed an algorithm for group updates that entirely eliminates the problematics of concurrency control, as their solution requires no locking whatsoever. Their definition also guarantees the recoverability of the updates, meaning that upon an unexpected interruption in the database system, the entire group update can be aborted as a whole, hence protecting the database from ill-defined states. Their solution uses an index stored in main memory and the creation of partial trees. For B-trees, a detailed discussion on the concurrency control involving group updates is given by Pollari-Malmi, Soisalon-Soininen and Ylönen [44].

Chapter 5

The Algorithm

This chapter describes the contribution of this thesis, an algorithm combining group update and relaxed balancing for EBSTs. At first in Section 5.1, the class of trees used in the algorithm are briefly summarized. Then each phase of the algorithm is presented in its own section both verbally and as pseudo-code. Preliminary analysis on the computational complexity is also given, but the exact analysis is not provided until Chapter 6.

Combinations of the two presented approaches, group update and relaxed balancing, have been successfully developed during the past few years. Hanke and Soisalon-Soininen have designed a group update algorithm for red-black trees [20] by combining the batch approach and the relaxed-balance the chromatic trees. Red-black trees were a justified selection, as they generally require less rebalancing work than AVL trees.

In their version, instead of the colors red and black, nodes are given weights and each path connecting a leaf to the root is required to have the same weight in order for the tree to be in balance. The group insertions notify of the modification they make in the form of such weight values in the inserted nodes that break the balance condition. Also here the burden of storing and calculating weights complicates implementation and analysis.

A more general overview on combining the two approaches is given by Malmi *et al.* in [35]. They define a group insertion for any external search tree, in which small balanced subtrees are generated from sorted key sequences and inserted at proper position nodes. After the attachment of such a subtree into the index tree, they use the standard rotations for relaxed rebalancing. This approach has been adopted also to the algorithm of this thesis. The locate phase resembles highly of that used by Malmi [33].

The algorithm presented here also combines these approaches for external binary search trees (EBST). As the others, it is also divided in tree steps:

Locate : Locating the appropriate position nodes for a given set of key material and recording the nodes subject to imbalance.

Update : Performing the requested operation at each of the position leaves that were assigned key material in step one.

Balance : Balancing the tree according to the imbalance records constructed in step one. The record structure is followed in predefined, orderly manner to maintain its validity.

Steps one and two are always processed as a couple, whereas the third step may take place at a later point of time. In further discussion the couple of the first two steps will be called *index operation* or operation for short. The balancing process will be called the *rebalancer* as Guibas and Sedgewick originally named it [18].

As mentioned earlier, details of concurrency control are not included in the following discussion. Should two processes (either normal index operations or rebalancers) operate on the index structure simultaneously, they must involve a disjoint set of nodes. This is analytically equivalent to accessing the index sequentially, which eliminates the need to address concurrency issues without the loss of generality [29].

5.1 Summary of the EBST

I will review the properties of EBST briefly to ease following of the forth-coming analysis. The index tree used with the algorithm defined in this chapter is a *full* binary search tree, each internal node having exactly two children. In an external tree i.e. *leaf-oriented*, tree the data is stored only in the leaves and the inner nodes act as routers to this data.

The router values are either current or former leaf key values. The tree order is such that the keys in the left subtree of a router all have smaller values than the router value, and the keys in the right subtree are greater than or equal to the router value. The keys used are simplified to integer values, but any other key type processed in constant time would make a valid substitute. The supported operations are *search*, *insert*, and *delete*.

No other information but the parent and child links, the actual height and the key value (routing value for inner nodes) is stored in the tree. Additional balancing information is recorded on a separate structure to keep the tree simple. Leaves naturally do not contain child links.

5.2 Locate Phase

The key material of the group update is either stored in a sorted order or sorted before beginning the locate procedure.[1] In the model implementation, an array-like list is used for simplicity, but any other structure sortable and divisible on a specified value could be used. To simplify the discussion, the group of key material is referred to as a list.

The list is divided into subgroups located each in their own position leaf by traversing the tree recursively. The traversing begins at the root, checking whether the first key in the list is smaller than the root's router value. If so, the left subtree needs to be traversed. The router value of the root is sent to the left subtree together with the list, in order for the locate process to know what is the upper limit for values located there. When the locate process arrives at a leaf, it removes all those keys from the beginning of the list that are smaller than the upper limit. These removed keys all share the same position leaf and are consequently processed together.

During the traversing, *branching nodes* are marked and stored together with position leaves

[1]An interesting strategy could be using the root value of the tree as the pivot element for quicksort, and storing the data while recursively traversing the tree.

to be used later for rebalancing purposes. Of course, this recording can be skipped if it is known whether an update or simply a search is being performed. The procedure for a router can be written as follows, treating the key list as a queue. The queue uses a `peek` method to retrieve the top value without removing it.

```
procedure locate(node current, list keys, int limit) : boolean {
  boolean wentLeft, wentRight ← false ;
  node temp;
  node store;
  branch ← false ;

  if (keys is not empty and key.peek() < current.key) {
    wentLeft ← locate(current.left, keys, current.key);
  }
  if (keys is not empty and key.peek() < limit) {
    wentRight ← locate(current.right, keys, limit);
  }
  if (wentLeft and wentRight) {
    // MARK THIS ROUTER AS A BRANCHING NODE
    current.branch ← true ;
    // STORE IT INTO THE BALANCING QUEUE
    queue(current);
  }
  return (wentLeft or wentRight);
}
```

On the leaf level the key list changes, so the first key is not necessarily the same when entering the right subtree than it was upon entering the left subtree. To understand how this happens, I also add the pseudo-code for the locate procedure on the leaf level; now again the list is a queue, with a `pop`-operation for removing the first item (and returning it), and an `append`-operation for adding to the end of the list. An illustration is provided as an example later with the discussion on balancing.

```
procedure locate(node current, list keys, int limit) {
  list subgroup;

  while (keys is not empty and keys.peek() < limit) {
    // STORE THE SUBGROUP WITH THE POSITION LEAF
    subgroup.append(keys.pop());
  }
  if (subgroup is not empty ) {
    return process(current, subgroup);
  }
}
```

The processing takes place in the update phase; simple data retrieval for searches, more complicated manipulation for insertions and deletions. The `process` method also includes the queuing of position leaves. If the entire operation at the position leaf fails, the method returns `false`, and if it succeeds for any of the keys in the subgroup, it returns `true`.

5.3 Update Phase

The update phase performs one of the basic index operations. The first, the update phase of a search operation, is trivial. It is not an actual update, but a mere retrieval of the contents of the position leaf. The experimental implemented model described in detail in Section 7.3 only announces which of the requested keys were found at the index.

The actual update operations, insert and delete, modify the index tree. The insert operation discussed here excludes those keys that are already in the tree, but the modification to the second real-life scenario — combination of the data stored with the same key into an entity — is easily adapted from the presented insertion procedure. Delete ignores any keys not found in the index tree (i.e. the deletion of those keys fails) and removes the rest. Due to their different nature, insertion and deletion are addressed separately. Insertion is discussed in detail and presented in pseudo-code, as it has a straightforward group update implementation, but the more controversial deletion is handled on a more general level.

5.3.1 Insertion

The principles of the group insert are the same as generally described in Section 4.2.1. The keys that are positioned on the same leaf are initially a sorted list. If the key of the position leaf is included, it is replaced by the existing leaf key. In the model implementation, the keys are simply integer values, but in real-life applications, a key to insert would also contain or refer to the updated data, which would be replaced here as well. If the leaf key is not in the key list, it is added to the proper position. As mentioned earlier, adding an element on its place on a sorted list can be done in $O(\log m)$ time, where m is the size of the list.

The contents of the key list are then used to construct a balanced EBST in linear time $O(m + 1) = O(m)$. In the model implementation this is achieved by taking the median value of the list as the root and constructing the left subtree recursively from the beginning of the list and the right subtree from the rest of the list, including the median.

The last step is to replace the position leaf by the root of the newly constructed subtree, which can be done in $O(1)$ time. In the model implementation the only operations needed are the setting of the parent router link and the corresponding child link. After this, the updated position leaf is queued for rebalancing.

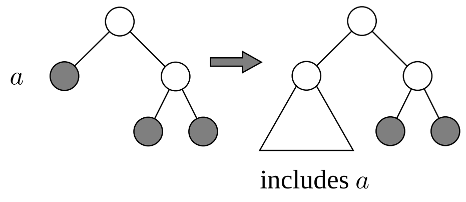


FIGURE 5.1: Group insertion by attaching a subtree.

Thus the update phase of a group insertion consists of the following three parts with total complexity of $O(\log m) + O(m) + O(1) = O(m)$, where m is the size of the group. This is the best possible upper bound, as each key needs to be assigned a leaf, which will inevitably require $O(m)$ time for m leaves.

To summarize group insert algorithm, a list of the necessary operations is provided with the corresponding pseudo-code:

(i) Insert the key of the position leaf into the key list.

(ii) Construct a balanced EBST from the key list.

(iii) Replace the position leaf by the root of the subtree.

(iv) Add the parent to the balancing queue.

```
procedure insert(leaf position, list keys) {
  node subtree;

  if (keys contains position.key) {
    // ONE OF THE INSERTIONS FAIL
    if (keys.size is 1) {
    // THERE ARE NO OTHER KEYS TO INSERT
    return false ;
  } else {
    replace the occurrence in the list
  }
  // BUILD A TREE FROM THE INSERTED SUBGROUP
  subtree = construct(position, keys);
  // PLACE IT WHERE THE POSITION LEAF WAS
  position.getParent().replaceChild(position, subtree);
  // APPEND THE MODIFIED NODE INTO THE BALANCE QUEUE.
  queue(position);
}
```

5.3.2 Deletion

For the deletion, the first step is to see whether the key material of a position leaf contains the key of the position leaf. If it does not, then the deletions of all the keys in the key list have failed. If it is, the position leaf is removed. (If there were more than one key positioned there, the other deletions failed.)

The removal of the leaf leads into the replacement of its parent, as in an EBST each router is required to have exactly two children. The parent router is thus replaced by the sibling of the removed leaf, as earlier shown in Figure 2.4. This is all done in constant time $O(1)$.

To provide a more efficient group deletion, it must be examined whether the sibling nodes of the node to remove are also being deleted. If so, it may be possible to delete an entire subtree starting from a specified router. Hence instead of directly beginning to replace the parent, the usual group deletion algorithms look for the lowest common ancestors of the keys to delete. After all the keys on the deletion list have been handled, the replacements are processed from the ancestor list, with the possibility of significant savings.

However, in most uses of indices, especially in connection with databases, deletes are quite rare, and it is hardly necessary to go through such trouble removing those leaves. It is often

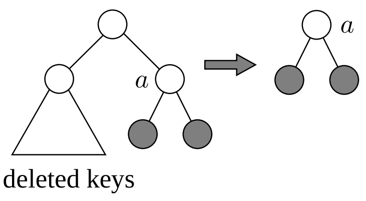


FIGURE 5.2: Group deletion by removing a subtree.

sufficient simply to mark those leaves as invalid; this occupies only one bit per key, and it is quite likely that an insert in the same position will eventually replace the redundant node. Thus it is not necessarily useful to construct a specific algorithm for deletions. One could be added to this algorithm later, but simply adding the invalid mark bit to the leaves (where the routers have the branch-bit) and marking that after locating the nodes is sufficient. The leaves marked as invalid could be later replaced by insertions or removed later as a larger batch by the means of finding the lowest common ancestor, i.e. as an actual group deletion.

5.4 Balance Phase

The balancing is performed entirely with the standard rotations defined in Section 2.5. It is organized bottom-up, gradually proceeding towards the root, possibly pausing at times and allowing search operations. After certain double rotations, the algorithm needs to proceed downwards until balance is ensured, after which the upward climb is resumed.

The current control structure being a list (as the key list in the earlier two phases, with operations `peek`, `pop`, and `append`), it is not trivially updated with interleaving insertions, but a similar tree structure would provide for that functionality as well. For analytical and experimental reasons, this algorithm restricts the interleaving to searching.

When the balancing finally begins, it will traverse the control structure in an orderly fashion, removing the imbalance from one subtree at a time. There is no objection to running more than one operations and balance processes simultaneously, as long as proper locking is implemented on the index tree and the control structure. The order is straightforward in the array; always popping the topmost node and balancing from there until a branching node is met, balance achieved, or the root comes along. The reason for accessing the list in order is the visiting order of the nodes in the locate phase; the recursing is done in post-order, preserving the preconditions for balancing branching points.

Figure 5.3 illustrates an example scenario for generating the balance queue. The dotted line and the arrows indicate the path followed by the locate procedure with keys belonging to the position leaves A, B, D, F, G and H. As the locate has to visit both subtrees of the router C, it becomes a branching point after the visit to the right child. Similarly E, I, J and finally K are marked as branching points. Each position leaf is added to the end of the queue while the inserted subtree is created, and each branching point as it is marked. The contents of the queue after the update phase are $\{A, B, C, D, E, F, G, H, I, J, K\}$.

By processing the queued items in this exact order, the tree will be balanced in the proper manner, balancing both subtrees before attempting to balance the parent node. In each router, as many rotations are performed as necessary before climbing upwards. As at times

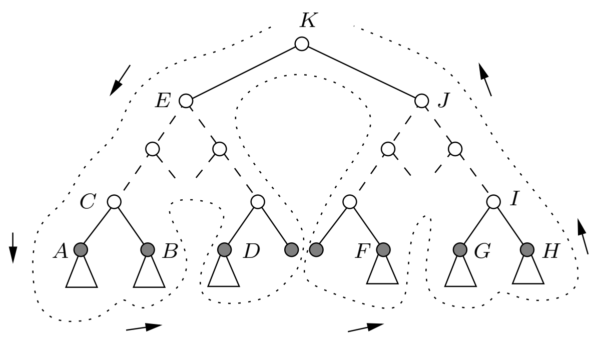


FIGURE 5.3: Recording the balancing information during locate: traverse order according to the arrows, storage in left-to-right order.

more than one rotation is needed, the balancing continues from the new position of the original point of rotation. In order to avoid climbing back up to the position where the point if rotation originally was, the algorithm stores the parent before rotations and returns that as the next node to balance. This avoidance is however not possible after double rotations, as explained later.

If the upward climb of the balancing process meets a branching node, it stops and continues according to the queue. The main procedure pops nodes for balancing from the control structure and forwards them to the procedure that proceeds in the tree as high as possible without interfering with other groups. If the subtree currently being balanced regains balance before reaching the root or a branching point, the procedure returns and the next queued node is processed. The balancing is finished when the queue is empty.

```
procedure balanceRelaxed(list queue) {
  while (not waiters.empty()) {
    balanceFromQueue(queue.pop());
  }
}
```

The following pseudo-code provides for the tree traverse and stopping conditions; the actual rotations are performed in the router according to the balance criteria. That procedure is provided shortly after this one. The stopping condition is simple: at a certain node, if no rotation is necessary to balance it and the height of the node does not change, the subtree is balanced and does not need further work. Note that the height may change even without a rotation, if the subtrees have grown in height but are of proper heights with respect to each other.

```
procedure balanceFromQueue(router current) {
   router previous;
   int oldHeight;

   // If the queued router is not a branching point,
   // but a subgroup attachment position, it is already
   // in balance and may be skipped.
   if (current.branch) {
      current.branch ← false ; // Mark no longer needed
   } else { // Jump to parent node
      current ← current.parent;
   } while (current is not null ) {
      oldHeight ← current.height;
      // Cease balancing at a branching point
      if (current.branch) {
            return ;
      }
      previous ← current;
      current ← current.balance();
      if (balancer.rotation and current is null ) {
         // Necessary after a double rotation:
         previous.balance();
         return ;
      } else if (not balancer.rotation and oldHeight is previous.height) {
      return ;
   }
}
```

Thus the procedure begins at a recorded node and proceeds while not in a branching points or the root, as necessary in the example of Figure 5.3. After the subtree becomes balanced, the balancing stops there and continues to the next queued node, if any. The branching mark needs to be removed from queued nodes before entering the loop in order to pass the stopping condition preventing interference of subtrees under the same branching node. Branching nodes are not balanced until both of their subtrees are in balance (ensured by queue order) and only the subtree balancing needs to know whether an ancestor was a branching node or not, so as a branching node is popped from the queue, the mark is unnecessary.

Should both of the subtrees that caused a node to become a branching node reach balance before the branching node, there will be nothing left to balance there. A condition for this may be added to the pseudo-code, checking with a boolean variable whether all previous balancing procedures have ended before a branching point or by adding the information to the branching node itself as an “unfinished” bit. If so, the next upcoming branching point is also in balance and may be skipped. However, as no balancing is required there, the overhead caused by checking it in the balancing procedure is a small constant. The `whichChild` procedure returns an integer value determining whether the node is the left or right child of its parent, and similarly `getChild` procedure return the requested child based on its parameter.

```
procedure balanceRouter(router current) : router {
   router next ← current.parent;
   int direction ← current.whichChild();
   boolean double ← false ;
   int $h_l$ ← current.left.height;
   int $h_r$ ← current.right.height;
   int $h_a$, $h_b$;
   balancer.rotation ← false ;

   while ($|h_l - h_r| > 1$) {
      if ($h_l > h_r$) { // LEFT SUBTREE IS HIGHER
         $h_a$ ← current.left.left.height;
         $h_b$ ← current.left.right.height;
         if ($h_a \geq h_b$) {
            singleRotateRight(current);
         } else {
            doubleRotateLeftRight(current);
            double ← true ;
         }
      } else { // RIGHT SUBTREE IS HIGHER
         $h_a$ ← current.right.left.height;
         $h_b$ ← current.right.right.height;
         if ($h_a \leq h_b$) {
            singleRotateLeft(current);
         } else {
            doubleRotateRightLeft(current);
            double ← true ;
         }
      }
      balancer.rotation ← true;
      $h_l$ ← current.left.height;
      $h_l$ ← current.right.height;
   }
   current.setHeight(); // ENSURING CORRECT HEIGHT

   if (double) {
      router position ← current.parent;
      while (position is not null and position is not next) {
         position ← balance(position);
      }
      rotation ← true ; // MIGHT BE OUT OF DATE
   }
   return next;
}
```

Note the special treatment of double rotations; under certain conditions, one rotation is not necessary to take care of the imbalance, but proceeding downwards in the sibling branch may understandably be necessary, as the highest subtree changes its sibling. After the downward trend ends, it is generally safe to move on to the original parent. However there may be still imbalance above after a double rotation, which needs to be fixed before continuing to the parent node of the starting position. This situation is portrayed in Figure 5.4: if the subtree being paired with the smallest subtree was one step smaller than its original sibling, the right subtree of the root may become two steps lower than the left, requiring a rotation at the root. Normally the subtree root is not the root and the imbalance is taken care of by the `balanceRouter` routine above, but if it actually is the root, the outer procedure `balanceFromQueue` needs to check for the imbalance.

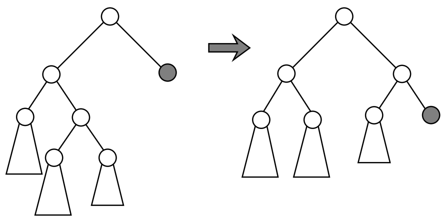


FIGURE 5.4: A double rotation requiring an additional balance check at the root after balancing the right subtree.

As an example of a standard rotation, the single left rotation is provided below. The other three are very similarly implemeted as by Knuth [26]: three link updates for a single rotation four for a double. See the illustrations presented earlier in Section 2.5 for a concrete representation on which node is which. The naming convention here follows that of the illustrations.

```
procedure singleRotateLeft(router current) {
  router u ← current.right;
  router parent ← current.parent;
  node a ← u.left;

  // SETTING THE NODES IN CORRECT NEW POSITIONS
  if (parent is not null ) {
    parent.replaceChild(current, u);
  } else { // NEW ROOT
    u.parent ← null;
  }
  u.replaceChild(b, current);
  current.replaceChild(u, a);
  // SETTING THE CORRECT HEIGHTS
  current.setHeight();
  u.setHeight();
}
```

The `setHeight` method simply compares the heights of the subtrees and sets the maximum plus one as the current height. This is a valid measure, as the subtrees are already previously balanced.

Chapter 6

Algorithm Analysis

This chapter analyzes the algorithm of the previous chapter. Section 6.1 verifies that the algorithm does maintain the structure as a relaxed EBST at all times, the index operations attain their purpose, and that the balancing results in a balanced EBST. In Section 6.2 the computational complexity and other performance related issues are analyzed for each of the three phases of the algorithm.

Formally analyzing an algorithm is an efficient method for gaining general understanding of the performance and correctness of an algorithm not necessarily even implemented; a verbal representation is often sufficient, even though writing the procedures as pseudo-code clarifies the situation. The purpose is to identify the effects of different steps of the algorithm and verify how many times each of them need to be taken and under what conditions. The presentation is ordinarily quite mathematical.

For the problem at hand, database indexing, the approach is well-suited. The tree structure is formally defined and the operations adequately straightforward. A common metric for evaluating search mechanisms as indices is the number of value comparison needed to provide the result (for a search operation true or false) [5]. For trees this maps to the number of links traversed in order to determine the outcome. This and other criteria of algorithmic efficiency are formally treated in the following discussion.

6.1 Proof of Correctness

The goal in analyzing the correctness of a algorithm is to examine whether all the operations preserve the structure in question as a valid member of its class, and whether they have the intended outcome and no unnecessary or harmful side effects. I will begin with the maintenance of a tree as an EBST with relaxed balance. First, this class needs to be defined.

Definition 6.10. A relaxed EBST is an EBST that does not necessarily meet the balance condition for EBSTs, but meets all other requirements: data in leaves, each router having two children, and the order of the key values is as defined in Definition 2.1.

It needs first to be shown that this class of trees is closed under insertion, meaning that an insertion performed on any relaxed EBST will produce a valid relaxed EBST. A subcase is that for any balanced EBST, the resulting tree will be a relaxed EBST, as the balanced trees are also included in the set of trees. If the update operations are not interleaved with balancing, the latter case is sufficient.

Theorem 6.2. All of the index operations preserve a relaxed EBST T as a relaxed EBST.

Proof of Theorem 6.2. For search operations this is trivial, as in single operations stated in Lemma 2.1, as no modifications are made to the structure. For marked deletes no changes occur either. Also for removal deletions and insertions, straightforward reasoning will yield the theorem.

Both of them were noted to preserve an EBST for single operations in Lemma 2.2 and Lemma 2.3 by simple observations: each single insert causes a split of one leaf into a balanced subtree of one router and two leaves. As this balances subtree is a valid EBST and attached properly into an otherwise valid EBST, the resulting tree will also be an EBST, possibly out of balance, making it a relaxed EBST unless a balancing operation is strictly coupled to follow. If the two operations of update and balancing are considered an atomic couple, the relaxed stage will not be observed by any other process.

For a group insertion the position leaf is also replaced with a subtree of at least the same size than resulted in the split. That subtree is constructed by definition to be an EBST. The attachment point of this subtree will remain valid, as it had to have two children to begin with. Therefore the new subtree necessarily has a proper sibling node. Should this subtree replacement occur in more than one position, each of them separately maintains the tree as a relaxed EBST, and as all of the insertions of the same position leaf are grouped together in the locate phase, the subtrees are mutually exclusive. This naturally concludes in the tree remaining as a valid relaxed EBST.

The order of the keys is preserved by properly constructing the subtree, as all of its values, including the key previously stored in the replaced position leaf, belong in the tree order defined in Definition 2.1 to that specific location in the tree and are mutually ordered by the constructing process. Thus no key is in danger of being misplaced, and the parent router to which the subtree is attached needs no updating for its routing value.

For deletions the procedure is alike. From single operations, it can be seen that by removing one leaf and replacing its parent with the position leaves sibling preserves both the order and the rule of two children. The deletion of a subtree can be viewed as a series of separate deletions each performed as a single deletion, but grouped together as an atomic group operation. Each of the consequtive deletions will leave the tree in proper order, and the same effect cam be obtained than with the subtree deletion. Eventually the sibling subtree of the removed subtree will replace of the common parent router.

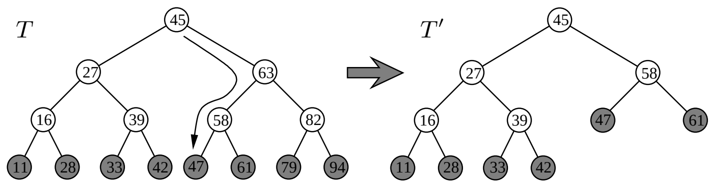


FIGURE 6.1: Deleting more than one key from an EBST.

This can be seen in the illustration of group deletion in Figure 6.1. First deleting key 79 from the right side tree T in Figure 6.1 will cause the router 82 to be replaced with the leaf with key 94. Further deleting this leaf will cause its parent router 63 to be replaced with its other subtree, routed at 58. This will result in tree T' on the left, also obtained by direct deletion of the entire subtree routed at 82.

What comes to the validity of the outcome of these operations, after the attachment of the subtree into the position node in an insertion, the keys are properly accessible through the attachment point and orderly. After a deletion, depending from the implementation strategy, the leaves are either no longer present and thus correctly inaccessible, or clearly marked as invalid. With the implementation using validity bits, it needs to be noted that the index operations must include a check of this attribute. $\Box$

As the index operations have been shown valid, also the balancing must be proven correct. I will begin with stating that the standard rotations will be able to balance a relaxed EBST without ever increasing the height of the tree or performing an improper or unhelpful rotation when the following balancing criteria, thoroughly explained and illustrated in Section 2.5 is used:

Definition 6.11. For a node t with *left*$(t) = u$ and *right*$(t) = v$, the following criteria is used to select the appropriate balancing rotation. The rotation is continued at the same router until balance is achieved.

$|\textit{height}(u) - \textit{height}(v)| \leq 1$:
Balance condition fulfilled; no rotation necessary at t

$(\textit{height}(u) \geq \textit{height}(v) + 2) \wedge \textit{height}(\textit{left}(u)) \geq \textit{height}(\textit{right}(u))$:
A single right rotation at t

$(\textit{height}(u) \geq \textit{height}(v) + 2) \wedge (\textit{height}(\textit{left}(u)) < (\textit{height}(\textit{right}(u))$:
A double left-to-right rotation at t

$(\textit{height}(u) \leq \textit{height}(v) - 2) \wedge (\textit{height}(\textit{left}(v)) \geq \textit{height}(\textit{right}(v)))$:
A single left rotation at t

$(\textit{height}(u) \leq \textit{height}(v) - 2) \wedge ((\textit{height}(\textit{left}(v)) > \textit{height}(\textit{right}(v)))$:
A double right-to-left rotation at t

From Lemma 2.4 it can be obtained that the class of relaxed EBSTs is closed for any single standard rotation. Basic abstract algebra states that if for any operation $f(x)$ where $x \in S$, $f(x) \in S$, then also $f(f(x)) \in S$. This is the same reasoning used for deriving the validity of group deletions from the corresponding single operations already found valid. It is therefore evident that a relaxed EBST will stay such when subjected to any number of the standard rotations, using the above selection criteria.

Next, I will prove that a sequence of such rotations performed upon the appropriate criterion being fulfilled, will result in a balanced EBST. A general result was presented by Bougé *et al.* for AVL trees: “Applying the original AVL rebalancing rules to an arbitrary tree (even very unbalanced!) in an arbitrary order, does evetually reshape it into an AVL tree, even in the presence of incomplete information on the heights of the subtrees” [11]. With a modified rule set, they achieve a complexity of $\Theta(n^2)$ for such rebalancing. This is evidently a very large number of operations, with n being some millions of keys.

As EBSTs are a subgroup of AVL trees, excluding those AVL trees that allow only one child in a router, so this shows that a relaxed EBST can be balanced, and as the standard rotations preserve the number of child links, the resulting tree will be an EBST. But as my intent is also to derive the number of rotations needed for the complexity analysis in Section 6.2.3, a more detailed result is necessary.

Theorem 6.3. Any tree T that is a relaxed EBST can be transformed into a balanced EBST T' by using a finite amount of the standard rotations so that either *height*$(T') =$ *height*(T) or $height(T') =$ *height*$(T) - 1$.

Proof of Theorem 6.3. As shown earlier in Section 6.1, the class of relaxed EBSTs is closed under the rebalancing operations. Obviously the balanced EBSTs are a subset of the relaxed, from the definition of EBSTs in Section 2.2, definition of their balance in Definition 2.6, and the definition of the relaxed class in Definition 6.10.

First the following lemma can be written:

Lemma 6.6. For a tree T that is a relaxed EBST and not in balance, one or more of the rebalancing operations will be applicable. For each node t not fulfilling the balance condition of Definition 2.6, exactly one of the rotations is applicable.

Proof of Lemma 6.6: The selection criteria for the rotations includes the cases $|h_1 - h_2| \leq 1$, $h_1 > h_2 + 2$ and $h_2 < h_1 + 2$, where h_1 and h_2 are the heights of the subtrees. As h_1 and h_2 are positive integers, these conditions cover the entire continuum. One of these is always met; if the node is not in balance, one of these rotations will be selected. □

Similarly, the following observation is recorded:

Lemma 6.7. The rebalancing of a relaxed EBST does not terminate until the tree is completely in balance, although it may pause.

The pausing is included as the algoritm is defined to allow search operations to interleave. Such temporary pausing does not modify the structure. After rebalancing a single node, the relaxed balancing algorithm proceeds to the parent unless a branching point is met. The balancing of that parent will continue after the sibling subtree has been rebalanced, as it naturally proceeds over the branching node if necessary. If updates have occurred in both subtrees of the root, then the root is the last branching point in the queue, due to the post-order queuing. Thus it is the last node visited. It may prove to be in balance already, in which case the rebalancing stops there, the root and subsequently the entire tree being in balance. If there is imbalance left in the tree, one of the rotations is applicable as shown in Lemma 6.6 until the balance condition is fulfilled. Thus Lemma 6.7 is valid. □

As stated earlier in Lemma 2.4, none of the standard rotations can increase the height of the tree, i.e. $T' \leq T$. Both claims of the height of T' are included in this observation. Lemma 2.4 also states that each rotation can reduce the height of the rotated tree at most by one. For a small tree it can be easily seen that the height of the tree before and after the rotation is either same or differs by one.

All four standard rotations are provided together in Figure 6.2 to illustrate the following discussion. The single rotations are each other's mirror cases, and so are the doubles. Hence I only explain the cases of the single left rotation and the right-to-left double rotation.

The *single left rotation* is performed when the right subtree B is higher by at least two steps than the left subtree A and *height*$($*right*$(B)) \geq$ *height*$($*left*$(B))$. If the subtrees of B are of

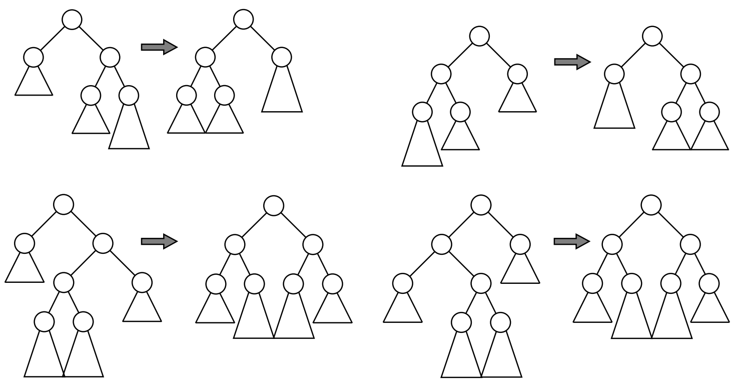


FIGURE 6.2: All of the standard rotations: single left and right rotations on the upper row, double right-left and left-right below.

the same height, the total tree height remains the same. If the right subtree of B is higher, the total height of the tree is reduced by one as it is lifted one step closer to the root. Thus *height*$(T') \geq$ *height*$(T) - 1$. The result holds also for single right rotation.

The right-left double rotation is performed when the right subtree B is higher by at least two steps than the left subtree A and *height*(*right*$(B)) <$ *height*(*left*$(B))$. This rotation lifts the subtrees of B left child one step higher, lowering A by one step.

The height of A was at least two lower than the height of B to begin with, and the height of B higher subtree is necessarily one less that the original height of B by Definition 2.4. Thus the height subtree of this subtree must be again one less, i.e. equal to the original height of A. Its sibling must have a smaller height, as must the smaller subtree of B.

Thus in the final position, A's height is either the same than the height of the lifted subtree or less. Therefore the height of the tree is necessarily reduced by one unit, *height*$(T') =$ *height*$(T) - 1$. It cannot be reduced more: even when the largest lifted subtree is larger than A, it still rises only one step. The result holds for the other double rotation as well. This is recorded as a lemma for further reference:

Lemma 6.8. For any double rotation on a relaxed and imbalanced EBST T applies that if T' is the resulting EBST,

$$\textit{height}(T') = \textit{height}(T) - 1.$$

This observation has been generalized by Malmi and Soisalon-Soininen [34] for height-valued trees, which are of the exact same shape than the EBSTs and thus their reasoning holds in this case also. As the equality has been shown as the maximum value for *height*(T'), they approach the lower limit by a counter-argument. Assuming that *height*(T') is at least two units smaller than *height*(T), they note that this would only be possible if at least two height decreasing rotations were performed at the root.

This could only be possible with the first rotation being a double rotation. Let B_1 and B_2 be the children of the higher subtree of the higher child of the root (named as in the original illustrations in Section 2.5 on page 2.5. For the double rotation in question here, *height*$(B_1) <$ *height*(B_2) is required. None of the other cases would leave the tree still one unit imbalanced after the rotation, as required by the counter-argument. Malmi and

Soisalon-Soininen conclude this to be a contradiction, as “in the remaining case the second rotation at the root cannot be height decreasing” [34].

For the Theorem 6.3 it has been shown that a relaxed, imbalanced EBST can be transformed into a balanced EBST. What remains to be proven is that only a finite number of rotations is needed. To keep record on how much work needs to be done to get a tree in balance, a definition for the amount of imbalance in a node is needed:

Definition 6.12. The *imbalance* of a node t is defined as the height difference of the subtrees of t minus one, as one unit is allowed in a balanced tree:

$$\textit{imbalance}(t) = |\textit{height}(\textit{left}(t)) - \textit{height}(\textit{right}(t))| - 1.$$

The number of rotations needed to eliminate all imbalance depends on the number of rotations needed to remove one unit. Each rotation can remove at most one unit of imbalance from the point of rotation, but it may also have no effect on that. According to the following lemma, every other rotation of a balancing sequence must however reduce the imbalance.

Lemma 6.9. In a sequence of rotations on an imbalanced tree T, if the rotation that transforms T into T' will not remove imbalance from the point of rotation, the next rotation transforming T' into T'' must do so. Thus either $\textit{height}(T') = \textit{height}(T) \wedge \textit{height}(T'') = \textit{height}(T') - 1$ or $\textit{height}(T') = \textit{height}(T) - 1 \wedge \textit{height}(T'') = \textit{height}(T')$.

Proof of Lemma 6.9: According to Lemma 6.8, each double rotation necessarily decreases the total height of the tree. This is the case in which the first rotation already removes a unit by reducing the height difference between the siblings causing the imbalance.

For the second case, the first rotation must be a single rotation. Let the subtrees of the higher child of the root be A and B as in the illustrations of Section 2.5. If $\textit{height}(A) < \textit{height}(B)$, the first rotation will already decrease the height of the tree (see the preceding discussion on single left rotation), and the rotation falls into the first case. Thus the second case consists only of the rotations for which $\textit{height}(A) = \textit{height}(B)$. As the rotation moves either A or B to another branch in the tree, the children of the new root are necessarily of different heights as seen in the illustrations. Thus, any rotation here, single or double, will be a height-decreasing one. □

Returning to the proof of Theorem 6.3, we have that two rotations are needed per the amount of imbalance to balance a tree. For a node t with two balanced subtrees A and B, the number of rotations needed at t can be written directly with the subtree heights as $2(\textit{height}(A) - \textit{height}(B)) - 1$, or as $2 \cdot \textit{imbalance}(t)$. As EBSTs are by definition finite trees, the height difference of two subtrees must also be finite. Therefore the amount of rotations necessary is a finite number. This completes the proof of Theorem 6.3. □

It has now been shown that all of the operations of the relaxed-balance update algorithm of EBSTs have the desired outcome and the class of trees is closed for each of the operations. Thus the algorithm operates correctly. I will continue by analyzing the complexity of each of these operation to provide a total complexity for the algorithm as a whole.

6.2 Proof of Complexity

The very first factor of interest in analyzing the complexity of a group-update algorithm is the average number of links traversed per key in the locate phase [5]. This is calculated in Section 6.2.1. The next measure of interest is the complexity and the locality of the update phase; how much time does it take per key to perform the update operations, and how many nodes need to be operated on per updated key, i.e. how much does the operation interfere with other operations. The update phase is analyzed in Section 6.2.2. Finally, the complexity and locality of the balancing needs to be assessed; how many rotations are needed per updated key and how many nodes need to be accessed per key to bring the tree back in balance. This is done in Section 6.2.3. The complexity of the entire algorithm is drawn together in Section 6.2.5.

6.2.1 Locate Complexity

If tree T has n leaves, it has $n-1$ routers from to Lemma 4.5. Each router has two child links by the definition of full external trees, from which the number of links in an EBST is $l = 2(n-1)$. In the worst case the divide-and-conquer –method will have to traverse each link T once. This occurs when $m \geq n$, and each leaf of T is the position leaf of one or more keys in the batch. This is an unrealistically big batch size for a regular database, but it may occur in construction phase.

Lemma 6.10. The worst-case complexity for $m \geq n$ is $O(2(n-1)) = O(n)$.

To compare this with the same situation in a series of single operations to a perfectly balanced tree, each leaf again being the position leaf of at least one key. For a series of m single locate operations it is known from the height of a balanced EBST that the complexity is exactly $\Theta(m \log n)$. This is always bigger than $O(n)$ when $m \geq n$.

Normally, $m \ll n$, so this is not a practical way to evaluate the complexity. Generally speaking, for a series on m single keys, the locate complexity is $m \cdot O(\log n) = O(m \log n)$ for any $m \leq n$. For a group operation of size m, a loose upper bound is derived from the number k of position leaves occupied by the group, each of these position leaves being $O(\log n)$ steps from the root.

Theorem 6.4. The locate complexity by the number k of position leaves occupied by the group of m keys in a tree of size n is $O(k \log n)$, where $k \leq m \leq n$.

However, it is inevitable that if $k > 2$, additional savings are caused by the common prefixes to position leaf paths. Normally m is a few thousands, k in the magnitude of at least hundreds and n is a couple of millions. To better understand the savings made by the group location, a more thorough discussion is given on the effect of the number of position leaves.

For a random sequence of m keys each configuration of position leaves into a random balanced EBST with n leaves is equiprobable. If k is the number of position leaves occupied by the m keys of a group M and n is the total number of leaves, there are $\binom{n}{k}$ different ways to select the position leaves K from the set of all leaves N. Note that there cannot be more position leaves than keys in the group; $1 \leq k \leq m$.

As in practical applications $m \ll n$, the probability increases with the size of k, so it is necessary to evaluate how many position leaves is a group of m keys likely to occupy. This is essentially the problem of *partitioning* the integer m. A partition is essentially writing the integer as a sum of positive integers, where the order of the summands is insignificant. Thus each summand would present one position leaf, having as many keys in its subgroup as the summand represents. Thus there as many ways to express m as a sum of k integers than there are ways for m keys to be divided into k position leaves. For $k \leq m$ the number of possible divisions can be obtained from the following generating function

$$g(x) = \frac{1}{(1-x)(1-x^2)\ldots(1-x^m)}.$$

Another way of representing these partitions is with the following recursion:

$$P(m,k) = P(m-1,k-1) + P(n-k,k),$$

with the following conditions: $P(m,0) = 0$, $P(m,m) = 1$ and $P(m,k) = 0$ when $k > m$. This constructs the following triangle:

$$\begin{array}{ccccccccccc}
 & & & & & 1 & & & & & \\
 & & & & 1 & & 1 & & & & \\
 & & & 1 & & 1 & & 1 & & & \\
 & & 1 & & 2 & & 1 & & 1 & & \\
 & 1 & & 2 & & 2 & & 1 & & 1 & \\
1 & & 3 & & 3 & & 2 & & 1 & & 1
\end{array}$$

This figure is interpreted line be line; the bottom line, the sixth in the triangle, gives the number of divisions for the integer 6 into one to six summands. There is one way to represent six as the sum of six integers, $6 = 1+1+1+1+1+1$. This is the rightmost element of the bottom row. The next element to the left defines how many ways are there to represent six as a sum of five integers: $6 = 1+1+1+1+2$. Furthermore, there are two ways to use four summands: $6 = 1+1+1+3$ and $6 = 1+1+2+2$, etc.

By implementing this as a small Java-program, I constructed Figure 6.3 for $m = 150$, showing how many possibilities are there to divide m into k position leaves. Also in this case, the curve shape is essentially similar for all m, having weight below $m/2$.

Evaluating the probability of k with respect to m gives that k is likely to be much smaller than m. It can be concluded by observing Figure 6.3 that k is likely to be approximately $m/10 < k < 4m/10$.

There are $\binom{n}{k}$ ways to select these k position leaves, and depending on the mutual distances of the position leaves, the savings made in the link traverses can be approximated. Each position leaf needs to be traversed to, thus at most $O(k \log n)$ traverses are needed. From this amount, the common prefixes of the position leaf paths are also reduced to just one traverse. The worst case for the locate phase is that the position leaves are as far away from each other as possible on the leaf level, with approximately n/k leaves between them. Note that $n \gg k$. Still, when beginning from the root, as the positions are as far apart as possible and $k > 2$, the algorithm only needs to traverse each of the root's child links once, instead

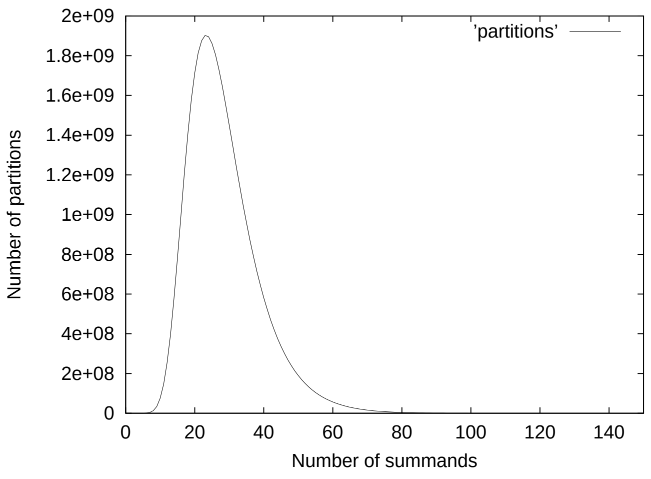


FIGURE 6.3: The division of 150 into partitions.

of once per key. The same occurs on the next level if $k > 4$, only needing to traverse 4 links instead of once per position. Similarly on level j, if $k > 2^j$, the traverse of $k - 2^j$ links is saved. Summing over those levels where $k > 2^j$ and bringing the constant k outside the summation gives

$$dk - \sum_{j=1}^{d} 2^j, \text{ where } n \text{ is such that } 2^d < k < 2^{d+1}.$$

If $m = 5\,000$, the previous results state that $50 < k < 2\,000$. For the limit values the savings are respectively 188 links and $17\,954$, expecting that the tree is sufficiently deep. As usually the size of the tree n is some millions and the maximum depth is directly related to the height of the tree according to Definition 2.5, the value of the tree depth can be approximated: $\textit{depth}(T) = \log n - 1 = \log 1\,000\,000 - 1 \approx 20 - 1 = 19$. Thus if $k \approx 1\,000$, the last level where the above summation guarantees savings is $d \approx 10$, which is half way down the tree.

The complexity for the locate phase could now be represented as the positive difference between the worst-case $2n - 2$ traverses and the links saved: $O(2n - 2 - dk - \sum_{j=1}^{d} 2^j)$. Another way to approach this is by the observation that until level d, the algorithm traverses every link above that level in the worst case, and $O(k \cdot h)$ afterwards, where h is the height from that level to the leaf level. In the above analysis it was stated that a level of depth j has 2^j links. Thus the algorithm traverses $\sum_{j=1}^{d} 2^j$ links and the paths from there to the leaves individually in the worst case. As the an upper bound to tree height is $1.44\,\log n + 0.328$ as in Theorem 2.1, the path left to traverse from depth d is $1.44\,\log n + 0.328 - d + 1$, as tree depth is defined as height minus one.

For the sum of exponentials of two, the following formula applies: $\sum_{i=0}^{j} 2^i = 2^{j+1} - 1$. Thus it is possible to simplify the expression of the number of link traverses, as $2^0 = 1$:

$$2^{d+1} + k\,(1.44\,\log n - d + 1.328) = 2^{d+1} + 1.44\,k\,\log n - kd + 1.328\,k. \qquad (6.1)$$

d was formerly defined as the level where $2^d < k < 2^{d+1}$. Thus 2^{d+1} is of a similar magnitude with k, varying from just above k to approximately $2k$, as $2^d < k$ and $2^{d+1} = 2 \cdot 2^d < 2k$. I will examine the complexity for both ends $2^{d+1} \approx k$ and $2^{d+1} \approx 2k$.

If $2^{d+1} = k$ with reasonable accuracy, $d = \log k - 1$. Substituting these in 6.1 gives $k + 1.44\,k\,\log n - k\,\log k - k + 1.328\,k = 3.328\,k + 1.44\,k\,\log n - k\,\log k$.

For the second case, $2^{d+1} = 2k$ and $d = \log 2k - 1 = \log 2 + \log k - 1 = 1 + \log k - 1 = \log k$. Substitution into 6.1 gives now $2k + 1.44\,k\,\log n - k\,\log k + 1.328\,k = 3.328\,k + 1.44\,k\,\log k - k\,\log k$, which is the same result than yielded by the previous substitution. Both of these have obviously the same $O(k\,\log n)$ complexity than was directly derived earlier, but the saving can be better understood by these calculations. And if k is of the magnitude $m/3$, as suggested by the shape of the partition curve in 6.3, savings of approximately every third or second link can be expected in comparison to the single operations of $O(m\,\log n)$ complexity. This estimation can be useful when m and n are known and the key distribution is expected to be even.

For other distributions, an approach based on the mutual distances of the position leaves is more useful. The following result by Brown and Tarjan [15] for $2-3$ trees can be used as a starting point:

Lemma 6.11. For a $2-3$ tree with n internal nodes labeled $0, 1, \ldots, n-1$ in a left-to-right order, the number of distinct nodes N_p on the paths from k external nodes $p_1 < p_2 < \ldots < p_k$ to the root is

$$N_p \leq 2\left(\lceil \log n \rceil + \sum_{1<i\leq k} \lceil \log(p_i - p_{i-1} + 1) \rceil\right).$$

Soisalon-Soininen and Widmayer [48] have rewritten this for AVL trees:

Lemma 6.12. Let $t_1, t_2, \ldots, t_q$ be leaf nodes of an AVL tree T such that $t_1 < t_2 < \ldots < t_q$. For the number N_t of nodes that lie on the search paths of $t_1, \ldots, t_q$ holds

$$N_t \leq 4\left(\lceil \log n \rceil + \sum_{i=2}^{q} \lceil \log(t_i - t_{i-1} + 1) \rceil\right).$$

From Lemma 6.12, the following upper bound complexity is directly obtained:

Theorem 6.5. The group search of m keys $k_1, k_2, \ldots, k_m$ requires $O(\log n + \sum_{i=1}^{m-1} \log d_i)$ link traverses, where d_i the the distance of leaves k_i and k_{i+1}.

To benefit from this result, it is understandable that the approximate probabilities of position leaf separation distances need to be known. Both this approach and the partitioning approach above give rise to expectations that the complexity of the group locate should be better than that of a series of single locates. As a series of single locates requires $O(m\,\log n)$ link traverses, and this has been shown to be the worst case of group location as well with savings shown, the assumption of a tighter upper bound is justified.

6.2.2 Update Complexity

As stated earlier in Section 6.1 while discussing the behavior of the index operations in the update phase, all update operations begin with a check: is the position leaf key included in the subgroup of that position. As the subgroup size is m and it is stored as an ordered list, this check requires $O(\log m)$ time. If this check is positive and the operation is search, nothing more needs to be done; the key information is returned as a signal of a successful operation. The same result applies for deletion by validity marks, as there we simply mark the position leaf if the check is positive. For both search and delete failing the check signifies operation failure.

Theorem 6.6. The search and delete group operations have $O(\log m)$ complexity with m as the size of the group.

For the index operation the check is performed in order to avoid replacing of an existing key. If the check is positive (providing worst-case complexity), the key of the position leaf is added in the subgroup in the correct position. This can be done in $O(\log m)$ time, as the sequence is ordered. After this, in the second step of the group insert, a balanced tree is constructed from the subgroup in $O(m)$ time. This is the best possible upper bound as each key needs to be turned into a leaf, taking $m \cdot O(1)$ time. After the tree has been built, it is attached on the place of the former position leaf, requiring the parent link to be set and the parent's child link updated. This requires $O(1)$ time.

Thus the total complexity of the insertion is $O(\log m) + O(1) + O(m)$, which reduces into $O(m)$, as $1 < \log m < m$.

Theorem 6.7. The group insertion has $O(m)$ complexity with m as the size of the group.

6.2.3 Balance Complexity

For the balance phase, the most influential factor what comes to algorithm performance is obviously the number of rotations needed per key to return an EBST into balance [5]. As each of the standard rotations defined in Section 2.5 requires a constant time and the balance procedure needs to traverse the tree in order to restore the balance, the total running time of the balance phase is proportional to the number of rotations plus the number of visited nodes [34].

In proving the correctness of the balancing algorithm it was shown that the number of rotations needed to balance a tree is finite and a value was provided for the rotations needed to balance an imbalanced node t.

Lemma 6.13. The number of rotations needed to balance node t is $2 \cdot \textit{imbalance}(t)$.

From Definition 6.12 it is easily derived that when a group insertion occurs on a balanced EBST, the imbalance caused by the insertion at the parent of the position leaf is directly proportional to the height of the newly attached subtree T_p. Malmi and Soisalon-Soininen [34] have analyzed the total number of rotations needed for height-valued trees, showing that the tree reaches balance possibly even before reaching the root.

Their proof of complexity is valid for the algorithm presented in this thesis as well, as the main difference is in marking the imbalance, but the balancing proceeds in the same exact manner. This section basically summarizes their solution. The analysis begins with determining the number of rotations needed to balance a tree where a group of m keys were inserted in the same position leaf, $k = 1$. This is further generalized to apply for any value of k.

The balancing proceeds as defined by the algorithm in Section 5.4 from the leftmost position leaf's parent up until it reaches a branching node or a root. No router is balanced until both of its subtrees are known to be in balance. As explained in the algorithm, the subtree in question may become balanced even before reaching a branching node. The nodes in which this happens, are called *stopping nodes*.

Lemma 6.14. A node u_s on the path from u_k to the root u_1 that does not require rebalancing after both of its subtrees are rebalanced is called a *stopping node* and $s \geq k - 2h - 1$.

Here h is the height of the inserted subtree of size $m+1$, $h = \log(m+1)$. They proved this with a counter-argument by observing the heights of the three consecutive trees of Lemma 6.9. Further they showed that after u_s, at most one more rotation is needed on the path $u_s, u_{s-1}, \ldots, u_1$:

Lemma 6.15. Assume that the tree T has been balanced up a point where a stopping node is found. Then *at most one more rotation* is needed to get T completely in balance.

After u_{s+1} has been balanced and u_s needs no more balancing, one more rotation may be necessary, as the rotation that balanced u_{s+1} could have increased the height of u_s. This increase was at most one unit. Therefore it is possible that this one extra unit still needs to be removed from the subtree height difference in some of the nodes $u_s, u_{s-1}, \ldots, u_1$. As it was stated in the lemma that u_s is in balance, that possible imbalance is above it.

Let u_r be the node that is the location of the unit of this imbalance. Thus a rotation is performed at u_r, which must be height-decreasing as otherwise the height of u_n would have not originally increased. Thus the height of u_r does not change due to the rotation there and no more imbalance can exist above u_r, as the heights have not changed and the tree was originally in balance. □

From these lemmas Malmi and Soisalon-Soininen [34] concluded that at most $O(h^2)$ rotations are needed, i.e. with $h = \log(m+1)$, $O(\log^2(m+1)) = O(\log^2 m)$ rotations.

Theorem 6.8. The number of rotations needed to balance an originally balanced EBST after a successful group insert of m keys in one position leaf ($k = 1$) is $O(\log^2 m)$.

Let the number of position leaves now be $k \geq 1$, and the new subtrees constructed for these leaves $T_1, T_2, \ldots, T_k$ in left-to-right order by position in the tree T. The balancing is assumed to start from T_1 and continue in the given order. As presented in Section 5.4, the balancing stops if a branching node is reached. By definition, branching nodes are those routers from which at least one insertion was made in both of its subtrees. These can also be defined as *the lowest common ancestors* of the trees T_i.

Malmi and Soisalon-Soininen [34] state that the *balancing of an inserted subtree T_i* ends at the first node u_k, for which one of the following applies:

(i) $\textit{height}(u''_{k-1}) \geq \textit{height}(u'_{k-1}) + 2$

(ii) u_k is not a branching node and is in balance after the path below it has been balanced

(iii) u_k is the root of the tree

They formulate the following lemma, where stopping nodes are as in Lemma 6.14 and additionally branching nodes. The proof of this lemma bases essentially on Lemma 6.9

Lemma 6.16. Let u_k be the node where the balancing of T_i ends, and let $u_{s_1}, \dots u_{s_p}$ be the stopping nodes in the path $u_1, \dots, u_k$, where u_1 is the root of T_i. Then the number of nodes before the first stopping node, between any two consecutive stopping nodes, and after the last stopping node is $O(\textit{height}(T_i))$ [34].

They continue the generalization by rewriting Lemma 6.15 in the following form:

Lemma 6.17. Let u be the node where the balancing of a subtree T_i ends. Then at most two more rotations are performed before reaching the the next node on the path to the root, where the balancing of another T_j ends. If there is no such node, these two rotations suffice for reaching the root in balance.

This is based on the observation that the height of u may have been increased in such a way that the first rotation performed above it will not be height-decreasing, but another is needed.

With Lemma 6.16 and $O(\log m)$ rotations sufficing for a single position leaf as recorded in Theorem 6.8, Malmi and Soisalon-Soininen [34] derive the general result:

Theorem 6.9. The number of rotations needed to balance an EBST T that was originally in balance is at most $\sum_{i=1}^{k} c_i \cdot \textit{height}(H_i)^2$, where c_i is a constant and H_i is the ith position leaf where a subgroup was inserted.

Proof of Theorem 6.9: The proof follows the outline given by Malmi and Soisalon-Soininen [34]. Let u be the node where the balancing of at least one T_i ends. The trees for which the balancing stops here are consecutive, $T_j, \dots, T_{j+s}$, one of which is T_i. Lemma 6.16 states that the number of rotations needed before, between and after a stopping node on the path to the root is $O(\textit{height}(T_i))$. From Lemma 6.13, the number of rotations needed is at most $O(\textit{height}(T_i))$ in all of the nodes on the path from there to the root. As there are s subtrees that reach balance at u, there are $l-1$ stopping nodes.

The rotations needed before the first stopping node are naturally assigned to T_j, the earliest subtree from the left. Similarly the rotations needed between nodes u_{j+p} and u_{j+p+1} are assigned to the subtree T_{j+p+1}. The rotations needed between the last stopping node and u can be assigned to any $T_j, \dots, T_{j+s}$, as it is shared by all the paths from any of these subtrees to the root. The same applies for the at most 2 additional rotations from Lemma 6.17. For any single T_i, more than $O(\textit{height}(T_i)^2)$ rotations are necessary. From these observations and the fact that each T_i, $i = 1, \dots k$, has exactly one ending node u, the theorem is concluded. □

In addition to the number of rotations, the balance phase complexity includes the path traversed by the algorithm. As in the analysis of locate complexity, this depends on the number of position leaves. A loose upper bound is obtained similarly; the path from each position leaf to the root may need to be traversed, but savings may result from common postfixes and possible stopping before the root. As each path is at most $O(\log n)$ steps from the subtree root to the root of the tree and there are k position leaves, the complexity is again at most $O(k\ \log n)$. Also here, similar considerations with the group size m can be done as I did in Section 6.2.1. This can be summarized into the complexity of the balancing step:

Theorem 6.10. The total complexity of the balance phase of the algorithm is $O(\sum_{i=1}^{k} \log^2 m_i + k\ \log n)$, the further term being the rotation cost and the latter the maximum path length.

The upper bound for the length of the path between a node and the root given in Theorem 6.5 is also valid for the backward propagation of the rebalancer. The difficult part is to include the effect of the tree possibly reaching balance and terminating the traverse before reaching the root. From Lemma 6.14 it is known that the balancing of a branch is close to being finished at a node u_s, $s \geq k - 2h - 1$, where u_1 is the root. Thus the inequality seems to provide an upper bound for the path length on the route from one position leaf towards the root. However, there is still one possible imbalance somewhere above u_n, and as its position or distance is not known, the balancing may as well climb up to the root.

For optimally balanced binary search trees, in which shortest and longest path differ at most by one, Andersson and Lai [4] obtain a balance complexity of the same order with their reconstruction technique (partial rebuilding after the tree size has doubled). Providing that there are no deletions, they show that the amortized number of rotations is $O(\log^3 n)$ per each insertion. They employ stricter balance rules than the AVL variants, aiming at superior performance when the trees are balanced, but risk the tree height during periods of growth.

As the complexity for each of the three phases have now been analyzed, comparisons can be made to the strictly-coupled single operation complexity. Therefore, the analysis of the single case is summarized in the following section, after which the total complexity of both approaches is evaluated.

6.2.4 Strictly-Coupled Single Update Complexity

Before comparing the complexity of the presented algorithm, the corresponding complexities for strictly coupled single operations are provided. As noted on several occasions, the path length from the root to any single leaf is $O(\log n)$, which is the complexity of a single search operation, as the comparison on whether a match was found is done in $O(1)$ time. Also the insertion is simple: the position leaf is divided in two in $O(1)$ time. The more complicated question is the balancing; how likely is a single insertion into a balanced tree to result in a rotation? This is not trivial, as the tree is in balance if the difference in subtree heights is at most one. Thus it is well possible to make an insertion without the need to perform a rotation, but eventually as a subtree grows sufficiently, rotations are needed.

Tarjan [51] has shown that balanced binary tree can be balanced in $O(1)$ rotations after a single insertion. This also applies for a series of m single insertions, regardless of whether they take the same position leaf or spread differently. The worst case, inserting a sorted sequence into the same position leaf (always splitting the leaf created by the previous insertion), is most similar to the group insertion. This gives $m \cdot O(1) = O(m)$ for the number of rotations for a series of keys. The same result applies for a sequence of updated in weight-balanced trees, such as the BB$[\alpha]$ trees [10]. That result is also valid for the rotations caused by deletions.

This result applies for EBSTs; as each insertion causes at most one rotation in the tree, as a loose upper bound as this is obvious. A practical experiment of inserting a sequence of keys, all having the same position leaf in growing order proves this quickly. Let the leftmost leaf of a balanced tree be $n + 1$. Inserting the sequence $\{1, 2, 3, 4, ..., n\}$ in order one by one will cause the following operation: inserting 1 splits $n+1$, inserting 2 splits 1, inserting 3 splits 2, and finally inserting n splits $n - 1$. Almost every single one of these insertions will result in a rotation. Occasionally two consequent insertions are made with no rotation, but the trend is one rotation per one insertion. Were the keys not sorted, rotations would be more rare, but still in linear relation, as they are in no way related to each other.

Another approach for analyzing the properties of balanced tree structures is *fringe analysis* first sketched by Yao [53] and further developed by Brown [14]. The fringe is the bottom part of a tree, obtained for example by deleting all leaves that have at least a specified number of leaves below them or by defining that it consists of *fringe nodes*, i.e. those nodes that have at least one leaf child [14, 36]. After obtaining the fringe, the top of the tree is ignored during the analysis, thus simplifying the extent of the operations.

Yao [53] himself analyzed $2 - 3$ trees and the average number of nodes of order m in B-trees, showing the asymptotic storage utilization to be approximately $\ln 2$. Ottmann and Wood [43] used this strategy to estimate the average storage required by $1-2$ brother trees, a class of trees easily transformed into AVL trees and back. Brown [14] employed the method directly for AVL trees, although only performing a partial analysis. He used fringe analysis to determine the portion of completely balanced nodes, i.e. those nodes whose subtrees are of equal height.

As the fringe is formed by cutting of the top of the tree, it is obvious that the resulting nodes for a disjointed set of trees. These trees can be classified in a finite number of different types, i.e. the type collection is closed. The trees of this forest for an AVL tree can be classified into two main types: the N-trees that have one internal node with two leaf children, and the M-trees that have one leaf child and the other child is an N-tree (two mirror cases) [14]. Brown analyzes the proportions of these trees in the fringe upon insertions, not reaching the same accuracy than Yao did with $2 - 3$ trees, as the height balanced trees occasionally require rotations outside the fringe. This obstacle has since then been removed, and amongst all Baeza-Yates [7] explains how to use fringe analysis for weakly closed collections, thus allowing the rotations.

Mehlhorn [36] extends this analysis to include also random insertions. He uses the same definition of fringe than Brown and derives the average number of balanced nodes to be $4n/9 \leq \bar{B}(n) \leq 7n/8 + o(n)$. Baeza-Yates, Gonnet and Ziviani used fringe analysis to obtain a result needed in this thesis: a probability of a rotation in an AVL tree.

Theorem 6.11. For the probability P_r of a rotation upon an insertion into an AVL tree, the following limits apply:

$$0.37 < P_r < 0.73$$

The exact value of P_r seems to be less than 0.56 [6, 8].

To prove this, they first define a closed collection of AVL trees that appear in the fringe, define bounds for the rotation probability, generalize the fringe tree collection into a weakly-closed collection, define the recurrence for the fringe and solving for the above result in symbolic form. As the rotations of EBSTs are the same than those of AVL trees and the strictly coupled single insertions into these trees are identical, the result is also valid for EBSTs.

This result on rotation probability will be useful both when comparing the analytical performance of the single operation approach to the group algorithm and also when evaluating the reliability of the experiments of Chapter 7. As a point of interest, the fringe analysis is in many ways analog to Markov chains and urn processes [6].

Another question is the length of the path traversed by the rebalancing of this sequence. This depends on how high a rotation takes place. For single insertions, as the tree reaches balance after at most one rotation, the balancer will terminate after performing a rotation. Another stopping criterion is that the balancing meets a node whose height does not change due to the update even though no rotation is performed. Mehlhorn and Tsakalidis [37] call the last imbalanced node, i.e. the node immediately below the stopping node a *critical node*. They show the following theorem:

Theorem 6.12. The length of the path from the position of the update to the critical node has amortized length of 2.618 per insertion, giving directly $2.618\,n$ total length for a sequence of n arbitrary insertions.

For completely random insertions, the provide an improved bound for the length of the balancing path: $1.47 \leq$ path ≤ 2.26 [37]. I have not yet seen a similar result for the group updates, which is a major point of interest in future work.

Huddleston and Mehlhorn [21] have estimated the distribution of rebalancing operations on the different levels of (a, b)-trees with hysteresis calculations, arriving at a result for the number of rebalancing operations on each level for three different cases. They begin their analysis by examining the propagation of the changes caused by updates, observing the relations of splits and fusions of the nodes. Unfortunately their analysis is not easily adopted for AVL trees and the analysis is quite heavy to be conducted or included in a Master's thesis. It would nevertheless be desirable to provide such analysis for EBSTs as well in future work.

6.2.5 Total Complexity

The case of the group insert complexity is straightforward; the path length in the locate phase is $O(k \log n)$, and likely much smaller due to the possibility of savings. For a series of single search operations, the complexity is $\Theta(m \log n)$, and as $k \leq m \leq n$, the group case is always at least as good as the single case what comes to link traverses. As there is no compact formula for the additional savings, I have constructed the graph of Figure 6.4 to show the number of links traversed for $n = 1\,000\,000$, $m = 5\,000$ as the value of k goes from 1 to m.

For the group search, an additional $O(m \log m)$ is required for sorting the keys, but the sorting may be done either by the client application or at the index, so that computational load is not necessarily served by the indexing system. In some application areas, the keys may be more or less ordered to begin with; this is the also idea behind Malmi's group

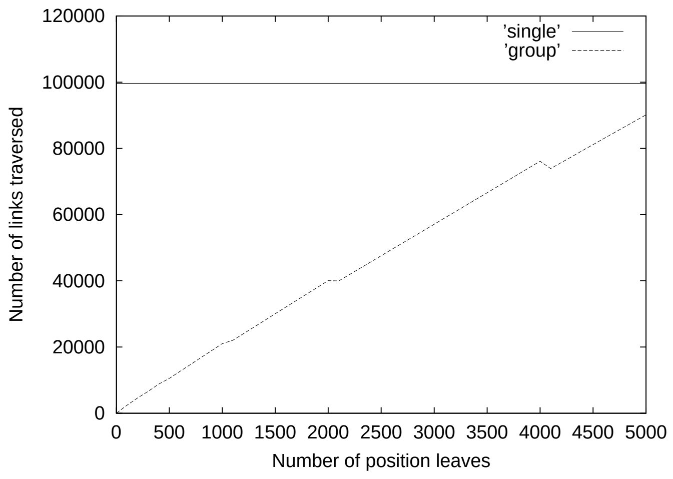


FIGURE 6.4: The number of links traversed for single and group locate procedures with $k \in (1, m)$.

search algorithm that uses the dependencies between consequent keys to make savings in traversing [33]. In such a case the sorting is considerably faster.

For the insertion, the locate phase is exactly the same as for the search operations. Both the single and group updates on the leaf level are $O(1)$ per leaf, giving $O(m)$ for m single insertions and $O(k)$ for a group operation with group size m. As $k \leq m$, the group update is at least as efficient as single updates when attaching the new leaves to the tree. Note that this does not include the balancing, but merely the update phase.

For the rotation performance, the complexity for group operations is $O(\sum_{i=1}^{k} \log^2 m_i)$ by Theorem 6.10 and the probability of a rotation in single case is estimated to be close to 0.56 by Theorem 6.11. For a series of m insertions, each independent, the total number of rotations is naturally $0.56\,m$ with sufficient accuracy.

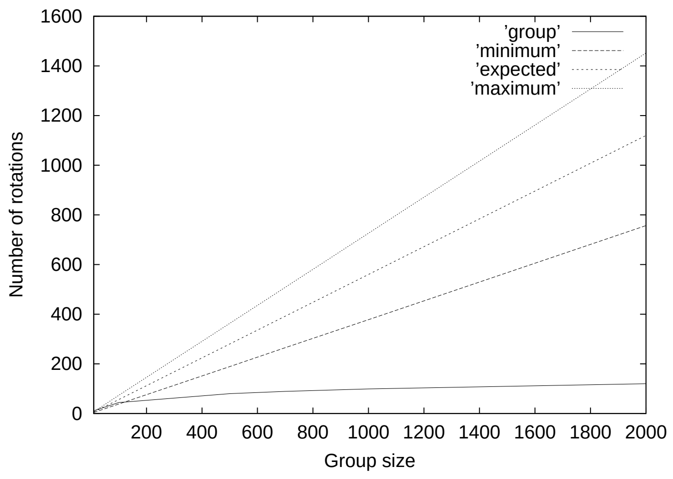


FIGURE 6.5: The expected number of rotations for different subtree sizes in one position leaf.

Figure 6.5 has been constructed for $m \in \{10, 100, 500, 700, 1\,000, 2\,000\}$ and it shows the curve for $O(\log^2 m)$ together with the limit values of single rotation probability and the estimate from Theorem 6.11. Another question is how the rotate complexity behaves for

$1 \leq k \leq m$. To provide an idea on the behavior, I will return to the partitioning problem discussed Section 6.2.1.

For any $m = \sum_{i=1}^{k} m_i$, there are always the two trivial partitions $k = 1 \Rightarrow m_1 = m$ and $k = m \Rightarrow \forall i \; m_i = 1$. As Theorem 6.9 states the rotation cost as an expression of the subtree height, and the actual subtree contains the inserted keys plus the original position leaf, the rotation cost for $k = 1$ is of the magnitude $\log^2(m+1)$ and the rotation cost for the latter case is $k \log^2(1+1) = k$. If $m = 16$, these costs are the same (note that the logarithm is taken in base two). For $m < 16$ choosing $k = m$ produces the lower cost and for $m > 16$, $k = 1$ produces the lower cost. As an example, I have calculated the rotation cost magnitudes $\sum_{i=1}^{k} \log(m_i + 1)$ for some of the partitions of $m = 500$. The results below show that neither one of the trivial cases is thus the maximum, but the $k = 1$ case is apparently the minimum.

k	m_i	cost
1	500	80
3	$100 + 150 + 250$	160
5	$100 + 100 + \ldots + 100$	222
7	$50 + 75 + 75 + \ldots + 75$	266
10	$50 + 50 + \ldots + 50$	322
20	$25 + 25 + \ldots + 25$	442
50	$10 + 10 + \ldots + 10$	598
100	$5 + 5 + \ldots + 5$	668
250	$2 + 2 + \ldots + 2$	628
500	$1 + 1 + \ldots + 1$	500

As the last cost summand, the expected height of a rotation is a problematic issue, it is best compared by the experiments of Chapter 7. For both cases the longest possible path reaches the root, and for the group case this gives $O(k \log n)$ links and $O(m \log n)$ for the singles, $k \leq m$.

From the discussion of this chapter, the total complexity for the relaxed-balance group-update algorithm can be concluded as follows:

Theorem 6.13. The complexity of the presented algorithm for group insertions is the total of the complexity of the following phases: sorting the key list of m keys in $O(m \log m)$ time, locating the k position leaves in $O(k \log n)$ time, performing the subtree construction in $O(m)$ time, attaching the subtree to the correct position in $O(1)$ time, performing $O(\log^2 m)$ rotations and traversing at most $O(k \log n)$ steps upwards in the tree.

$$\begin{aligned}
&O(m \log m + k \log n + m + 1 + \textstyle\sum_{i=1}^{k} \log^2 m_i + k \log n) = \\
&O(m \log m + k \log n + \textstyle\sum_{i=1}^{k} \log^2 m_i).
\end{aligned}$$

If the sorting is done externally, the complexity is $O(k \log n + \log^2 m)$, but for internal sorting, as $m > \log m$, the complexity can be expressed as $O(m \log m + k \log n)$. In all of these expressions, $k \leq m \leq n$.

For comparison, a similar worst-case expression can be written on the complexity of a sequence of m strictly balanced single operations. This consists of the $O(m \log n)$ traverses, $O(m)$ leaf splits, $O(m)$ rotations and $O(m \log n)$ traverses during balancing. The total is therefore $O(m \log n)$. Using the inequalities $k \leq m \leq n$ it is evident, that

$k \log n \leq m \log n$ and $m \log m \leq m \log n$. The problematic comparison is the sum expression for the rotation cost. Judging by the above calculations for $m = 500$, it seems to be safe to expect that it does not rise much above m but can be significantly less. Thus it is of similar or smaller magnitude with $O(m)$ for the single case. Therefore the group algorithm with relaxed-balancing has practically either a lower or the same upper bound than the single case.

Chapter 7

Experiments

This chapter documents the performance analysis conducted on EBST-based indices. The variations included in the experiments are listed in Section 7.1 together with system parameters. The metrics of the evaluation are discussed in Section 7.2 and the implementation used in the experiments is described in Section 7.3. Section 7.4 explains the measurements made and the test environment. The results of the experiments are reported and analyzed in Section 7.5 and the outcome of the evaluation analyzed in Section 7.6.

In discussing previous versions of group-update algorithms for relaxed-balance search trees, the main question is whether the performance is better than in the traditional strictly coupled single updates. Malmi [33] conducted comprehensive experiments on the performance of group and singular operations, investigating whether the group approach speeds up the processing in practice and if so, in what circumstances. He found that the group approach performed better for a dense distribution on the queried keys and worse on a sparse distribution.

One of the main goals in developing a new algorithm is to provide such a solution that will not perform worse than single updates under any conditions, i.e. will not introduce computational overhead for an even key distribution, but will preserve the savings made by Malmi [33] achieved for a dense distribution.

The mathematical analysis of the previous chapter can be used to prove the correctness and to provide an upper bound to the time and space requirements of the algorithm, but modeling and experiments are necessary to investigate the influence of key distributions and such factors with reasonable effort. A mathematical analysis could not provide information that would be more useful than the trends observed in practical experiments, but it would be much more tedious to conduct. Malmi [33] points out that “especially worst-case analysis can give too pessimistic bounds for the running time of an algorithm,” which is also a threat in my analysis. Clearly a more realistic comparison can be made when also experimental approach is taken.

7.1 System Variations

In order to conduct reliable experiments, a careful design of both the experiments and the model used is necessary [22]. In this thesis the goal of the experiment is to compare the performance of the defined algorithm to other possible approaches. Three system variations are chosen for measurements:

- Single updates with strictly coupled balancing
- Group updates with strictly coupled balancing
- Group updates with relaxed balancing

The relaxed balance for single operations is omitted as the relaxed-balance algorithm is not suitable as is for traversing the tree more than once before balancing. Marking of the branching nodes is problematic in single case: the marks signifying left and right proceeding can not be initialized to be false within the locate phase, as the tree is traversed more than once per a series of single keys. A separate procedure would be needed to clear a tree of these marks. This is quite inconvenient, but constructing a separate single-case relaxed-balance algorithm was not of interest here.

The traversing and modifying the tree needs to be done as similarly as possible, using the same procedures for traversing the tree in the locate phase in both update modes, and the same criteria and implementation for the standard rotations in both balance modes. The same locate phase implementation is used for the group approaches and another one for the single case. Similarly the group cases share the update phase implementation use. However if the subgroup of a position leaf is of size one, the leaf-split procedure of the single case is employed.

All three variations use the same inner balancing procedure of Section 5.4. It balances one group entirely if the check for branching is coupled with a check for the balance mode, so that the system does not stop when there is only one group to balance.

The model needs to generate a specified number of keys using different distributions to create them. The implementation is such that once the parameters are set, a specified number of test sequence could be ran and the significant figures of the processing could be recorded. The parameters of an operation sequence are summarized in Table 7.1.

TABLE 7.1: System parameters in measurements.

Parameter	**Value**
Initial tree size	100 000
Key range Operations per sequence	1 – 5 000 000 300
Size of a group	500, 1 000, 2 500
Key distribution	Even and generated
Operation type	Insert
Operation mode	Group and single
Balance mode	Strict and relaxed

To limit the number of combinations, the initial tree size is chosen a constant value. It would of course be interesting to compare the behavior of the four strategies in trees of different sizes, but the goal of these experiments is more limited and one tree size will suffice for my purposes here. The implementation would however allow the use of different initial tree sizes, which leaves the door open for later experimentation. Those parameters that are varied are called *factors* [22].

The selection of tree size, key range, and number of keys per operation sequence are reduced from the figures used by Hanke with red-black trees [19]. It would be desirable to examine the performance of the system variations for vary large trees and long operation sequences, but due to limitations in time and hardware, I settled for figures not trivially small (not calculated by hand) but descriptive of the actual situation.

The group sizes used were selected to match the estimations of Ylönen [54] and other researchers for some common applications of indices; a group of 5 000 keys corresponds to a small article in full-text indexing, a natural batch size is around 10 000 for many other applications as well. Thus as normally n is some millions and I use $n = 100\,000$, I also scaled down the real-life group sizes dividing them by ten. As the operation count is 300, for group size 500, $150\,000$ keys are used. Similarly for $m = 1\,000$ the number of keys is $300\,000$ and for $m = 2\,500$ the count is $750\,000$. This is also valid for single operations as explained below.

The initial trees are constructed with an even distribution, which was also chosen by Malmi [33] for the same main reason: the effect of the key distributions is more easily derived as the initial tree is filled randomly rather than using a pattern. Neither the initial tree nor the groups of keys are allowed to contain duplicate values; therefore also in single operation tests, the keys are generated as alike sequences to remove the duplicates from there as well. The only difference is that the group update uses sorted sequences, which would be unnecessarily harmful to the single case performance and thus random order is preserved there. Thus also for single operations, the number of keys inserted is m times the number of operations in a test sequence, each list of m having no duplicates.

The two distributions chosen for these experiments are the even or *uniform* distribution generated directly by `java.util.Random` of the Java API and a 'dense' distribution about $13\,000$ keys wide, not producing anything outside that area, generated by translating the words of the Bible into a sequence of integers, each of the words having a unique representation. As the location of the peak is not important, rather the existence, it is positioned at the beginning of the key space. To understand the nature of these two distributions, I constructed graphs of the sequences they generate. For the even distribution, Figure 7.1 was plotted from a million random numbers generated from the interval $(0, 100)$.

From the text-generated distribution, mainly the dense area in the interval $(1, 12\,566)$ is shown; the rest is empty for an interval with a maximum larger than the number of distinct words in the Bible. This is shown in Figure 7.2. The total number of words in the Bible would limit the number of operations per sequence, as after about $750\,000$ keys the sequence restarts, the peak is shifted by $12\,566$ after the first cycle not to overlap. Thus the dense distribution in a sense traverses the key space, wrapping at the end if necessary. As the largest m used is $2\,500$ and the operation count is 300, the word list will not need to wrap in these experiments.

The implementation also allows other distributions, as explained in Section 7.3; I hope to be able to examine the performance of the algorithm for various real-life indexing distributions in future work. A thorough analysis of more than two distributions would however expand this work beyond a Master's thesis.

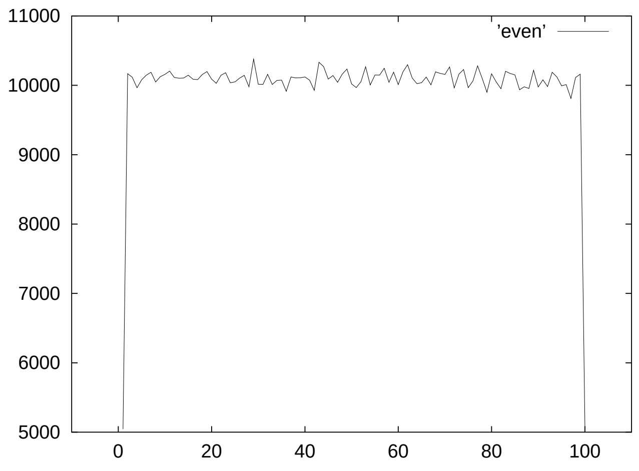


FIGURE 7.1: The even distribution in $(1, 100)$ generated by a million operations.

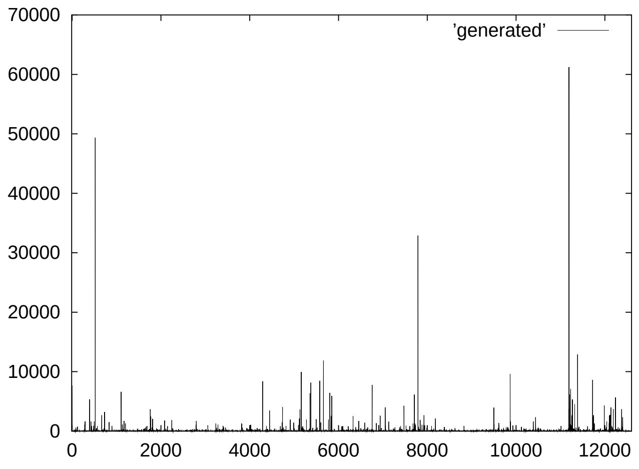


FIGURE 7.2: The distribution generated by using distinct words of the Bible as keys generated by the first $750\,000$ words.

The even distribution was the natural choice to be included, as it is the same one used in most of the formal analysis and it was also found by Malmi [33] to provide for the worst performance for a group update algorithm. Thus if my algorithm can perform fairly with the even distribution, it should not go below the single case performance in other, more favorable circumstances. The second distribution was used to provide at least a touch of realism by using actual English text to generate the key stream and also, as the number of distinct words used in the Bible does not cover the modern English entirely, it would cause peaks in actual full-text indexing as well. Although now it provides for only one large peak, as the number mapping does not account for the distribution of these words into all of English vocabulary. The Bible was chosen as the text source, as it well available as plain text on the Internet and is sufficiently long.

As for the index operations, I have chosen to run only insert operations, as each insert already includes a search operation; the search operation does nothing more, only less than an insert, and the link traverses in the locate phase of an insertion match exactly those done in the search of the same key list. Also deletions are omitted as marking the leaves as invalid is reducible to a search operation plus a constant-time change of one bit, and no practical implementation for the group deletion was constructed in this thesis or found in literature.

With the factor values of Table 7.1, excluding the problematic single-relaxed –combination, the outcome is 18 combinations as the each of the factors needs to be singled out in order to draw valid conclusions [22]. Each of these experiments is iterated until a defined confidence interval of is met. The information recorded per each test run is listed in Table 7.3 in Section 7.3.

7.2 Metrics

As in the mathematical analysis, there are three issues to address in these experiments: the length of the traversed path in the locate phase, the number of nodes visited in an update operation, and the number of rotations made during rebalancing [5]. Thus the model implementation needs to record at least the following information:

Path length: the number of links traversed per operation

Locality: the number of links traversed per successful operation during balancing

Balancing effort: the number of rotations made per operation sequence

The number of rotations cannot be recorded per each single operation, as the relaxed balancing will only be performed after some sequence of operations. The *metrics for* performance of an entire operation sequence would then be the following:

- The number of link traverses per key while locating the position
- The number of link traverses in balancing per successfully inserted key
- The number of single and double rotations per successfully inserted key

Also, as a curiosity, the number of keys per position leaf and the number of successful operations per position leaf are recorded alongside with these metrics, as the number of position leaves is a critical factor in the formal analysis of the algorithm.

By conducting the same operation sequence using the same key stream with all of the variations will give comparable figures for each of these, and running the same operation sequence with several different key streams all generated by the same distribution will give statistically meaningful figures for comparing the behavior of the four variations. For each of these criteria, the smaller the value, the better the performance. The selection of these metrics and the preceding factors has been set to meet those used in previous, similar work. One common performance metric is the run-time of an algorithm, but as the model implementation operations also include the recording of the measurement information and it does not handle an actual database, the run-time values are not very realistic and will not be used in evaluating the performance of the variations.

7.3 Description of Implementation

For evaluating the performance of the algorithm I have implemented a small-scale Java library on the problem area. The library includes an implementation of the EBST discussed in this thesis complete with integer keys, traversing and the simple rotations used in rebalancing, and a tools for generating keys with determined distributions and running test operation sequences on the structure. There are two operation modes: the single update mode, inserts one key at a time, and the group update mode for batches of keys. In both modes the balancing may be either coupled with the update operation or triggered later as relaxed balancing. All four combinations are evaluated in the experiment set.

As the details of concurrency control, discussed in Section 3.2 are not a part of this thesis, only one operation sequence is operating on the index tree at a time. The sequence can be defined with a parameter file. The parameter sets is the same than was defined in Table 7.1, although the implementation supports more variability — the selection of factors is not hard-coded into the implementation, but may be altered.

7.3.1 Packages

The implementation is divided in four packages, briefly described in Table 7.2. For each package a short overview of the contents and functionality follows, but the details of the implementation are not discussed or the Java program code included.

TABLE 7.2: The Java library packages.

Package name	Brief description
`index.ebst`	EBST implementation
`index.operation`	Control structures for EBST update and balancing
`index.test`	Experiment setup and measurement tools
`index.distrib`	Distributions for integer keys

The package `index.ebst` consists of only one class hierarchy, defining the class of external binary search trees. The abstract superclass `Node` provides for the attributes common for all nodes: the height of the node, the key or routing value, and the parent link. Each node is able to modify the parent link, request for its own removal from its parent and forward a key of a group of keys to the correct direction in the tree. The class also includes a static operation for constructing EBSTs from a list of integer keys. Routers and leaves are implemented correspondingly in `Router` and `Leaf`. Each router can perform the standard rotations using itself as the starting point of the rotations. The only special property in leaves is that they can split themselves into two leaves and a router as their parent. This is needed for single insertions and for groups of size one.

The package `index.distrib` only includes the inheritance hierarchy of distributions. The abstract superclass `Distribution` contains the information on the minimum and maximum of allowed key value and a random number generator needed for properly varying distribution. Its subclasses are required to be able to produce decimal values from the interval $(0.0, 1.0)$. By resorting to this production, the superclass provides methods to generate either just one or a sequence of integer keys with no overlapping values, using a source following a distribution defined in one of the subclasses. The standard distribution source provided in the Java API is the even or uniform distribution in `EvenDistribution`.

A parser class `UserDistribution` is provided for using other distributions. The user-defined distributions are given to the tester as a text file, providing in left-to-right order the probability distribution in the interval $(0.0,\ 1.0)$. The distribution is approximated with rectangles of arbitrary length, the column height being the probability of a key belonging to the subinterval from the starting point of the column to the end point. For each column, the end point and the probability are given, expecting that the first rectangle begins at the origin.

As the probabilities of these pillars should add to exactly 1.0, the width of the last column is not needed, but can be calculated. Also the end point for the last column must be 1.0. By using a random number generator in the same basic interval $(0.0, 1.0)$, I select an interval from the user-defined distribution directly based on the random value, thus obtaining the wider columns more often, accordingly to the distribution. Then I generate another random number and scale it to be a key value from the relative interval of the defined distribution.

The most complex package is `index.operation`, which includes the classes of operating on an EBST and recording status information. The `Controller` class takes care of the operation and balancing modes and processes index operation requests located at given position leaves with given data. The balancing is supervised by `Balancer` either strictly coupled or relaxed. Information on the current state of the tree, number of link traverses in each locate phase, and other such data is recorded. A complete list of the recorded statistics is given in Table 7.3.

The last of the four packages is separate from the other three in the sense that the other three depend on each other to provide the necessary functionality for an index tree, they are completely unaware of the classes in package `index.test`. These classes interpret the parameters, run the test sequences and compute the results from the output statistics. The runnable class is `Tester`, which loads the parameter sets and uses in turn `Operation` to operate on an EBST instance according to a specific set of parameters. This package also contains the `Metric` class for analyzing the operation sequences.

7.3.2 Operating on the Index Tree

Figure 7.3 explains the basic work flow between the classes: the experiment sequence begins in `index.test.Tester`, where the parameter sets are sent to `Operator` in the same package for generating operation sequences with the given parameter definition. `Tester` controls that each sequence is iterated until a desired confidence interval is met with defined accuracy, at least 30 times and at most 90, stopping as the requirements are met. The next step is that the `Operator` sets up the initial tree and runs the amount of operations defined in the parameters. After each round, it records the statistics for that operation sequence and informs the `Tester` of the current figures of statistical reliability.

An index operation is performed by first setting the operation mode for `Controller` in package `index.operation` and sending properly generated key material to the tree from the root node, an instance of `index.ebst.Node`. The data traverses the tree until it arrives at a position leaf, where it is forwarded to the `Controller` to process (i.e. to return the result of a search or perform the insert). After the possible update has been

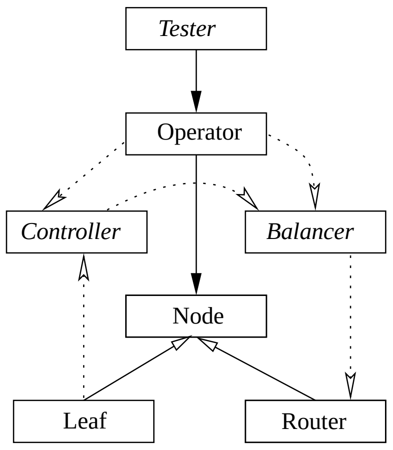


FIGURE 7.3: The main structure of the model implementation.

done, balancing takes place. For strict balancing, the `Controller` calls the `Balancer` of the same package, which in turn requests for rebalancing of individual nodes in the tree, instances of `index.ebst.Router`. For relaxed balancing, the positions are queued in `Balancer` and balancing is initiated by the `Operator` after a group of operations (one group operation or a series of single operations of the same size).

Some of the classes, as `Distribution` and its subclasses for generating integer keys and the `Statistics` class for recording information are not shown in Figure 7.3 for simplicity. The classes with names in italics are static and the others owned and instantiated by those referring to them.

The `Operator` uses a utensil class `Metric` to control the statistical measures of the test runs. The functionality of `Metric` follows closely of that used by Esko Nuutila [42]. For each metric, an instance of the class is created that holds e.g. the number of samples, their sum and square-sum. From the stored information, mean value, standard deviation, variance, etc. are calculated, as recommended by Jain [22] to avoid drawing conclusions from only the average of the samples.

Whether a desired accuracy has been reached can be determined in two ways: ϵ is the upper limit to the ratio between the half-length and the mid-point, whereas δ is the upper limit to the amount of error in an estimate. If the collected data meets at least one of these limits, and the initial 30 iterations have been ran, the application may move on to the next factor configuration. The iteration thus ceases when either the relative error ϵ or the absolute error δ becomes sufficiently small. The probability that a sample belongs into the confidence interval is selected through α; the probability is $100(1 - \alpha)\%$.

Thus the confidence interval is defined with three variables, α, δ, and ϵ, defined to each metric separately as constructor parameters. The program code that performs the calculation and evaluated the confidence level during run-time is adopted from Nuutila [42] as well by translating the C++ code into Java.

For the measurement set, α and ϵ for both given the value 0.025 for each metric. The even distribution test runs would easily meet tighter bounds, but the dense distribution expresses irregularity that causes oscillation in the statistics.

The absolute error δ cannot be defined the same for all, as it depends of the expected magnitude of the measured value. With the number of links traversed per key, the average value is of the same magnitude than the tree height $1.44\ \log n - 0.328$, which for $n = 100\,000$ gives approximately 24 and the tree grows throughout the test sequence. As the actual number of link traverses is an integer, it is sufficient to know the result of the metric in the accuracy of an integer. Thus $\delta = 0.025$ was chosen for the both of the link traverse metrics, as the path in balancing is of the same order than the path in locate, although hopefully with a smaller coefficient, as it should end earlier.

TABLE 7.3: Information stored per operation sequence.

Phase	**Information**
Locate	Number of links traversed
Update	Number of successful operations
	Number of position leaves
Balance	Number of links traversed
	Number of single rotations made
	Number of double rotations made

For the metric measuring the number of rotations, i.e. the link reorganization related to rotating. Each single rotation requires three link updates and each double rotation four (see e.g. Section 2.5). The magnitude of the total number of rotations is relative to the group size by $O(m)$ in single insertions and $O(\sum_{i=1}^{k} \log^2 m_i)$ in relaxed balancing. Dividing the total number of link updates in rotations with the number of inserted keys, which is close to the group size, should give values close to one. As this is a smaller value than the others, a tighter $\delta = 0.00025$ is chosen to ensure similar accuracy for all metrics.

7.4 Measurements

The experiments were conducted on a workstation with a single 800 MHz Pentium III processor running Java 1.3 on Red Hat Linux, release 6.2. The cache size was 256 KB and there was 512 MB of main memory. As processor ticks or other temporal measures are not used to evaluate the performance, the details of the equipment are not significant. For the same reason, it was not necessary to eliminate all other computing from the workstation at the time of the test runs. They were mainly run on the background, as the application is very independent in performing the test runs.

As the model application starts by executing `Tester` in package `index.test`, it reads in application parameters from a file whose name is given as a command line argument. Currently this file only holds the name of the file containing system parameters for the text runs. The class `Operator` of the same package first parses the system parameters and factors from that file and then begins a loop, in which an initial tree is first generated and then a sequence of operations is performed on that tree accordingly to the system parameters. This is repeated until the desired confidence is attained for the metrics, namely the number of link traverses in locate and balance, and the number of rotations. More metrics are easily added into the implementation as necessary.

For each parameter set, the application records the significant figures of each iteration, 30

at minimum, 90 at most, in an output file. At the end of each such file it appends the results from the metrics; all the metric information, most importantly the mean value and the standard deviation. These results are also summarized together from all parameter sets in a single file for simplicity of further processing.

The results of the test runs were used to construct the tables and figures of the following section. The processing could easily be automated and integrated in the model application, but it is beneficial to look at the plain figures and check whether they look reasonable before analyzing the actual outcome. Even though the implementation has been tested and examined intensively, it is hardly ever certain that a program would not face on unexpected situation that distort the results.

7.5 Test Results

For each of the 24 parameter sets, the three metrics were recorded and analyzed. As these sets are identified by the system variation, distribution, and sequence size, I will review here the system variations, metrics, and the recording procedure. The three variations measured are:

- Single updates with strictly coupled balancing
- Group updates with strictly coupled balancing
- Group updates with relaxed balancing

The three metrics measured are:

- The number of links traversed per key in locate phase
- The number of links relocated as rotation work in balance phase
- The number of links traversed per inserted key in balance phase

For the group variations, also the number of keys located in the same position leaf is of interest, as the number of position leaves was relevant in the algorithm analysis of the locate phase. As not all keys are successfully inserted, I also recorded the size of the subgroups actually inserted into the index.

The values for the metrics are calculated over an entire operation sequence, so direct comparison of different group sizes on the same distribution and system variation is not reasonable. As the group size changes, also the rate of tree growth changes. The resulting tree for the smallest group size is somewhat below 250 000, whereas for the largest group size the resulting size is around 850 000. Thus comparing the number of e.g. link traversals in otherwise the same setup gives a pessimistic trend, as the tree growth introduces a longer absolute path from the root to the leaves. The main intention of the experiments is to compare the different system variations for two different distributions using three group sizes and not to examine the trends within a system variation.

When interpreting the measurement results, it is also important to keep in mind that the dense distribution does not exhibit any pattern as less keys are used than obtained from the bible. Thus also the details of the key distribution change along with the group size.

In this section, the collected data is presented and explained. Conclusions are not drawn until the next section to provide a more objective view on the situation, as all of the data can be viewed first.

7.5.1 Single Operations

The test runs with the even distributions could have easily met even unnecessarily tight confidence levels, as the distribution met its definition well and the metrics used were directly measurable instead of resorting to approximation. Table 7.4 summarizes the mean (average) values and the standard deviation for the three metrics. The effect of tree growth is visible here; as the key distribution is uniform and the initial tree size only ten percent of the key space, an insertion is quite likely to succeed. Thus the larger the group size, the faster the growth, and the longer the search paths. Also the number of link updates (three in a single rotation, four in a double rotation) grows moderately as the tree grows, but in a lower order of magnitude than the search path. This is because the number of rotations is not directly dependent on the tree size.

TABLE 7.4: Statistics for single insertions with even key distribution

Metric	Group size	500	1 000	2 500
Path per key	Mean value	18.41	18.83	19.53
in locate	Std dev.	$1.67 \cdot 10^{-3}$	$1.93 \cdot 10^{-3}$	$1.51 \cdot 10^{-3}$
Link updates	Mean value	1.567	1.587	1.606
per insertion	Std dev.	$5.10 \cdot 10^{-3}$	$4.29 \cdot 10^{-3}$	$3.46 \cdot 10^{-3}$
Balancing path	Mean value	3.768	3.774	3.779
per insertion	Std dev.	$1.85 \cdot 10^{-3}$	$1.50 \cdot 10^{-3}$	$1.13 \cdot 10^{-3}$

As the tree size was chosen as $100\,000$, and $\log_2 100\,000 \approx 16$ and the corresponding AVL height from Theorem 2.1 is approximately 24, it can be seen that the calculated path lengths match their expected range.

The number of link updates can be related to the number of rotations by signifying one rotation as 3.5 link updates, average from single and double rotations, roughly of equal probabilities. Thus as the probability of a rotation, calculated from the values of Table 7.4 is approximately 45 percent. This is again a good match to Theorem 6.11, estimating the probability below 0.56 and guaranteeing it to be above 0.37.

In the model implementation also the last traverse to the parent node is counted to be a part of the balancing path, although it is not traversed as a rotation is already performed. This is because the proceeding to the parent is included in the rotation selecting procedure presented in Section 5.4, used both for the single and group operations. Therefore, by subtracting that one traverse from the figures of the table, the resulting values are very close to the amortized bound 2.618 given in Theorem 6.12.

As all of the metrics meet their analytic estimates so well, I trust the model implementation and the measurements to be reliable. A more difficult case is the dense distribution generated from the Bible, as it exhibits more irregular behavior. Those values are summarized in Table 7.5.

TABLE 7.5: Statistics for single insertions with dense key distribution

Metric	Group size	500	1 000	2 500
Path per key	Mean value	16.87	16.86	17.01
in locate	Std dev.	$1.65 \cdot 10^{-1}$	$1.37 \cdot 10^{-1}$	$2.21 \cdot 10^{-1}$
Link updates	Mean value	1.741	1.786	1.941
per insertion	Std dev.	$2.23 \cdot 10^{-2}$	$2.85 \cdot 10^{-2}$	$3.74 \cdot 10^{-2}$
Balancing path	Mean value	3.818	3.831	3.878
per insertion	Std dev.	$7.48 \cdot 10^{-3}$	$9.20 \cdot 10^{-3}$	$1.21 \cdot 10^{-2}$

The striking feature is that the path length is smaller than in the corresponding even distribution. This is due to the short interval from which the keys are selected: there are only 12 566 different possibilities, so the majority of the operations fail and the tree does not grow as it does in the even case. This also drops the number of successful insertions to almost exactly 12 566 per sequence, if the key sequence happens to include every word of the Bible once. Thus the standard deviations are larger, as the number of successful samples has decreased. They are however still sufficiently small not to inhibit profitable analysis.

As already pointed out, the tree growth is much slower and the differences in location path length minor. However, as the insertions go in the same portion of the tree, a little more than ten percent of the tree width (100 000 versus 12 566), that one portion is the only one that grows. Thus rotations are more likely to occur and at a higher level. The amount of rebalancing work appears to grow more rapidly with the group size for the dense distribution.

Keeping in mind that the shorter location paths are only a result of slower growth (i.e. more failed operations), it is evident that the single-case strict-rebalance system performs best for an even distribution.

7.5.2 Group Operations

The test sequences with group update used both of the balancing schema. First, as a better analogy to the single-case, I present the data collected from the strictly balanced group operations. They used the exactly same locate and update phase implementations than the relaxed-balance version, but only called the group rebalance operation once from the update position, ignoring all branching points and proceeding upwards until balance is achieved. Table 7.6 summarizes the results, now also containing the size of the subgroups in the locate phase and the actual number of keys inserted per position. The reason for presenting both is that some of the insertions of course fail, thus causing the actually constructed subtrees to be smaller than the subgroup size would suggest. The subgroup size itself is interesting as it helps to understand how many operations have failed.

Here the path per key drops as the group size grows, even though the tree grows in the same rate than for the evenly distributed single case. The rate is the same as both operation sequences use key lists that contain no duplicates. This is essentially the reason why the concept of group size was introduced on the single operations as well. Thus, as it is known that the tree grows from the single case, it is evident that the savings made by resorting to group update increase with group size and even more for larger trees. The number of

TABLE 7.6: Statistics for strictly coupled group insertions with even key distribution

Metric	Group size	500	1 000	2 500
Path per key	Mean value	9.54	8.95	8.31
in locate	Std dev.	$3.97 \cdot 10^{-3}$	$2.84 \cdot 10^{-3}$	$1.88 \cdot 10^{-3}$
Link updates	Mean value	1.572	1.596	1.620
per insertion	Std dev.	$4.54 \cdot 10^{-3}$	$3.34 \cdot 10^{-3}$	$2.14 \cdot 10^{-3}$
Balancing path	Mean value	4.664	4.683	4.699
per insertion	Std dev.	$3.76 \cdot 10^{-3}$	$2.99 \cdot 10^{-3}$	$2.09 \cdot 10^{-3}$
Subgroup size	Mean value	1.0027	1.0040	1.0058
in locate	Std dev.	$1.21 \cdot 10^{-4}$	$1.17 \cdot 10^{-4}$	$8.57 \cdot 10^{-5}$
Keys inserted	Mean value	1.0000	1.0000	1.0001
per position	Std dev.	$1.00 \cdot 10^{-5}$	$1.08 \cdot 10^{-5}$	$1.45 \cdot 10^{-5}$

rotations is almost exactly the same as in the respective single case; the number of re-linkings is only one hundredth part larger in the strictly coupled group case. The probability of a rotation per key is therefore only couple of percent higher.

The length of the rebalancing path has grown by a little less than one, due to the occasional downward proceeding in the sibling subtree of the originally imbalanced one. special Also the treatment needed after double rotations in group update explained in Section 5.4 is a probable cause of the risen balance-traversing costs. The balancing path still increases very moderately with the tree size, as in the single case.

The dense distribution was also used to examine the behavior of the group update. As in prospect by the algorithm analysis and the results of Malmi [34], the dense distribution gains even larger savings in the locate phase, as shown in Table 7.7.

TABLE 7.7: Statistics for strictly coupled group insertions with dense key distribution

Metric	Group size	500	1 000	2 500
Path per key	Mean value	4.73	4.26	4.15
in locate	Std dev.	$1.31 \cdot 10^{-1}$	$6.27 \cdot 10^{-2}$	$2.49 \cdot 10^{-1}$
Link updates	Mean value	1.837	1.905	1.964
per insertion	Std dev.	$2.33 \cdot 10^{-2}$	$2.71 \cdot 10^{-2}$	$1.17 \cdot 10^{-1}$
Balancing path	Mean value	4.727	4.695	4.147
per insertion	Std dev.	$6.48 \cdot 10^{-2}$	$7.72 \cdot 10^{-2}$	$2.49 \cdot 10^{-1}$
Subgroup size	Mean value	1.0178	1.0208	1.0174
in locate	Std dev.	$3.69 \cdot 10^{-3}$	$3.79 \cdot 10^{-3}$	$4.80 \cdot 10^{-3}$
Keys inserted	Mean value	1.0258	1.0451	1.2310
per position	Std dev.	$1.60 \cdot 10^{-2}$	$1.87 \cdot 10^{-2}$	$7.81 \cdot 10^{-2}$

The locate path drastically drops to about one half of what it was for the even i.e. sparse distribution. The balancing path is of the very same magnitude, but the amount of rotation work rises. This is the same effect as in the corresponding single-case transition from even to a dense distribution: as more of the insertions come on the same area, that area overall grows more rapidly than the neighboring areas, causing more rotations. Also note that the sizes of the subgroups and consequently the constructed subtrees grows as the inserted keys come from a thin interval in the key space.

The last two tables contain the data collected from the third system variation, the complete algorithm of Chapter 5: group updates with relaxed rebalancing. Many of the data is naturally almost identical to the strictly-coupled group update, as the only difference is in rebalancing.

TABLE 7.8: Statistics for relaxed-balance group insertions with even key distribution

Metric	Group size	500	1 000	2 500
Path per key in locate	Mean value	9.54	8.95	8.31
	Std dev.	$2.93 \cdot 10^{-3}$	$2.95 \cdot 10^{-3}$	$1.79 \cdot 10^{-3}$
Link updates per insertion	Mean value	1.569	1.592	1.612
	Std dev.	$5.69 \cdot 10^{-3}$	$3.30 \cdot 10^{-3}$	$2.80 \cdot 10^{-3}$
Balancing path per insertion	Mean value	5.597	5.595	5.576
	Std dev.	$4.80 \cdot 10^{-3}$	$3.46 \cdot 10^{-3}$	$2.99 \cdot 10^{-3}$
Subgroup size in locate	Mean value	1.0027	1.0040	1.0058
	Std dev.	$1.13 \cdot 10^{-4}$	$1.31 \cdot 10^{-4}$	$1.01 \cdot 10^{-5}$
Keys inserted per position	Mean value	1.0000	1.0000	1.0001
	Std dev.	$8.65 \cdot 10^{-6}$	$8.72 \cdot 10^{-6}$	$1.17 \cdot 10^{-5}$

The number of link updates is a little smaller than in the strictly coupled case, but only less than one hundredth part. This it can be concluded that the rotation work definitely does not increase when relaxed-balance is taken into use. The balancing path increases by a little less than one step, which is due to the branching point checks. When checking for balance in a branching point, the procedure `balanceRouter` retrieves the parent node just as for any other node, even when the branching node was in balance to begin with. If the implementation only calculated those link traverses that were actually used to proceed in the balancing, the figure would be smaller. Nevertheless, as the number of branching points cannot exceed the number of position leaves, which is a little less than the number of keys judged by the subgroup size, removing this overhead is not likely to result in exactly the same balancing path.

For the dense distribution with relaxed balancing, the likelihood of a rotation again increases due to the locality of the insertions on only one area. At first it seems surprising that also the balancing path appears to drop, but the slower rate of tree growth must be taken into consideration. It is natural that when the tree is smaller, the path length drops some. This however does not explain why the balancing path gets smaller. The same trend is apparent in all of the group variations with both distributions.

One influential factor for the relaxed case is that the subtrees inserted in sibling positions ’cancel out’ each other’s effect, but this is not the explanation as the phenomena is also inherent in the strictly-coupled group updates. The actual reason here is that the dense distribution is combined with a larger group size, the inserted subtrees begin to grow larger. This is apparent from the statistics for keys inserted per position. The dependency of rotation cost in subgroup sizes begins to be visible and causes the observed decrease, although the subgroups are still generally quite small.

Note that the match between the strictly-coupled and relaxed case is not as exact with the dense distribution as it was with the even distribution. This is because of the irregularity of the dense distribution; depending on the position of the distribution reader in the source data, essentially different key lists are created within the thin interval and the amount of successful insertions depends on the initial tree, generated again for every test run. There is

TABLE 7.9: Statistics for relaxed-balance group insertions with dense key distribution

Metric	Group size	500	1 000	2 500
Path per key in locate	Mean value	4.69	4.25	3.52
	Std dev.	$1.33 \cdot 10^{-1}$	$8.46 \cdot 10^{-2}$	$1.26 \cdot 10^{-2}$
Link updates per insertion	Mean value	1.779	1.791	1.681
	Std dev.	$3.11 \cdot 10^{-2}$	$4.02 \cdot 10^{-2}$	$2.71 \cdot 10^{-2}$
Balancing path per insertion	Mean value	5.248	5.049	4.068
	Std dev.	$1.08 \cdot 10^{-1}$	$4.02 \cdot 10^{-2}$	$7.36 \cdot 10^{-1}$
Subgroup size in locate	Mean value	1.0188	1.0218	1.0149
	Std dev.	$4.01 \cdot 10^{-3}$	$5.19 \cdot 10^{-3}$	$1.79 \cdot 10^{-3}$
Keys inserted per position	Mean value	1.0277	1.0506	1.1875
	Std dev.	$1.74 \cdot 10^{-2}$	$2.57 \cdot 10^{-2}$	$2.21 \cdot 10^{-2}$

also a curiosity here: the number of link updates suddenly drops for the largest group size, when in all of the other variations it grows together with the group size. This is again due to the increasing subgroup size: as less rotations are needed altogether, the path traversed drops as well.

7.6 Evaluation

To fluently compare the performance of the system variations, I constructed diagrams of the recorded metrics. The histograms of Figure 7.4 show the length of the path traversed in locate phase by the single-case (with darker columns) and the group approach (with lighter columns) for each group size m. It is evident that as the even distribution is the sparsest distribution available (making it sparser somewhere would cause dense intervals elsewhere), the group operations will locate the position leaves with half or less of the effort that it takes for the single operations to reach the position leaves. With the generated dense distribution, the savings are nearly 75 percent. The graph was generated from the corresponding tables in the previous section by rounding the values to the nearest half.

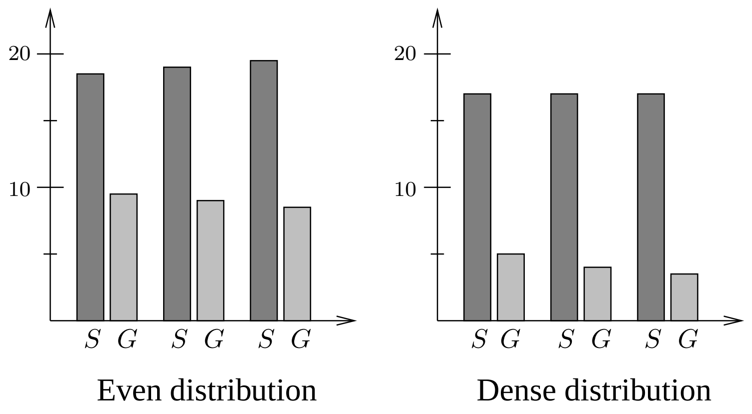


FIGURE 7.4: The number of links traversed in the locate phase for the single-operation S and the group-operation G algorithms with $m \in \{500, 1\,000, 2\,500\}$.

Similarly, the number of link updates per inserted key is illustrated in Figure 7.5, constructed from the tables of the previous section rounding as necessary. Note that the value axis are neither of equal spacing nor position. One rotation can again be considered about 3.5 link updates.

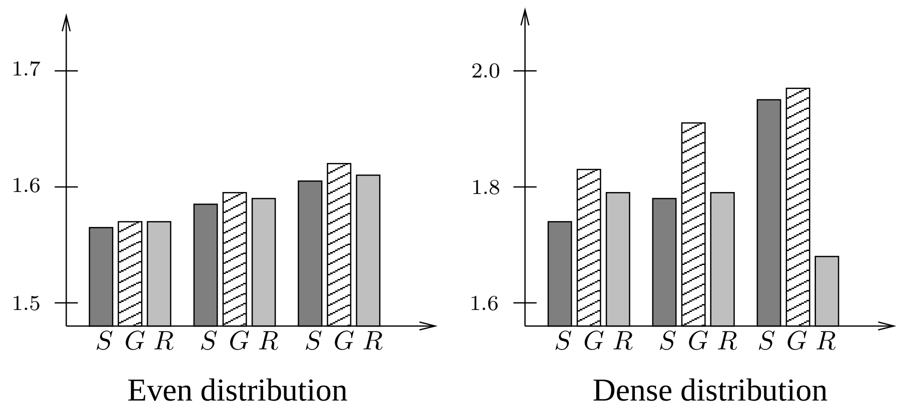


FIGURE 7.5: The number of link updates in the balance phase for the single-operation S, the strictly coupled group-operation G, and the relaxed-balance group-operation R algorithms with $m \in \{500, 1\,000, 2\,500\}$.

It is apparent that the number of link updates and thus the probability of a rotation grows with the group size, as the rate of tree growth is directly dependent on that. In a larger tree, there are more possible balance conflicts to resolve. The differences between system variations are only about one percent. These results do not give reason to make general assumptions on the superiority of one approach to another, as the differences are of such small order. It is safe to conclude that the number of rotations is fairly constant no matter how a certain number of keys is inserted in a tree of a certain size, as long as there is essentially one key per position leaf. The assumption of small subgroup size leads from the fact that single operations always have exactly one key per position at a time, and the statistics presented in the previous section show that the average size of a subgroup is practically one for even distribution and does not rise higher than 1.23 even with a dense distribution covering a little more than 10 percent of the key space. The behavior of the dense distribution in Figure 7.5 seems quite erratic for the relaxed case, which is due to the increasing subgroup size as discussed in the previous section.

What comes to the third metric, the path length in balance phase, the group size seems to have no effect when using the even key distribution; the difference between system variations stays nearly constant. The relaxed-balance group-update algorithm spends the most effort in traversing the tree due to the branching points in the queue. The strictly coupled group approach also requires some extra traversing caused by the downward traverse due to multiple rotations in one router. Clearly the most efficient rebalancing traverse is performed by the single-case scenario, saving a little less than one link traverse in comparison to the strictly balanced group update and a little less than two link traverses compared to the relaxed-balance group update.

As the group size has almost no effect, only $m = 500$ is drawn in Figure 7.6 for the even distribution. Again the dense distribution behaves in an unexpected fashion, and is therefore drawn for all group sizes.

In the single case, the balancing path grows a little along with the group size. The growth is more apparent than in the corresponding situation with the even key distribution. For both of

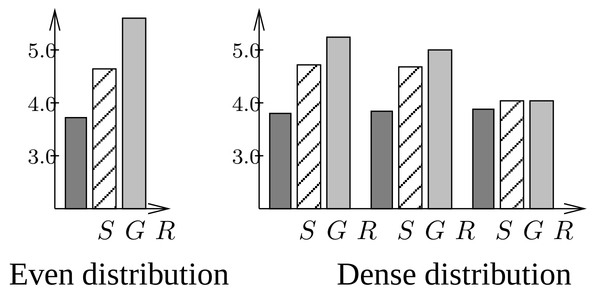


FIGURE 7.6: The number of links traversed in the balance phase for the single-operation S, the strictly coupled group-operation G, and the relaxed-balance group-operation R algorithms with $m \in \{500, 1\,000, 2\,500\}$. Only $m = 500$ drawn for the even key distribution.

the group approaches, the balancing path drops as the group size grows. This is a clear trend with both system variations, but more palpable in the relaxed-balance approach. As the keys are generated from a thin interval, the increase in group size causes the mutual distance between position leaves to shorten. As consequence of this, the common postfixes in the balancing path grow longer and the individual balancing path per subgroup gets smaller, hence decreasing the total balancing path.

Considering the three phases as an entity, it is safe to conclude that the proposed group-update relaxed-balance solution performs the best out of the three tested variations. It brings savings in the locate traverse that are multiples of the extra cost it introduces in the balance traverse, and the difference in rotation work is so diminutive that the locate savings cover that up as well. The strictly-coupled group-update algorithm causes less traversing in the balance phase, which practically comes from the fact that the balance check in a branching point increases the path length even when it does not proceed anywhere. As that is a mere implementation detail of the measurements, it cannot be regarded as an advantage over the relaxed version. The strictly coupled group approach also requires more rotation work than the relaxed alternative; it is expected on the basis of the formal analysis that this difference will grow more significant as the subgroup size increases.

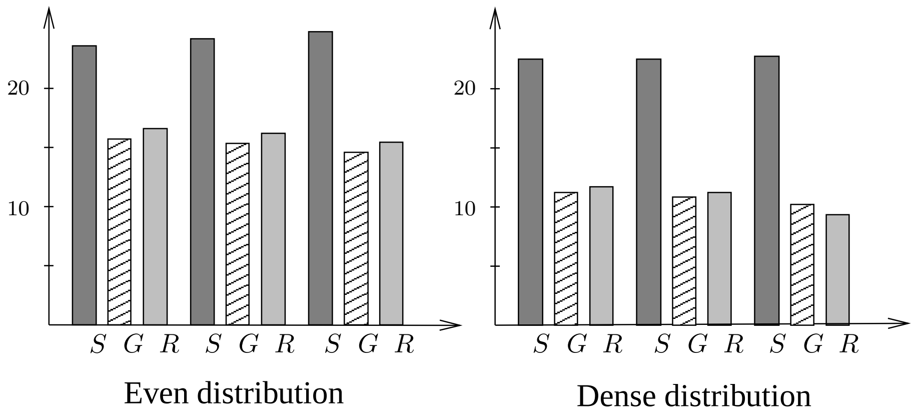


FIGURE 7.7: The total cost of the single-operation S, the strictly coupled group-operation G, and the relaxed-balance group-operation R algorithms with $m \in \{500, 1\,000, 2\,500\}$.

Calculating the measured cost of all phases together for any group size (one link update is considered as expensive as a link traverse for simplicity), the superiority of the group update is evident. This is shown in Figure 7.7. With these measurement results, the comparison relaxed balancing and strict coupling in the group case cannot be conclusively set in the favor of either one, but is left to the formal analysis. Further experiments will likely settle this in favor of the relaxed balancing, considering the possibility of neighboring subtrees compensating for each other's growth and the analytical results.

To obtain such results in the future, it would be of interest to conduct experiments that describe the behavior of the algorithm when large subgroups are the default outcome. These experiments focused on quite small subgroups as the goal was to examine whether the performance is inferior to the single case in unfavorable circumstances. Obviously, as the most common subgroup size was one, the full power of the group-update relaxed-balance algorithm is not shown, as the predicted $\log^2 m$ for the number of rotations only has an effect for larger group sizes. One way to encourage larger group sizes would be to allow duplicate keys; full-text indexing with duplicate keys accompanied by position information would easily increase the subgroup size.

Another point of further interest is to measure the true span of the rebalancing process with more metrics than just the apparent balancing path to find out what if anything causes differences in the span of the strictly-coupled update to the relaxed-balance version. The reason why group update cannot in general reach the same path length than the single case is obviously the downward proceeding caused by multiple rotations on one router, which cannot be avoided unless the standard rotations are replaced with another rebalancing schema.

Chapter 8

Conclusions

The proposed algorithm successfully combines the two improvement strategies of group update and relaxed balancing. The search of keys from the index is substantially faster when performed to sorted groups of keys than for a sequence of single keys in any order. In many application areas the sorting of the keys can be fluently done outside the index on the client side, and many applications produce data that is at least partially in order by default. It is also typical for many applications that the data arrives not individually but clustered.

The combinatorial discussion of the algorithm analysis together with the experimental results lead to the conclusion that the required path length is at most one half of the single-case path length. For a key distribution other than uniform, the savings are even larger. The savings are based on the common prefixes on the search paths and also on the possibility of two or more keys sharing the same position in the tree. In a search operation, only one of the keys in the same position leaf can be successful, but as an insert operation they can all succeed.

As each update operation begins with a search, or more descriptively, a locate phase, the same savings are made for insertions and deletions. The actual implementation of the update in a concurrent index will also require less locking in the group mode — none at best. Each position leaf is replaced by a new subtree, and the number of positions is at most the number of keys. Combinatorics again show that it is quite likely for a group of keys to have common positions, saving at least one lock per each key that shares a position leaf with another.

In the rebalancing phase, the placement of more than one key in the same position can result in savings because of the common balancing path. The relaxed-balance approach introduces new savings by allowing the insertions of adjacent leaves to balance each other even without rotations. The biggest computational advantage of the relaxed balancing is however the benefit from common postfixes in balancing paths: as two branches of rebalancing join at a router, only one of them needs to continue upwards. The proposed algorithm also presents the stopping conditions of balancing formulated for the group update, allowing to stop rebalancing as soon as all imbalance is known to be eliminated.

The main goal of the experiments was to examine the performance of the proposed algorithm in situations favorable to the single approach: keys being uniformly distributed and the subgroup sizes being very small, practically just one key per position leaf. These experiments were able to demonstrate the savings for the locate phase and to verify that the rebalancing is of very similar complexity for small key sequences regardless of whether they are inserted individually or as a group and whether they are balanced strictly after the update or together as relaxed balancing. Also a dense distribution was used in the experiments to demonstrate the increase in locate savings and the effect of position distance and subgroup size (even though still very small) to the balance complexity.

Further research is needed to examine the scale of the improvement achieved in situations closer to real-life indexing, but the experiments on the evenly distributed keys reliably shows the competitive strength of the proposed algorithm. It is also of future interest to define the locking procedures for the algorithm and to refine the balancing algorithm to allow also update operations to interleave. Further work on employing efficient group strategies in deletions or combined operation sequences is also of practical interest.

It is safe to conclude that the group-update relaxed-balance algorithm presented in this thesis does not cause noteworthy downgrade in the index performance in the most unfavorable circumstances, simultaneously enabling substantial savings under most conditions. Relaxed balancing also allows for a more flexible scheduling of the index operations through the possibility of running the rebalancing on the background and pausing as necessary. Similarly both group update and relaxed balancing ease the concurrency control of the index by reducing the locking frequency and preventing congestion, allowing more efficient use of the index.

Bibliography

[1] G. M. Adel'son-Vel'skii and E. M. Landis. An algorithm for the organization of information. *Soviet Mathematics Doklady*, 3:1259–1263, 1962.

[2] Laurent Alonso, Jean-Luc Remy, and Rene Schott. A linear-time algorithm for the generation of trees. *Algorithmica*, 17(2):162–182, 1997.

[3] Arne Andersson, Christian Icking, Rolf Klein, and Thomas Ottmann. Binary search trees of almost optimal height. *Acta Informatica*, 28(2):165–178, 1990.

[4] Arne Andersson and Tony W. Lai. Fast updating of well-balanced trees. In John R. Gilbert and Rolf G. Karlsson, editors, *SWAT 90, 2nd Scandinavian Workshop on Algorithm Theory*, volume 447 of *Lecture Notes in Computer Science*, Bergen, Norway, 1990. Springer.

[5] J.-L. Baer and B. Schwab. A comparison of tree-balancing algorithms. *Communications of the ACM*, 20(5):322–330, 1977.

[6] Ricardo Baeza-Yates, Gaston H. Gonnet, and Nivio Ziviani. Improved bounds for the expected behaviour of AVL trees. *BIT*, 32(2):297–315, 1992.

[7] Ricardo A. Baeza-Yates. Fringe analysis revisited. *ACM Computing Surveys*, 27(1):109–119, 1995.

[8] Ricardo A. Baeza-Yates, Gaston H. Gonnet, and Nivio Ziviani. Expected behaviour analysis of AVL trees. In John R. Gilbert and Rolf G. Karlsson, editors, *SWAT 90, 2nd Scandinavian Workshop on Algorithm Theory*, volume 447 of *Lecture Notes in Computer Science*, pages 143–159, Bergen, Norway, 1990. Springer.

[9] Rudolf Bayer and Edward M. McCreight. Organization and maintenance of large ordered indexes. *Acta Informatica*, 1(3):173–189, 1972.

[10] Norbert Blum and Kurt Mehlhorn. On the average number of rebalancing operations in weight-balanced trees. *Theoretical Computer Science*, 11(3):303–320, 1980.

[11] Luc Bougé, Joaquim Gabarró, Xavier Messeguer, and Nicolas Schabanel. Height-relaxed AVL rebalancing: A unified, fine-grained approach to concurrent dictionaries, March 1998.

[12] Joan Boyar and Kim S. Larsen. Efficient rebalancing of chromatic search trees. *Journal of Computer and System Sciences*, 49(3):667–682, 1994.

[13] S. Brin and L. Page. The anatomy of a large-scale hypertextual Web search engine. In *Seventh International World Wide Web Conference*, 1998.

[14] Mark R. Brown. A partial analysis of random height-balanced trees. *SIAM Journal on Computing*, 8(1):33–41, 1979.

[15] Mark R. Brown and Robert E. Tarjan. Design and analysis of a data structure for representing sorted lists. *SIAM Journal on Computing*, 9(3):594–614, 1980.

[16] Hsi Chang and S. Sitharama Iyengar. Efficient algorithms to globally balance a binary search tree. *Communications of the ACM*, 27(7):695–702, 1984.

[17] Philippe Flajolet and Andrew Odlyzko. Exploring binary trees and other simple trees. In *21st Annual Symposium on Foundations of Computer Science*, pages 207–216, Syracuse, New York, 1980. IEEE.

[18] Leo J. Guibas and Robert Sedgewick. A diochromatic framework for balanced trees. In *Proceedings of the 19th Annual Symposium on Foundations of Computer Science*, pages 8–21. IEEE Computer Society, 1978.

[19] Sabine Hanke. *Rot Schwarz Bäume in Mehrbenutzerumgebungen*. PhD thesis, University of Freiburg, 2000.

[20] Sabine Hanke and Eljas Soisalon-Soininen. Group updates for red-black trees. In *Proceedings of the 4th Italian Conference on Algorithms and Complexity*, Lecture Notes in Computer Science, pages 253–262. Springer-Verlag, 2000.

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

[22] Raj Jain. *The art of computer systems performance analysis : techniques for experimental design, measurement, simulation, and modeling*. John Wiley and Sons, Inc., New York, 1991.

[23] T. Johnson, P. Krishna, and A. Colbrook. Distributed indices for accessing distributed data. In *IEEE Symposium on Mass Storage Systems (MSS ’93)*, pages 199–208. IEEE Computer Society Press, 1993.

[24] J. L. W. Kessels. On-the-fly optimization of data structures. *Communications of the ACM*, 26:895–901, 1983.

[25] Donald E. Knuth. *Fundamental Algorithms*, volume 1 of *The Art of Computer Programming*. Addison-Wesley, Reading, MA, USA, third edition, 1997.

[26] Donald E. Knuth. *Sorting and Searching*, volume 3 of *The Art of Computer Programming*. Addison-Wesley, Reading, MA, USA, second edition, 1998.

[27] W. I. Landauer. The balanced tree and its utilization in information retrieval. *IEEE Transactions on Electronic Computers*, 12(6), 1963.

[28] Sheau-Dong Lang and James R. Driscoll. A unified analysis of batched searching of sequential and tree- structured files. *ACM Transactions on Database Systems*, 14(4), 1989.

[29] Kim S. Larsen. AVL trees with relaxed balance. In *Eighth International Parallel Processing Symposium*, pages 888–893, 1994.

[30] Kim S. Larsen. Amortized constant relaxed rebalancing using standard rotations. *Acta Informatica*, 35:859–874, 1998.

[31] Kim S. Larsen and R. Fagerberg. B-trees with relaxed balance. In *9th International Parallel Processing Symposium*, pages 196–202, 1995.

[32] Kim S. Larsen, Thomas Ottmann, and Eljas Soisalon-Soininen. Relaxed balance for search trees with local rebalancing. Technical Report PP-1997-14, Department of Mathematics and Computer Science, Odense University, 1997.

[33] Lauri Malmi. *On Updating and Balancing Relaxed Balanced Search Trees in Main Memory*. PhD thesis, Helsinki University of Technology, 1997.

[34] Lauri Malmi and Eljas Soisalon-Soininen. Group updates for relaxed height-balanced trees. In *Proceedings of the Eighteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Dystems*, pages 358–367, 1999.

[35] Lauri Malmi, Eljas Soisalon-Soininen, and Peter Widmayer. Group updates for search trees, 2000.

[36] Kurt Mehlhorn. A partial analysis of height-balanced trees under random insertions and deletions. *SIAM Journal on Computing*, 11(4):748–760, 1982.

[37] Kurt Mehlhorn and Athanasios Tsakalidis. An amortized analysis of insertions into AVL-trees. *SIAM Journal on Computing*, 15(1):22–33, 1986.

[38] Otto Nurmi and Eljas Soisalon-Soininen. Uncoupling updating and rebalancing in chromatic binary search trees. In *Proceedings of the Tenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 192–198, 1991.

[39] Otto Nurmi and Eljas Soisalon-Soininen. Chromatic binary search trees: A structure for concurrent rebalancing. *Acta Informatica*, 33:547–557, 1996.

[40] Otto Nurmi, Eljas Soisalon-Soininen, and Derick Wood. Concurrency control in database structures with relaxed balance. In *Proceedings of the Sixth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 170–176, 1987.

[41] Otto Nurmi, Eljas Soisalon-Soininen, and Derick Wood. Relaxed AVL-trees, main-memory databases and concurrency. *International Journal of Computer Mathematics*, 62:23–44, 1996.

[42] Esko Nuutila. *Efficient transitive closure computation in large digraphs*. PhD thesis, Helsinki University of Technology, 1995.

[43] T. Ottmann and D. Wood. $1-2$ brother trees or AVL trees revisited. *The Computer Journal*, 23(3):248–255, 1980.

[44] Kerttu Pollari-Malmi, Eljas Soisalon-Soininen, and Tatu Ylönen. Concurrency control in B-trees with batch updates. *IEEE Transactions on Knowledge and Data Engineering*, 8(6):975–984, 1996.

[45] Markku Rossi. Concurrent full text database. Master's thesis, Helsinki University of Technology, 1997.

[46] Ben Shneiderman and Victor Goodman. Batched searching of sequential and tree structured files. *ACM Transactions on Database Systems*, 1(3):268–275, 1976.

[47] Eljas Soisalon-Soininen and Peter Widmayer. *Relaxed Balancing in Search Trees*. Kluwer Academic Publishers, 1997.

[48] Eljas Soisalon-Soininen and Peter Widmayer. Concurrency and recovery in full-text indexing. In *String Processing and Information Retrieval Symposium*, International Workshop on Groupware, pages 192–198, 1999.

[49] J. Srivastava and C.V. Ramamoorthy. Efficient algorithms for maintenance of large database indexes. In *Fourth International Conference on Data Engineering*, pages 402–408, 1988.

[50] Q. F. Stout and B. L. Warren. Tree rebalancing in optimal time and space. *Communications of the ACM*, 29(9):902–908, 1986.

[51] Robert Endre Tarjan. Updating a balanced search tree in $O(1)$ rotations. *Information Processing Letters*, 16(5):253–257, 1983.

[52] Athanasios K. Tsakalidis. AVL-trees for localized search. *Information and Control*, 67(1–3):173–194, 1985.

[53] Andrew Chi-Chih Yao. On random 2-3 trees. *Acta Informatica*, 9(2):159–170, 1978.

[54] Tatu Ylönen. An algorithm for full-text indexing. Master's thesis, Helsinki University of Technology, 1992.

[55] Justin Zobel, Hugh E. Williams, and Sam Kimberley. Trends in retrieval system performance. In *Australasian Computer Science Conference*, pages 241–248, Canberra, Australia, 2000.