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

Interval Deletion in B-trees

Timo Lilja

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

Supervisor	Eljas Soisalon-Soininen
Instructor	Eljas Soisalon-Soininen

Espoo, June 1st, 2005

TEKNILLINEN KORKEAKOULU DIPLOMITYÖN TIIVISTELMÄ

Tekijä:	Timo Lilja		
Työn nimi:	B-puiden intervallipoistot		
Päivämäärä:	1. kesäkuuta 2005	Sivumäärä:	vi + 57
Osasto:	Tietotekniikan osasto		
Professuuri:	T-106 Ohjelmistojärjestelmät		
Työn valvoja:	professori Eljas Soisalon-Soininen		
Työn ohjaaja:	professori Eljas Soisalon-Soininen		

Intervallipoistossa B-puuhakurakenteesta poistetaan kaikki annetulle avainvälille kuuluvat avaimet. Intervallipoisto on erikoistapaus yleisestä ryhmäpoisto-operaatiosta, jossa joukko avaimia poistetaan puusta. Kun poistettavan ryhmän aliryhmä muodostaa jatkuvan avainvälialueen puussa, voidaan soveltaa intervallipoistoa. Relaatiotietokannoissa transaktion toteuttava prosessi tuottaa indeksiin kohdistuvia intervallipoisto-operaatioita tietyissä tilanteissa silloin, kun viite-eheys on säilytettävä.

Ylhäältä alaspäin etenevä tasapainotus on menetelmä, jossa puu tasapainotetaan päivitysoperaation etsintävaiheessa. Alunperin tasapainotukset suoritettiin vasta lisäys- tai poisto-operaation jälkeen erillisellä tasapainotusvaiheella. Rinnakkaisia järjestelmiä tarkasteltaessa ylhäältä alaspäin etenevä tasapainotus takaa sen, että solmuja tarvitsee lukita ainoastaan vakiomäärä, kun taas alhaalta ylöspäin etenevässä tasapainotuksessa lukkojen lukumäärä on pahimmassa tapauksessa suhteessa puun korkeuteen. Siispä ylhäältä alaspäin etenevä tasapainotus mahdollistaa prosessien tehokkaamman rinnakkaisen suorittamisen.

Tässä työssä luodaan katsaus B-, B^+-, $\mathrm{B}^{\mathrm{link}}$- ja (a, b)-puihin ja niissä sovellettaviin tasapainotusmenetelmiin sekä selvitetään nykyisin tunnettuja intervallipoisto-operaation toteutustapoja. Työssä esitellään yksityiskohtaisesti ylhäältä alaspäin etenevä yksivaiheinen intervallipoistoalgoritmi, joka suorittaa vakiomäärän tasapainotusoperaatioita tasoitetun vaativuuden mielessä.

Työssä esitellään myös kokeellinen osuus, jossa pyritään vertailemaan algoritmin suorituskykyä muihin lähestymistapoihin ja hakemaan kokeellista tukea esitetylle teorialle.

Työn uusi tulos on ylhäältä alaspäin tasapainottava yksivaiheinen intervallipoistoalgoritmi.

Avainsanat: tietorakenne, algoritmi, hakupuu, puu, ryhmäpäivitys, ryhmäpoisto, ryhmälisäys, intervallipoisto, avainvälipoisto

HELSINKI UNIVERSITY OF TECHNOLOGY

ABSTRACT OF MASTER'S THESIS

Author:	Timo Lilja	
Title of thesis:	Interval Deletion in B-trees	
Date:	1st June 2005	Pages: vi + 57
Department:	Department of Computer Science	
Professorship:	T-106 Software Technology	
Supervisor:	Professor Eljas Soisalon-Soininen	
Instructor:	Professor Eljas Soisalon-Soininen	

In interval deletion a range of keys is to be removed from a B-tree index structure. Interval deletion is a special case of generic bulk deletion where there is a group of keys to be removed from the tree. When a subgroup of this group forms a continuous region in the tree an interval deletion can be applied. In relational databases where key-integrity constraints are to be maintained it is common, in certain situations, for the processes implementing these constrains to create interval deletion operations for secondary B-tree indices.

Top-down is a strategy where the balancing operations are done during the search phase of an update procedure. The original approach was to first perform the insertion or deletion operation and only after that the balancing within a separate balancing phase. This strategy is called bottom-up balancing. When considering a concurrent environment where a set of operations are simultaneously run, top-down balancing guarantees that only a constant number of nodes needs to be locked, whereas in bottom-up balancing the number of locks needed, in the worst case, is in relation to the height of the tree. Thus top-down provides higher concurrency for the processes.

In this thesis we give a brief overview of B-, B^+, B^{link}- and (a, b) trees and their balancing strategies and survey current approaches to top-down interval deletion. Next, we describe a top-down single-pass interval deletion algorithm thoroughly and prove that the algorithm performs a constant amount of balancing operations when a sequence of operations is considered in the amortised sense.

Experiments where the new algorithm is compared to another approach were performed in order to study and validate the theoretical claims stated previously.

A novel result in this thesis is the top-down single-pass interval deletion algorithm.

Keywords: data structures, algorithms, search trees, tree, group update, bulk update, group deletion, bulk deletion, group insertion, bulk insertion interval deletion, range deletion

Acknowledgements

Writing this thesis has been a long but interesting task. It would never have happened without the help of my colleagues and friends in the Laboratory of Information Processing Science in the Department of Computer Science at the Helsinki University of Technology.

I am especially grateful to my instructor and supervisor, professor Eljas Soisalon-Soininen for providing me this interesting problem, guidance and an opportunity to work on it. I also like to thank Riku Saikkonen, Kerttu Pollari-Malmi and Kenneth Oksanen for providing me assistance, discussions and peer review.

In addition I wish to thank my parents, all my friends and my upper secondary school math teacher, Jyrki Rauhalinna, without whom I would never have got interested in Mathematics and Computer Science. Finally, I thank Hal Abelson and Gerald Jay Sussman for providing the best Computer Science introduction book [1] that I have so far encountered.

Contents

List of Algorithms

Chapter 1

Introduction

B-trees, originally introduced by Bayer and McCreight in [5], and their variants are common search structures. They are the index structure of the choice in data base management systems (DBMS) where the qualities of the B-trees make them especially suitable for systems where there are two-level memory hierarchies like main memory and secondary disk storage. B-trees have found their place even in the two-level hierarchy of CPU cache and the main memory [39]. In other words, B-trees have gained popularity as an applicable index structure in cases where previously only AVL trees or red-black trees had been considered. Normally, search tree structures implement the *dictionary* abstract data type (ADT). That is, a given key can be either *searched*, *inserted* into or *removed* from the tree.

Nowadays, the need for bulk updating search structures is getting more common. Data mining and other applications generally require that the search structures can handle the classic dictionary operations for large data-sets. These kind of operations where a group of insertion or deletion operations are combined into a single operation are commonly called *bulk updates* or *bulk insertions* and *bulk deletes*. For example, a USENET[22] news server might do a bulk update into its internal search structures when a batch of new news is fed into it from another server. Another example where bulk updates are common is the case where updates are deferred and collected into a separate structure. In certain time intervals these pending updates are brought into the main structure [38].

Bulk operations have been subject to large study in the literature. Bulk insertion operations for B-trees were studied in [38, 44] and for the red-black trees [12] in [14]. AVL trees [2] were considered in [34]. Other papers that consider bulk updating in various data structures include [3, 4, 9, 11, 20, 26, 28].

The B-trees are *balanced*. That is, each path from the root node into the leaves is of same length and contains certain number of keys. In order to maintain these balancing conditions, special care has to be taken when implementing the insertion and deletion operations. Basically there are three alternatives: *bottom-up*, *top-down* [12] and a separate balancing process [40]. In bottom-up, the leaf node where the key is to be inserted or removed is first looked, then the operation is performed and finally the tree is brought back to balance with a separate balancing procedure. In top-down balancing the balancing is intermixed into the search phase of the update operation. In a separate balancing process the tree is not balanced at all during the update operation. The balancing is done only afterwards, e.g., at a time when the load of the system is in its minimum.

Interval deletion is a special case of bulk or group deletion. In interval deletion we have a given B-tree index structure and a key interval $[L, R]$. The goal is to remove all the keys that belong to interval $[L, R]$ from the tree so that the tree remains still a valid balanced B-tree. In this thesis we present a top-down balanced single-pass interval deletion algorithm. That is, the balancing phase is intermixed with the search phase and the algorithm does only one pass into the tree.

The structure of this thesis is that in Chapter 2 we give an overview and survey of the current B-tree algorithms without considering the bulk operations. In Chapter 3 we define the interval deletion problem, survey the current state of the art and present our new algorithm and give some theoretical remarks. Chapter 4 describes the experiments that were carried in this work in order to gain some empirical data on the behaviour of the new algorithm. We give a summary of the work, conclusions and suggestions for further work in Chapter 5.

Chapter 2

B-trees

2.1 Bottom-up B-trees

We define the directed tree T to be a *B-tree of order* m[1], where $m \geq 3$ is a natural number, if the tree T the is either empty or has the following properties:

1. Given integer $h \geq 0$, each path from the root to leaf has the same length h. This is the *height* of the tree.

2. Each node, except the root and the leaves, has at least $\lceil m/2 \rceil$ children. The root is either a leaf having no children, or has at least two children.

3. Each node, except the leaves, has at most m children.

A B-tree node N has the following structure

$$[p_0, k_0, \alpha_0, p_1, k_1, \alpha_1, \ldots, p_{n-1}, k_{n-1}, \alpha_{n-1}, p_n],$$

where $\lceil m/2 \rceil - 1 \leq n \leq m - 1$. The elements $k_1, k_2, \ldots, k_n$ are called the *keys*. A node that has n children, has $n-1$ keys. The keys are stored in increasing order, that is, $k_1 < k_2 < \cdots < k_{n-1}$. The elements $p_0, p_1, \ldots, p_n$ are pointers to child nodes of N. On leaf nodes, where there are no pointers to child nodes, they are given the value NIL. The α_i is the data in the index element (k_i, α_i). The triple (k_i, α_i, p_i) or omitting α_i is called an *entry*.

Each pointer p_i points to a subtree T_{p_i} with height one less than the node N containing the pointer p_i. Denote by $K(p_i)$ the set of keys in subtree pointed by p_i. One of the following properties shall hold:

1. $\forall k \in K(p_0) : k \leq k_0$

2. $\forall k \in K(p_i) : k_i < k \leq k_{i+1}$, $i = 0, 1, \ldots, n-1$

3. $\forall k \in K(p_\ell) : k_{n-1} < k$

The conditions above specify that the node p_i contains only the key range between the possible two adjacent keys and thus can be used for searching the

[1]Bayer and McCreight considered only the case where the m, the order of the tree, was odd, here we use the definition given by Knuth in [24, pp. 473–480].

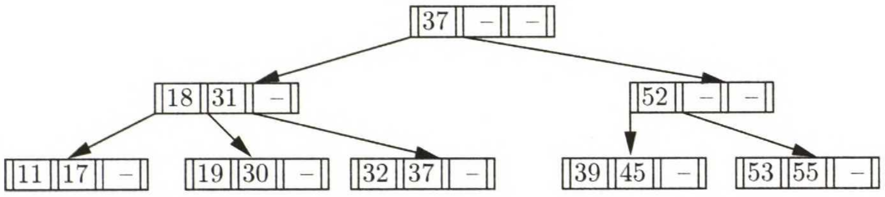

Figure 2.1: A B-tree of order $m = 4$ with the data α_i omitted.

keys. Usually B-trees are constructed so that the order of the tree or *branching factor* m is quite large. Large branching factor implies that B-trees grow height very slowly, which makes them a very suitable data structure for hierarchically organised memories: first levels of the tree can be kept in first level fast memory whereas the leaves can be stored in second level slow memory. Bayer and McCreight presented B-trees in 1971 [5]. See Figure 2.1 for an illustration of a B-tree.

B-tree nodes are commonly organised as a fixed size linear array in computer memory. There is usually some extra book keeping information associated in addition to the information presented above. For example, the number of elements n and whether the node is a leaf can be useful.

B-trees implement the *dictionary* abstract data-type. That is, the operations *search*, *insert* and *delete* are defined. We next define the operations carefully and give pseudo code implementations to them. The pseudo code uses some programming language like constructs instead of the more mathematical notation used previously: when referring to the parts of a node structure: The ith key k_i in a node is denoted node.key[i] and the pointer p_i is denoted node.p[i].

We assume that the tree cannot contain duplicate keys. It is not hard to modify the algorithms to support duplicates but it would cause unnecessary complexity to algorithms and it was considered irrelevant for the presentation given in this thesis.

Search

The pseudo code for B-tree search operation is presented in Algorithm 1. Searching a key k starts from the root node and proceeds towards the leaves. In every node, the correct key is searched with the procedure NODE-SEARCH[2]. In plain B-trees all nodes contain data, so we have to always check whether the key k is found and return the data α, if so (lines 4–5). Otherwise we descend to the corresponding child node p_i, whose key range contains the key k (line 7). Once the search ends up to a node with NIL value, i.e., we have reached and processed the leaf, we know that the key is not in the tree.

If the branching factor m of the tree is large, NODE-SEARCH procedure performs a search in the sorted array formed by the keys of the node. This can be implemented with *linear search* or if the branching factor m of the tree is large, with a *binary search.*

[2]We omit the definition of NODE-SEARCH for B-trees. In Section 2.2 we describe them thoroughly.

Algorithm 1 Internal bottom-up B-tree: search

```
B-TREE-SEARCH(root : node, k : key):
 1: node ← root
 2: while node ≠ NIL do
 3:    i ← NODE-SEARCH(node, k)
 4:    if k = node.k[i] then
 5:       return $\alpha_i$
 6:    else
 7:       node ← node.p[i]
 8:    end if
 9: end while
10: return NIL
```

Insert

Code for the insertion operation is given in Algorithm 2. The algorithm inserts the entry (k, α) to the B-tree index structure and balances the tree if needed. The balancing is described later.

If the tree is empty, a new node is created with the procedure NODE-ALLOC[3], the entry is inserted to that node and a pointer to this new root is returned (lines 1–5). Otherwise the tree has at least one node. We perform a search to that node (line 8) with NODE-SEARCH in order to check if the key is already in the tree and give an error message if needed (lines 9–10). If the key is not in the tree, we store the index entry i and a pointer the current node into the stack S and descend to the child node, whose key-range contains the key k (lines 12–13).

The loop terminates when node= NIL, that is, we have reached a leaf node. In B-trees, insertion occurs always at the leaf level, thus, we have found the correct spot where to insert the entry. First we pop the leaf out from the stack S (line 16) and check whether it is full (line 17). If the leaf is full, we have to *split* it before the entry can be inserted (lines 18–21).

In a split half of the entries are moved to a newly allocated node and a pointer to this new node and a key is inserted into the parent node. The splitting done is by calling SPLIT-CHILD. If the parent node is also full, it has to be split before the new node can be inserted as its child. This can propagate all the way up to the root (lines 24–29). If the root node is full, it has to be split like any other node, but since there is no parent a new root must be allocated and the old root and the split half of the root are inserted as this new root's children (lines 30–35).

This kind of balancing scheme, where the process first traverses the tree from root to leaf and stores the path into a stack and after the insertion backs up and balances the tree is called *bottom-up balancing.* We omit the definition of bottom-up balanced SPLIT-CHILD but later we define a variant from which it is easy to deduct the bottom-up SPLIT-CHILD implementation.

[3]Procedure NODE-ALLOC obtains a memory area and initialises the node structures (e.g., resets node counters). We omit the definition of NODE-ALLOC since it is highly implementation dependant.

Algorithm 2 Internal bottom-up B-tree: insert

```
B-TREE-INSERT(root : node, k : key, $\alpha$ : data):
if root = NIL then
    root $\leftarrow$ NODE-ALLOC()
    NODE-INSERT(root, k, $\alpha$)
    return root
end if
node $\leftarrow$ root
while node $\neq$ NIL do
    i $\leftarrow$ NODE-SEARCH(node, k)
    if k = node.k[i] then
        error "duplicate key"
    else
        push(S, node, i)
        node $\leftarrow$ node.p[i]
    end if
end while
(node, i) $\leftarrow$ pop(S)
if node.n = $m - 1$ then
    (parent, i) $\leftarrow$ pop(S)
    SPLIT-CHILD(parent, i)
    i $\leftarrow$ NODE-SEARCH(parent, k)
    node $\leftarrow$ parent.p[i]
end if
NODE-INSERT(node, k, $\alpha$)
node $\leftarrow$ parent.p[i]
while S $\neq$ NIL and node.n = $m$ do
    (parent, i) $\leftarrow$ pop(S)
    SPLIT-CHILD(parent, i)
    node $\leftarrow$ parent
end while
if root.n = $m$ then
    newroot $\leftarrow$ NODE-ALLOC()
    newroot.p[0] $\leftarrow$ oldroot
    SPLIT-CHILD(newroot, 0)
    root $\leftarrow$ newroot
end if
return root
```

Delete

The most complex operation in B-trees is the delete. The pseudo code is presented in Algorithm 3. The goal of the operation is to remove the given key k and its data α from the tree. The algorithm starts with searching the node that contains the key k storing all nodes in the path to this node into a stack (lines 2–9). If we end up into a NIL node, the key is not in the tree and an error is signalled (lines 11–12).

Otherwise we have found the node where the key k is. If this node is not a leaf, we need to get the smallest key larger than k from the subtree rooted at the same position where the key k lies. This is done with the FIND-MIN procedure which traverses the subtree and obtains the smallest key. This smallest key is always in the leftmost leaf of the subtree rooted in the position of k. Once we have obtained the smallest key, call it k', we replace k with k' and recursively perform B-TREE-DELETE in order to remove k' from the tree (lines 14–16).

If the node where the key k is located is leaf, we remove the key with NODE-DELETE procedure, which removes the key and data α, updates the node count (line 18).

After removing the key k the node may contain fewer than $\lceil m/2 \rceil$ entries and needs to be *compressed* in order to keep the B-tree properties valid. Here the path stored in the stack S comes in need. We pop the parent of the node and compress it with COMPRESS-CHILD (lines 19–23).

Compressing a node is somewhat more complex than splitting. Compression can be done by either *sharing* or *fusing*.[4] In sharing, the entries in the node are rearranged with an adjacent sibling of the node so that both have at least the minimum number of entries. In fusing, two adjacent nodes are merged together as single node. Fusing is the opposite of the splitting operation done during insertion. The choice between doing fusing or sharing depends on the number of elements in the neighbouring nodes. If both the node and its neighbour have exactly m elements, they are fused together. Otherwise the elements are redistributed evenly. The procedure COMPRESS-CHILD decides whether to fuse or share.

Finally, compressing the tree may cause the root to lose one of its children. If after the compression the root has only one child, the entire root can be removed and this only child is the new root. Thus, B-trees grow and lose their height only from root (lines 25–29).

We define COMPRESS-CHILD and the share and fuse operations in next section when we describe a more common variant of B-trees. The algorithms omitted here are almost identical to those presented in the next section. Since these original Bayer and McCreight B-trees are not used very much it was considered to be more useful to present the exact pseudo code when describing a more often used variant.

[4]The terms *fusing* and *sharing* are that of Mehlhorn et. al [35]. Bayer and McCreight use the terms *catenation* and *underflow* [5]. Nowadays, more common terms would *merging* and *redistribution*.

Algorithm 3 Internal bottom-up B-tree: delete

```
B-TREE-DELETE(root : node, k : key):
1: node ← root
2: while node ≠ NIL do
3:   i ← NODE-SEARCH(node, k)
4:   if k = node.key[i] then
5:     break
6:   else
7:     push(S, node, i)
8:     node ← node.p[i]
9:   end if
10: end while
11: if node = NIL then
12:   error "key not in the tree"
13: else if not node.leaf then
14:   l ← FIND-MIN(node.p[i])
15:   node.k[i] ← l
16:   return B-TREE-DELETE(l, node.p[i])
17: else
18:   NODE-DELETE(node, k)
19:   while S ≠ NIL and node.n = $\lceil m/2 \rceil$ do
20:     (parent, i) ← pop(S)
21:     COMPRESS-CHILD(parent, i)
22:     node ← parent
23:   end while
24: end if
25: if not root.leaf and root.n = 1 then
26:   oldroot ← root
27:   root ← root.p[0]
28:   NODE-FREE(oldroot)
29: end if
30: return root
```

2.2 Top-down B$^+$-trees

One of the most popular B-tree variants is the B$^+$-tree. In a B$^+$-tree all data and keys lie in the leaves. The internal node contain only *routers*, which direct the search towards the correct leaf node. The leaf nodes are linked together from left to right to provide easy sequential accessing of the indexed keys. Trees, where the data is located on every node are sometimes called *internal search trees* whereas trees where only the leaves contain the data are called *external search trees*. According to Maelbrancke and Olivié [33] Knuth was the first to mention the B$^+$-tree variant in [24] even though he did not give a precise definition to it.

Let us consider the implications of keys residing only in the leaves. Searching will always be processed from root to a leaf node, unlike in an ordinary B-tree, where encountering the key first time is an indication of the fact that key was found. If a search was unsuccessful, the search will terminate into a leaf, anyway. In ordinary B-trees failing search will always terminate in a leaf node, whereas successful search, may terminate in any level of the tree. Thus making searching a bit more expensive in B$^+$-trees. This cost can be neglected, if one considers the benefits of making the keys reside only in the leaves: An efficient implementation of sequential accessing would be impossible in an ordinary B-tree.

The B$^+$-tree node structure is similar to that of B-trees of Section 2.1. An internal node with n children has the following structure

$$[p_0, k_0, p_1, k_1, \ldots, p_{n-1}, k_{n-1}, p_n].$$

Similarly, a leaf node has the form

$$[\alpha_0, k_0, \alpha_1, k_1, \ldots, \alpha_{n-1}, k_{n-1}, p_{next}].$$

An internal node has at least $\lfloor m/2 \rfloor$ and at most m child nodes. Likewise, there are at least $\lfloor m/2 \rfloor - 1$ keys and at most $m - 1$ keys in an internal node. Also, every node has a variable that indicates the number of children (internal node) or keys (leaf node) the node has. In addition to this, there is a flag, indicating whether the node is a leaf. In top-down B$^+$-trees, the branching factor must be $m \geq 4$, whereas in bottom-up trees $m \geq 3$ would suffice. We discuss this further in Section 2.3.

In a leaf node, the pointers $p_0, p_1, \ldots$ point to some external data. The last pointer p_{next} points to the next adjacent leaf. Thus, there is room only for $m - 1$ data pointers and keys in a leaf node, if we want to make the leaf and internal nodes the same size. In ordinary B-trees the child pointers p_0, p_1 of a leaf were undefined and wasted space. An illustration of a B$^+$-tree is presented in Figure 2.2.

B$^+$-trees implement the dictionary and ISAM (*Indexed Sequential Access Method*) abstract data types. That is, in addition to the normal search, insert and delete operations of dictionary, a *search-next* operation is defined. The search-next operation searches the next larger key for a given key k. In an ordinary B-tree this would require traversing the entire tree whereas in B$^+$-trees with next links in the leaves, one can easily scan the leaves (i.e., the data nodes) with a basic *linked-list* scanning algorithm.

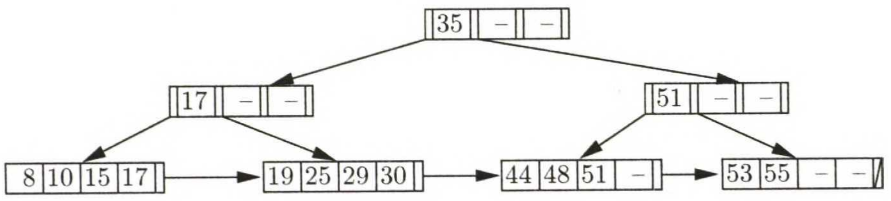

Figure 2.2: A B$^+$-tree of order $m = 4$. Internal nodes can have duplicates of the keys in the leaves' nodes. Leaves are linked together.

Top-down vs. Bottom-up Balancing

Now we are ready to discuss the implementation of the actual B$^+$-tree operations. In Section 2.1 we presented ordinary B-trees with bottom-up balancing, here we present B$^+$-trees with *top-down* balancing scheme. In top-down, the balancing is done in the search phase of the insert and delete operations. During an insert operation, whenever a node that has m children is encountered, it will be split immediately. In delete operations, if a node has exactly $\lceil m/2 \rceil$ elements, it will be either shared with a neighbouring node or fused if there are only $\lceil m/2 \rceil$ child elements in the neighbour too.

The benefits of top-down balancing compared to bottom-up come evident, when the tree is concurrently accessed by multiple processes. Ordinary bottom-up balanced algorithm has to lock the entire search path up to a node which with it can guarantee that no modifications will occur upper than that node. In worst case, this can mean that the root node has to be locked, thus no other process can access the tree at all. It is straightforward to implement top-down algorithms so that only constant amount of locks are needed during insertion and deletion operations.

Top-down balancing can do some extra work, making node splits or balancing when they are unnecessary: consider an insert operation that at first searches a path from root to leaf. Call this path $q_1, \ldots, q_h$. Let $q_1, q_2, \ldots, q_{i-1}$ and $q_{i+1}, \ldots, q_h$ be full nodes in the search path, let q_i be non-full node in level $i \leq h$. Now, bottom-up balancing will split only the nodes whose height $> i$ in the tree, whereas top-down will split every node except q_i. See also Figure 2.3.

Search

Algorithm 4 describes B$^+$-tree search. The idea is similar to that of ordinary B-tree search of Algorithm 1 in Section 2.1. But here, unlike in the B-tree search, we always traverse the tree to the leaf and only in the leaf we check whether the key is in the tree. The procedure returns the node and index i to the entry which contains the searched key. The node is needed so that the search-next-key operation can be effectively implemented. We leave the definition of the search-next-key procedure as an exercise for the reader.

Insert

The operation insert is presented in Algorithm 5. The insertion uses top-down balancing scheme. That is, it is always guaranteed that before the algorithm descends to a node, there will be room for one more entry.

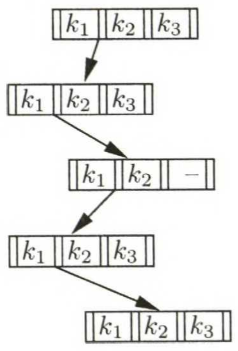

Figure 2.3: The case where top-down balancing does extra work. Only the tree lower nodes need to be stored to the stack during a bottom-up balancing since the balancing stops to the third level. In top-down, balancing is done conservatively and the entire path from the first to the last node except the level three node will be split. (The sibling nodes are omitted.)

Algorithm 4 External top-down B$^+$-tree: search

```
B+-TREE-SEARCH(root : node, k : key):
 1: parent ← root
 2: while not parent.leaf do
 3:    i ← NODE-SEARCH(node, k)
 4:    parent ← node.p[i]
 5: end while
 6: i ← NODE-SEARCH(node, k)
 7: if i ≠ NIL then
 8:    return (node, i)
 9: else
10:    return NIL
11: end if
```

If the tree is empty, we allocate a new node, insert the key to that node and return (lines 1–5). Otherwise, if the root is full, we allocate a new root, split the old root and put the split nodes as the children of the new root (lines 6–11). After this we start traversing the path from the root to the leaf (lines 13–21). If a node that is full is encountered, it will be split (lines 17–20). When the leaf is encountered, it will have enough room to fit the new node due to the top-down balancing. Finally, the node is inserted (line 23).

Algorithm 5 External top-down B$^+$-tree: insert

```
B+-TREE-INSERT(root : node, k : key, α : data):
 1: if root = NIL then
 2:    root ← NODE-ALLOC()
 3:    NODE-INSERT(root, k)
 4:    return root
 5: end if
 6: if root.n = $m$ then
 7:    newroot ← NODE-ALLOC()
 8:    newroot.p[0] ← root
 9:    SPLIT-CHILD(newroot, 0)
10:    root ← newroot
11: end if
12: node ← root
13: while not node.leaf do
14:    parent ← node
15:    i ← NODE-SEARCH(parent, k)
16:    node ← parent.p[i]
17:    if node.n = $m$ then
18:       SPLIT-CHILD(parent, i)
19:       node ← parent
20:    end if
21: end while
22: {node is leaf and is not full due to the top-down balancing}
23: NODE-INSERT(node, k)
24: return root
```

Split

Now we give the exact description of the top-down variant of the split operation. Algorithm 6 presents the relevant pseudo code. First, we allocate a new node (line 1), obtain a pointer to the full node (line 2), count the number of elements needed to move (line 3) and make room into the parent for a new node pointer (lines 4–9). Then, if the node is a leaf, we simply copy half of the keys and pointers to data into the new node (lines 11–14). If the node is an internal node, we copy half of the pointers (lines 16–18) and routers (lines 19–21). Finally we insert the new node and a router into the parent (lines 23–24), set up the node counters and other information (lines 25–28).

Note that whenever a leaf is split a *key* is copied upwards and becomes a *router* in the next level. That is, B$^+$-tree can have duplicates of keys as routers.

When an internal node is split, the routers do not get copied, which means that there is at most two instances of a key (the key and a router copy of it) in the tree at any time. See Figure 2.4 for an illustration of a split operation where the key is copied to upper level of the tree. In ordinary B-trees where the data is stored on every node there could not have been any duplicate keys since that would have destroyed the search tree properties of the structure.

Algorithm 6 B$^+$-tree: splitting a child

```
SPLIT-CHILD(parent : node, i : index):
 1: newnode ← NODE-ALLOC()
 2: fullnode ← parent.p[i]
 3: t ← $m/2$
 4: for j = parent.n − 1 downto i + 1 do
 5:     parent.p[j+1] ← parent.p[j]
 6: end for
 7: for j = parent.n − 2 downto i do
 8:     parent.key[j+1] ← parent.key[j]
 9: end for
10: if fullnode.leaf then
11:     for j = t + 1 to fullnode.n − 1 do
12:         newnode.key[j − t − 1] ← fullnode.key[j]
13:         newnode.p[j − t − 1] ← fullnode.p[j]
14:     end for
15: else
16:     for j = t + 1 to fullnode.n − 1 do
17:         newnode.p[j − t − 1] ← fullnode.p[j]
18:     end for
19:     for j = t + 1 to fullnode.n − 2 do
20:         newnode.key[j − t − 1] ← fullnode.key[j]
21:     end for
22: end if
23: parent.key[i] ← fullnode.key[i]
24: parent.p[i + 1] ← newnode
25: newnode.n ← $\lfloor m/2 \rfloor$
26: fullnode.n ← $\lceil m/2 \rceil$
27: newnode.leaf ← fullnode.leaf
28: parent.n ← parent.n + 1
```

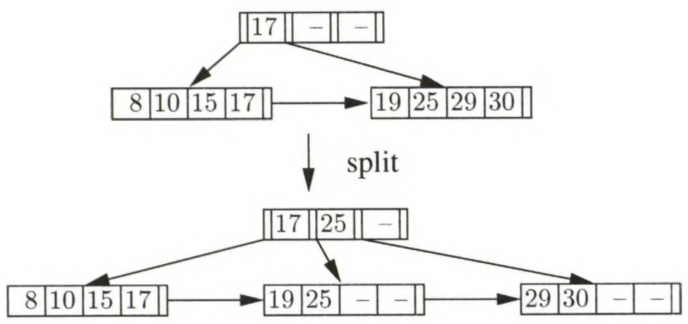

Figure 2.4: A B^+-tree of order $m = 4$ with a child split. The key 25 duplicates as a router into the parent node.

Delete

Deletion operation for B$^+$-tree is presented in Algorithm 7. The deletion algorithm is somewhat similar to that of insertion. The idea is to guarantee that when the node is descended it will have at least the $\lceil m/2 \rceil + 1$ entries. That is, the node can lose a child without becoming unbalanced.

At first, we check that the tree is not empty (lines 1–3). Then we traverse the path from the root to the leaf containing the key to be removed (lines 5–18). If we encounter a node that is half full, we compress it with COMPRESS-CHILD. That is, depending on the sizes of the neighbouring nodes, the node is either *shared* or *fused* (see the Algorithms 9 and 10). If the root node becomes under-full, that is, it has only one child, we remove the root node and set the only child as the new root. This is the only case where the height of the tree decreases (lines 13–17). After all nodes have been compressed, the key is removed from the leaf (line 20).

Jannink gives a good description of bottom-up B$^+$-tree deletion algorithm in [19]. Maelbrancke describes some optimisations in [33].

Algorithm 7 External top-down B$^+$-tree: delete

```
B+-TREE-DELETE(root : node, k : key):
1: if root = NIL then
2:     error "attempt to remove a key from an empty tree"
3: end if
4: parent ← root
5: while not parent.leaf do
6:     parent ← node
7:     i ← NODE-SEARCH(parent, k)
8:     node ← parent.p[i]
9:     if node.n = $\lfloor m/2 \rfloor$ then
10:        COMPRESS-CHILD(parent, i)
11:        node ← parent
12:    end if
13:    if not root.leaf and root.n = 1 then
14:        oldroot ← root
15:        root ← root.p[0]
16:        NODE-FREE(oldroot)
17:    end if
18: end while
19: {node is now leaf and can loose a child without becoming unbalanced}
20: NODE-DELETE(node, k)
21: return root
```

Compression

The procedure COMPRESS-CHILD (Algorithm 8) decides whether an under-full node will be shared or fused. In B$^+$-trees, the decision is simple: if the sum of the under-full node and its neighbour is exactly the size of the node, the nodes are merged together. If, on the other hand, the sum is greater than the node size, the contents of the nodes are shared. The procedure takes the parent node's pointer and index to the under-full child as its arguments and performs the compression operation by calling either the procedure FUSE or SHARE.

First, we obtain a pointer to the under-full node (line 1) and check whether the child has a right neighbour (line 2), if so, we set the procedure for sharing or fusing with it (line 3). Otherwise, the node is the rightmost child of the parent and it has to be compressed with a left neighbour (line 5). Finally, we obtain a pointer to the neighbour and FUSE or SHARE the elements depending on the number of elements in the nodes.

Algorithm 8 B$^+$-tree: compressing a child

```
COMPRESS-CHILD(parent : node, i : index):
 1: child ← parent.p[i]
 2: if i < parent.n − 1 then
 3:    j ← i + 1
 4: else
 5:    j ← i − 1
 6: end if
 7: neighbour ← parent.p[j]
 8: if child.n + neighbour.n < $m$ then
 9:    FUSE(parent, i, j)
10: else
11:    SHARE(parent, i, j)
12: end if
```

Fusing

In procedure FUSE (Algorithm 9) the contents of the under-full node is fused or merged together with a neighbour. In this implementation fusing is always done towards left. That is, if we request a fusing towards right it is first flipped by calling tail-recursively the procedure FUSE (line 22).

First, pointers for the under-full node and its neighbour are allocated (lines 1–2). Then we check that we are fusing into the correct direction (line 3). If the node is a leaf we copy keys and pointers from the neighbour into the node (line 6–9) and update the parent's router accordingly (line 10). If the node is internal, we move the routers from the neighbour into the node (line 12–14), move the key from parent into the node (line 15) and move the keys from the neighbour (lines 16–18).

At the end we shift the parent's routers and pointers one step left (lines 24–29) so that the neighbouring node gets removed. Then the node counters are updated (lines 30–31).

Note that we move the router from parent to the child (line 15) only if the fused node is internal. That is, the router from the parent is not moved downwards if the fused node is leaf. This is naturally due to the fact that there might be a router whose corresponding key has been removed already.

Sharing

The main procedure SHARE is given in Algorithm 10. It is merely a dispatcher that calls the appropriate sharing function SHARE-LEFT or SHARE-RIGHT de-

Algorithm 9 B$^+$-tree: fusing

```
FUSE(parent : node, i : index, j : index):
 1: empty ← parent.p[i]
 2: neighbour ← parent.p[j]
 3: if i < j then
 4:    {move elements from right to left}
 5:    if empty.leaf then
 6:       for k = 0 to neighbour.n − 1 do
 7:          empty.p[empty.n + k] ← neighbour.p[k]
 8:          empty.key[empty.n + k] ← neighbour.key[k]
 9:       end for
10:       parent.key[i] ← empty.key[empty.n + neighbour.n − 1]
11:    else
12:       for k = 0 to neighbour.n − 1 do
13:          empty.p[empty.n + k] ← neighbour.p[k]
14:       end for
15:       empty.key[empty.n] ← parent.key[i]
16:       for k = 0 to neighbour.n − 2 do
17:          empty.key[empty.n + k] ← neighbour.key[k]
18:       end for
19:    end if
20: else
21:    {move elements from left to right}
22:    return FUSE(parent, j, i)
23: end if
24: for k = j to n − 1 do
25:    parent.p[k] ← parent.p[k + 1]
26: end for
27: for k = j to n − 2 do
28:    parent.key[k] ← parent.key[k + 1]
29: end for
30: parent.n ← parent.n − 1
31: empty.n ← empty.n + neighbour.n
32: NODE-FREE(neighbour)
```

pending on whether the neighbouring node of the empty node is on the left or right side of the under-full node. Also, the main procedure sets the node counters after the actual sharing has been completed.

The procedure SHARE-LEFT (Algorithm 11) implements the actual sharing. First, obtain the pointers (lines 1–2). Variable t indicates the number of elements that are to be moved from the neighbouring node into the under-full node. In B^+-trees this variable is always t $= m/2$ where m is the branching factor. If the under-full node is leaf, we first copy t keys and pointers from the right neighbour (lines 5–8) and move the elements in the neighbour t steps left (lines 9–12). After this, the parent's router is set accordingly (line 13). If the under-full node is not leaf, the procedure is somewhat similar: first copy parent's router into the under-full node (line 15). Move t keys from the neighbour into the node (lines 16–18) move a router from the neighbour into the parent (line 19) and shift the neighbour elements t steps left (lines 20–25).

We leave the definition of SHARE-RIGHT as an exercise for the reader. It's analogous to SHARE-LEFT except that the order with which to copy and shift the elements from a neighbouring node into the under-full somewhat different. Due to this fact it's not possibly to just flip the i and j and call SHARE-LEFT as was done in the implementation of fusing in procedure FUSE.

The implementation given here is not very realistic: in a real implementation some kind of a block copying method[5] would be used instead of the looping constructs used here.

Algorithm 10 B^+-tree: sharing: main procedure

```
SHARE(parent : node, i : index, j : index):
1: empty ← parent.p[i]
2: neighbour ← parent.p[j]
3: if i < j then
4:     SHARE-LEFT(parent, i, j)
5: else
6:     SHARE-RIGHT(parent, i, j)
7: end if
8: neighbour.n ← neighbour.n − t
9: empty.n ← empty.n + t
```

Node Operations

There are some auxiliary node operations needed and they are given in Algorithm 12. They operate on a single node and abstract away some of the repeated tasks that need to be done in implementing the actual B^+-tree operations. These operations are described here because they are identical to all variants described in this chapter.

The procedure NODE-SEARCH performs a linear search into the array of keys of the node and returns an index to the element that contains the key. Naturally, in a real implementation where the size of the node can be quite large *binary search* would be much better approach. The procedures NODE-INSERT and NODE-DELETE insert and remove an element from the node.

[5] e.g., the `memcpy()` function of the operation system and/or the C library.

Algorithm 11 B$^+$-tree: sharing towards left

```
SHARE-LEFT(parent : node, i : index, j : index):
1: empty ← parent.p[i]
2: neighbour ← parent.p[j]
3: t ← $m/2$
4: if empty.leaf then
5:    for k = 0 to t − 1 do
6:       empty.key[empty.n + k] ← neighbour.key[i]
7:       empty.p[empty.n + k] ← neighbour.p[i]
8:    end for
9:    for k = t to neighbour.n − 1 do
10:      neighbour.key[i − t] ← neighbour.key[i]
11:      neighbour.p[i − t] ← neighbour.p[i]
12:   end for
13:   parent.key[i] ← empty.key[empty.n + t − 1]
14: else
15:   empty.key[empty.n − 1] ← parent.key[i]
16:   for k = 0 to t − 1 do
17:      empty.key[empty.n + i] ← neighbour.key[i]
18:   end for
19:   parent.key[i] ← neighbour.key[t − 1]
20:   for k = t to neighbour.n − 1 do
21:      neighbour.p[i − t] ← neighbour.p[i]
22:   end for
23:   for k = t to neighbour.n − 2 do
24:      neighbour.key[i − t] ← neighbour.key[i]
25:   end for
26: end if
```

Algorithm 12 Auxiliary node operations

```
NODE-SEARCH(node : node, k : key):
 1: i ← 0
 2: if node.leaf then
 3:   while i < node.n and node.key[i] < k do
 4:     i ← i + 1
 5:   end while
 6:   if i < node.n and node.key[i] = k then
 7:     return i
 8:   else
 9:     return NIL
10:   end if
11: else
12:   while i < node.n − 1 and node.key[i] < k do
13:     i ← i + 1
14:   end while
15:   return i
16: end if

NODE-INSERT(node : node, k : key, d : data):
 1: i ← NODE-SEARCH(node, k)
 2: if i ≠ NIL then
 3:   error "duplicate key"
 4: else
 5:   for j = node.n − 1 downto i do
 6:     node.key[j+1] ← node.key[j]
 7:     node.p[j+1] ← node.p[j]
 8:   end for
 9:   node.key[i] ← k
10:   node.p[i] ← d
11:   node.n ← node.n + 1
12: end if

NODE-DELETE(node : node, k : key):
 1: i ← NODE-SEARCH(node,k)
 2: if i ≠ NIL then
 3:   error "key not in tree"
 4: else
 5:   for j = i to node.n − 1 do
 6:     node.key[j] ← node.key[j+1]
 7:     node.p[j] ← node.p[j+1]
 8:   end for
 9:   node.n ← node.n + 1
10: end if
```

2.3 Top-down (a, b) trees

The (a, b) trees or weak B-trees trees are a generalisation of the B$^+$-trees. They were originally described in [17] by Huddleston and Mehlhorn. In (a, b) trees each node contains at least a elements and at most b elements. Further, $a \geq 2$ and $b \geq 2a - 1$ [17, 35]. The algorithms defined in Mehlhorn's and Huddleston's papers use the bottom-up balancing scheme. Here, we present (a, b) tree algorithms that use the top-down scheme. Mehlhorn considered top-down in [35, pp. 212] but left the implementation as an exercise. In top-down (a, b) trees $b \geq 2a$, because in the case $b = 2a - 1$, splitting a full node would yield a node who has less than a elements and thus violates the balancing condition.

The (a, b) trees as we define them here are external top-down balanced search trees similar to the B$^+$-trees of Section 2.2. Each internal node has at least a and at most b children plus a variable to indicate the exact number of children and a flag telling whether the node is a leaf. The structure for internal nodes:

$$[p_0, k_0, p_1, k_1, \ldots, p_{a-1}, k_{a-1}, \ldots p_n].$$

And like in B$^+$-trees, the leaves have the structure:

$$[\alpha_0, k_0, \alpha_1, k_1, \ldots, \alpha_{n-1}, k_{n-1}, p_{next}].$$

The (a, b) trees are somewhat more flexible than normal B$^+$-trees since the user has better control of the branching factor and thus the fan out rate can be made higher. Generally, (a, b) trees for small values of a and b are well suited for two-level memory structures[6]. Conversely, larger values are suitable for index structures stored in a hard disk.

Search

Searching in external top-down (a, b) trees does not differ from searching of similar B$^+$-trees. Thus, the procedure B$^+$-TREE-SEARCH in Algorithm 4 of Section 2.2 can be used as it is in order to perform a search in an (a, b) tree.

Insert

The algorithm for (a, b) tree insertion operation is analogous to B$^+$-TREE-INSERT procedure in Algorithm 5 of Section 2.2. The only difference is that the variable m has the value $m = b$. Because the changes needed to do in Algorithm 5 are trivial, we omit the exact pseudo code for (a, b) tree insertion.

Split operation is identical to that of SPLIT-CHILD in Algorithm 6. Here, too, the variable m will be replaced with the variable b.

Delete

The algorithm for deletion is almost identical to the procedure B$^+$-TREE-DELETE in Algorithm 7. Only the expression $\lfloor m/2 \rfloor$ is replaced with the variable a.

The most notable difference when compared to ordinary B$^+$-trees is in the compression procedures COMPRESS-CHILD, SHARE and FUSE. In (a, b) trees there is more latitude on how to do the compression. Let u and v be neighbouring

[6] E.g., processor cache and the main memory.

nodes in an (a, b) tree. Denote by $c(u)$ the number of elements (children or keys) the node u has. We assume that the node u has $c(u) = a$ elements and needs to be compressed. We define two variables: the sharing threshold t and the number of elements shifted during a share s. Now, the compression is done as follows: Nodes u and v are fused together if $c(u) + c(v) < t$. Otherwise the contents of the nodes u and v are shared. When sharing is performed, s elements are moved from node v into node u. The threshold must be between $0 \leq t \leq b - 2a$ and the shifting number in range $1 \leq s \leq t + 1$

The procedure COMPRESS-CHILD in Algorithm 13 is changed to reflect the (a, b) tree compression operation. Here we omit the exact algorithmic descriptions of the procedures SHARE and FUSE and merely point out that the changes needed to modify the Algorithms 9 and 10 are trivial.

Algorithm 13 (a, b) tree: compressing a child

```
COMPRESS-CHILD(parent : node, i : index):
 1: child ← parent.p[i]
 2: if i < parent.n − 1 then
 3:     j ← i + 1
 4: else
 5:     j ← i − 1
 6: end if
 7: neighbour ← parent.p[j]
 8: if child.n + neighbour.n < $t$ then
 9:     FUSE(parent, i, j)
10: else
11:     SHARE(parent, i, j, t)
12: end if
```

2.4 Complexity Results

Theorem 2.1. *Inserting (deleting) an element to a top-down (a, b) tree will cause at most $O(h)$ balancing operations, where h is the height of the tree.*

Proof. (The argument is similar to that of [35].) In Algorithm 5 the loop in lines 12–20 is repeated h times, where h is the height of the tree. Likewise, in Algorithm 7 the loop in lines 5–18 is repeated h times. There cannot be more balancing operations than there are iterations in the loops. Thus, the theorem follows. □

Theorem 2.2. *The height of an (a, b) tree is logarithmic with respect to the number of elements in the tree.*

Proof. This proof is analogous to Bayer's and McCreight's in [5] but here it is applied to (a, b) trees instead of internal B-trees. First we consider an (a, b) tree that is as empty as possible. That is, the root has two children and all other internal nodes have a children. We assume that the height of the root node is 1 and the height of the leaves is h.

$$N_{min} = 1 + 2(a^0 + a^1 + a^3 + \cdots + a^{h-2}) = 1 + \frac{2(a^{h-1} - 1)}{a - 1}$$

The (a, b) tree is as full as possible when all internal nodes have b children.

$$N_{max} = b^0 + b^1 + b^2 + \cdots + b^{h-1} = \frac{b^h - 1}{b - 1}$$

Hence, The number of nodes N of an (a, b) tree whose height is h is

$$1 + \frac{2(a^{h-1} - 1)}{a - 1} \leq N \leq \frac{b^h - 1}{b - 1}$$

Next, we consider the sum of the number of routers and keys in an (a, b) tree. The root node has at least 1 key or router and every other node has at least $a - 1$ keys or routers. There are at most $b - 1$ keys or routers in every node. That is,

$$K_{min} = 1 + (a - 1)\frac{2(a^{h-1} - 1)}{a - 1} = 2a^{h-1} - 1$$

and

$$K_{max} = (b - 1)\frac{b^h - 1}{b - 1} = b^h - 1.$$

We obtain

$$2a^{h-1} - 1 \leq K \leq b^h - 1$$

Solving h yields

$$\log_b (K + 1) \leq h \leq \log_a [(K + 1)/2] + 1$$

and the theorem follows. □

Theorem 2.3. *Assume that i insertions and d deletions are performed to an initially empty*[7] *(a, b) tree where $b \geq 2a + 3$. The rebalancing operations needed during an insertion or deletion in this sequence have amortised $O(1)$ complexity.*

Proof. We use Tarjan's potential function technique [43]. The amortised analysis presented here is somewhat similar to that of [28, 38] and is also published in [32]. The number of elements for the node u (children or pointers to external data) is $c(u)$. The potential of a non-root node is defined to be:

$$\Phi(u) = \begin{cases} 3, & c(u) = a \\ 1, & c(u) = a + 1 \\ 0, & a + 1 < c(u) < b - 1 \\ 2, & c(u) = b - 1 \\ 4, & c(u) = b \end{cases}$$

The potential of the root node is defined similarly but the variable a is replaced with the constant 2. The potential of the tree T is the sum of the potentials of all the nodes in the tree:

$$\Phi(T) = \sum_{u \in T} \Phi(u)$$

Insert

The actual insertion affects only the potential of the node where the element is inserted. The increase is at most 2 units.

Delete

The actual deletion affects only the potential of the node where the element is inserted. The increase is at most 2 units.

Split

Split changes the potential of the node to be split and its parent. Denote the node before a split q and its parent p. After the split, call the nodes q' and q'' and the parent p'. Before a split, we have $c(q) = b$ and due to the top-down balancing $a \leq c(p) \leq b - 1$. After the split, we have $c(q') = \lfloor b/2 \rfloor$ and $c(q'') = \lceil b/2 \rceil$. Because $b \geq 2a + 3$, we have $a + 1 \leq \lfloor b/2 \rfloor < b - 1$ and $a + 2 \leq \lceil b/2 \rceil < b - 1$. The inequality

$$\Phi(q) + \Phi(p) > \Phi(q') + \Phi(q'') + \Phi(p') \tag{2.1}$$

holds, since $\Phi(q) + \Phi(p) \geq 4$, because $\Phi(q) = 4$, $\Phi(q') + \Phi(q'') \leq 1$ and $\Phi(p') - \Phi(p) \leq 2$. Thus, the total decrease of the tree potential is at least 1 unit.

Compression

Compression occurs when a node q cannot lose a child. That is, when $c(q) = a$. Compression can be done either by *sharing* or *fusing*. Using q' to denote the sibling of q, we perform fusing when $c(q) + c(q') \leq 2a + 1$, otherwise the contents of the nodes are shared.

[7]See [18] on how to relax this constraint.

Fusing

In fusing, call the nodes before the operation q, r and p for the parent. After the operation, the combined node is called q' and the parent p'. Now, we show that the inequality

$$\Phi(q) + \Phi(r) + \Phi(p) > \Phi(q') + \Phi(p') \tag{2.2}$$

holds. Since $c(q) = a$, $a \leq c(r) \leq a+1$, it holds that $\Phi(q) + \Phi(r) \geq 4$. It is easy to see that $2a \leq c(q') \leq 2a+1 < b-1$, thus $\Phi(q') = 0$. The increase in parent's potential is $\Phi(p') - \Phi(p) \leq 2$.

Sharing

Sharing doesn't change the number of children in the parent node, thus the parent's potential does not change. Name the nodes before sharing q and r and after it q' and r'. We have $c(q) = a$ and $a+2 \leq c(r) \leq b$. We perform sharing so that the contents of the nodes q and r is distributed evenly. After sharing, we have $a+1 \leq c(q'),\ c(r') < b-1$. The inequality

$$\Phi(q) + \Phi(r) > \Phi(q') + \Phi(r') \tag{2.3}$$

is valid, since $\Phi(q) + \Phi(r) \geq 3$ and $\Phi(q') + \Phi(r') \leq 2$, because $a+1 \leq c(q'),\ c(r') < b-1$. Thus, the total decrease is at least 1 unit.

Conclusion

Now, consider a sequence i insertion and d deletion operations that are performed to an initially empty tree. Denote the potential of the tree after the jth operation by $\Phi(T_j)$. Clearly, $\Phi(T_j) \geq 0$, for all $1 \leq j \leq n = i+d$. Now, we have shown that inserting or deleting an element will increase the potential of the tree by at most 2 units and performing a balancing operation will decrease the potential of the tree at least 1 unit. The potential is always an integer. Thus, after n operations, the potential of the entire tree is bound by the following equation:

$$0 \leq \Phi(T_n) \leq 2n - r \cdot 1,$$

where r denotes the number of balancing operations performed. Now, solving r yields

$$r \leq 2n = 2(i+d),$$

which implies that, in the amortised sense, the number of balancing operations performed in a sequence of i insertions and d deletions is $O(i+d)$. □

In bottom-up (a,b) trees one can construct valid trees when $b \geq 2a-1$. The amortised constant bound is, however, valid only if $b \geq 2a$ [16, 17, 35]. In top-down trees $b \geq 2a$ and in cases $b = 2a$ and $b = 2a+1$ it is easy to construct a counter-example to contradict the amortised constant bound.

In the case $b = 2a+2$ I have been unable to find an operation sequence that breaks the amortised bound. According to Mehlhorn [35, p. 212] the amortised constant bound should be valid when $b \geq 2a+2$. However, I have been unable to find a proof of this claim nor have I found a counter-example. Thus it is an open

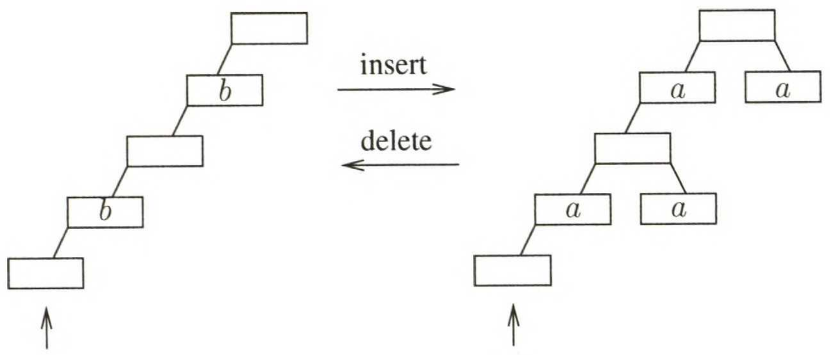

Figure 2.5: Part of an $(a, 2a)$ tree. Inserting a key to the leftmost leaf and deleting it afterwards causes balancing operation on every second level of the tree. The blank nodes are assumed to have enough children so that they can either loose or gain a child without further balancing. The node where the key is inserted in and deleted from is marked with a vertical arrow.

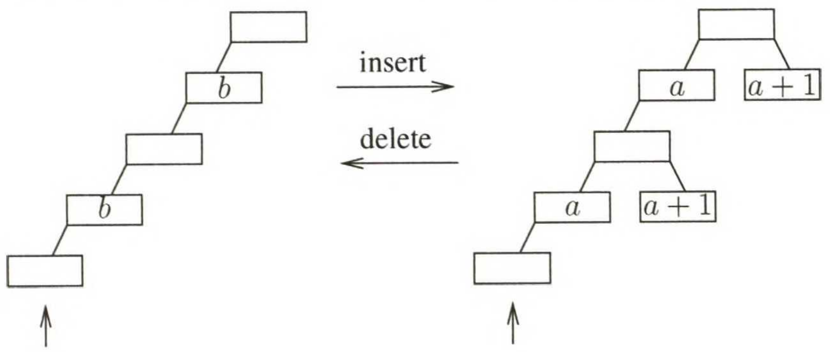

Figure 2.6: In the above fraction of an $(a, 2a+1)$ tree, balancing occurs on every second level of the tree during a sequence of insertion and deletion operations. Note that the figure and argument are analogous to the case of $(a, 2a)$ tree.

question whether the amortised bound holds for $b = 2a + 2$. (See Figures 2.5 and 2.6 for counter-examples in cases $2a \leq b \leq 2a + 1$.)

The above mentioned potential technique cannot be directly applied to the case $b = 2a + 2$, because it will contradict equations (2.1) and (2.2). If, in equation (2.1), $c(p) = b - 1$, then

$$\Phi(b - 1) > 2\Phi(a + 1).$$

Similarly, in equation (2.2), if $c(p) = a+1$ and $c(q) = a+1$. Substitution yields,

$$2\Phi(a + 1) > \Phi(b - 1)$$

i.e., a contradiction. Using sharing in the case where $c(r) = a + 1$ does not help, because share operation would move the one element from r to q and both sides of the inequality (2.3) would be equivalent. This result indicates that it is not possible to find a potential function that uses only the number of elements $c(u)$. I have also considered using the height of the node, the number of child

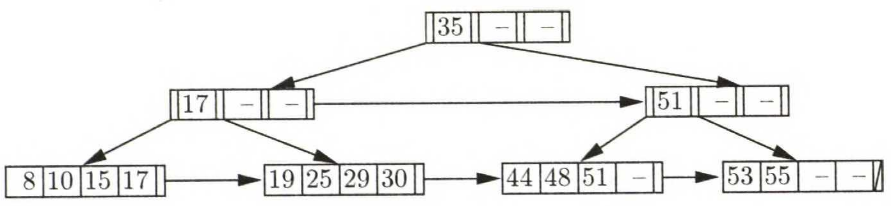

Figure 2.7: A B^{link}-tree. Each level has a pointer to its next to right sibling.

nodes and leaf nodes as a second argument to the potential function but none of them have given any positive results.

2.5 Other B-tree Variants

In addition the variants of the B-trees presented in the previous sections there are quite a few more. Many of these variations are applicable for certain problems with extra cost on some other aspects. Next, we survey few of the more common variants of the B-trees and their algorithms without going into details.

Lehman and Yao present the B^{link}-trees in [31] and gave the algorithms for searching and inserting elements to them. In B^{link}-trees there are links to the right sibling on each level of the tree. In B^+-trees these links exist only in the leaf level. See Figure 2.7. Later, Lanin and Shasha gave the algorithm for deletion in [27]. In a B^{link}-tree insertion or deletion operations need to lock only single node at a time, thus improving the concurrency considerably in comparison to previously discussed B-tree variants. The extra link pointer is used when the node does not contain the key it was thought to contain when the node was descended to. This can be the case when there are two concurrent processes, say P_1 and P_2, and a node N. Process P_2 holds a lock on N. Process P_1 obtains a pointer to the node. Process P_2 operates on the node and splits moving the key A is interested in to a new node. Process P_2 sets the link to point to this newly split node and releases the lock on N. Then P_1 obtains the lock and finds out that the node does not contain the key. Therefore it follows the next link to the node where the key is located. Inserting a router and a pointer to the newly split node into the parent is done later by a separate balancing process [40]. The exact description of the algorithms and proofs of their correctness are given in [27, 31].

Huddleston and Mehlhorn introduce *level-linked B-trees* in [17]. In level-linked B-trees every level has links to both siblings and the child nodes link to the parents. Level-linked trees are applicable when there is high areas of locality in the operations. Level-linked (a, b) trees can be used to implement many of the set operations in an optimal way. The balancing operations in both level-linked and normal (a, b) trees concentrate near the leaves. Thus, making the need to lock larger parts of the tree during a balancing phase uncommon.

One interesting alternative is so called *relaxed* B-trees. In relaxed B-trees the update operations need not uphold the balancing condition all the time. That is, for example, not all paths from root to the leaves are of the same size. Balancing is usually done in a separate process later when the system has less load. Larsen and others provide good discussion on this topic in [29, 30].

Knuth describes a variation called B*-trees where every node is at least 2/3 full [25, pp. 481–491] making the space utilisation much better than in ordinary B, B^+ or even (a, b) trees. As a matter of fact (a, b) trees can be much worse than ordinary B-trees when space utilisation is concerned: in B-trees nodes are always at least half full, in (a, b) trees this is only the case if $b = 2a$. If $b > 2a$ the utilisation is less than half. Also, the fan-out rate or branching factor does not need to be the same on each level of the tree [25]. Some authors—unfortunately—call B^+-trees defined in Section 2.2 B*-trees.

Normally the node structure is a linear array where the keys or routers and pointers are stored and a binary search is performed in order to find the proper key or router. Some variants, namely, RB-trees of [10] and AB-trees of [37] differ from the usual linear array approach. in RB-trees the keys are stored as red-black trees in the leaf nodes. In AB-trees AVL-trees are used. These non-conventional approaches are well suitable for efficient implementation of certain bulk update operations.

Comer [8] provides a good, though a bit dated, survey of the basic B-trees and its variants. Knuth provides rather concise and well-founded description of B-trees and its variants in [25]. Johnson and Shasha discuss various aspects relating to the performance of B-trees in [21].

Chapter 3

Interval Deletion

3.1 The Problem

The problem of *interval deletion* is the following: Given two keys L and R, $L < R$ and a B$^+$-tree T, the goal is to remove all the keys that lie in the interval $[L, R]$ in the tree and produce a valid balanced B$^+$-tree as a result. See Figure 3.1.

For example, consider executing the following SQL statement on a database with a primary B$^+$-tree index on attribute `C1`

```
DELETE
FROM T1
WHERE C1 > 100 AND C1 < 200
```

The above type of statement is frequently used according to [36]. Another example of a situation where an interval deletion might occur comes from data archiving [13]: Think of the relation `ORDERS(OrderNr,Customer,Date)` where a B$^+$-tree index I exists on the attribute `OrderNr`. If we want to remove the paid orders that are older than six months, we can implement the operation as a sequence of interval deletions on I, provided that there are only a few unpaid orders.

Database management systems generate the above kind of queries internally when implementing referential integrity and *cascade-on-delete* is to be applied [36]: assume that we have two tables X, Y and a key k. In table X

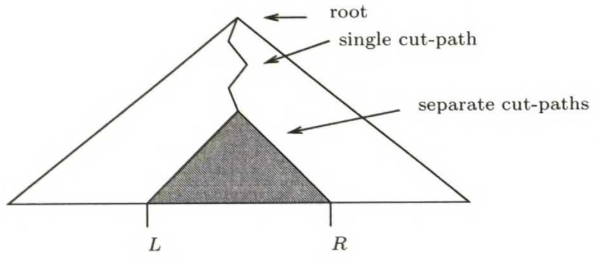

Figure 3.1: B$^+$-tree interval deletion operation removes the darkened area from the tree and balances the tree.

key k is the primary key and in table Y key k is part of a foreign key (k, x) and has a B$^+$-tree index I. Now, we perform deletion in the table X and since cascade-on-delete is applied, there will be a number of interval deletion operations performed in table Y's index I.

3.2 Previous Work

In this section, we survey various currently known algorithms for the interval deletion problem and discuss the assumptions that they are based on.

Normal deletion

The trivial approach to interval deletion problem is to perform ordinary deletion to every key that lies in the range $[L, R]$. This is very suboptimal since in the worst case there is $O(m \log n)$ balancing operations where m is the length of the interval and n is the number of keys in the tree.

Deletion with tree splitting and concatenation

In [15] Hoffman and Mehlhorn describe operations for splitting a B-tree into two halves and concatenating two B-trees together. They also present an interval deletion algorithm that uses these tree splitting and concatenation operations. The basic idea is to split the tree twice. First the tree is split with the key L to obtain two subtrees: one containing the keys that are smaller than L and one containing the keys larger or equal to L. Then the latter subtree is split again with the key R. The tree is now partitioned into tree subtrees one of which contains the deleted interval and the others rest of the tree. These other parts are concatenated together in order to produce a B-tree without the deleted interval. The splitting and concatenation algorithms are presented more thoroughly in [35, pp. 213–215].

Top-down three-pass algorithm

Carey et al. describe in [6, 7] a method for performing a range deletion in a B$^+$-tree index. The algorithm performs three passes to the tree. The first pass is the deletion phase where the tree is traversed from root towards the left and right limits of the deletion. The algorithm removes all subtrees that are contained in the deletion interval and updates the node counters to reflect this situation. Also, path from root to the leaves is recorded. In the second pass the path is traversed from the limiting leaves to the root in bottom-up manner and certain nodes are marked to be *in danger*. In the third pass the tree is once again traversed from the root towards the leaves and the nodes that are in danger will be rebalanced within their neighbouring nodes.

Bulk deletion with reorganisation

In [45] Zou and Salzberg describe a method for performing reorganisations to a B$^+$-tree index. Their method is to first compress the leaves so that they form a continuous region in a disk storage. The reorganisation is done *in-place*, i.e., the actual pages are moved. After this the upper levels of the tree are compressed.

This moving is done *new-place.* That is, new pages are allocated for the internal pages and information is moved to these pages. Concurrent operations that try to modify the pages already moved into a new place are collected into a *side file*, which is merged into the tree after the entirely reorganisation is complete. This approach guarantees that only one internal node needs to be locked at a time. After the internal pages have been reconstructed, the root pointer is switched to point this newly created tree.

Gärtner adapts the algorithm presented by Zou and Salzberg for the bulk deletion problem in [13]. Gärtner's approach is to perform the reorganisation only into some parts of the tree. That is, the algorithm works by find a *base node* which is typically chosen so that the subtree rooted at the base node can fit in the main memory. Then the bulk deletion is performed on the leaves of this subtree and the subtree is reorganised.

This algorithm is for generic bulk deletion where there is a predefined set of keys to be deleted from the tree. The keys need not form any intervals, i.e. there is no need to have consecutive keys deleted from the tree. In principle, this algorithm will compress the entire tree and avoids clustering. The algorithm is, however, not very well suited for generic interval deletion, because it will perform lots of extra work due to the compression and de-clustering.

Other approaches

Mohan considers interval deletion in [36] but the focus of his paper is in the concurrent handling of the interval deletion transaction and he omits entirely the structure modifications that are done to the B$^+$-tree index.

3.3 Top-down single-pass Interval Deletion

The new interval deletion algorithm presented here is top-down single pass. It performs removing and compression during the single traversal from the root to leaf nodes. Unlike in the bottom-up algorithms, there is no need to store the entire search path and only two levels of the tree need to be locked at the same time.

The general idea behind the algorithm is to delete all subtrees contained entirely in the interval $[L, R]$ from an (internal) node and make the subtrees that contain the left and right key adjacent in the node. After this, the node is compressed with its neighbouring nodes or, in the case of root, simply removed if the node becomes under-full. After compression is done, the algorithm descends to the lower level and performs the same operations there. This process is repeated until the leaf nodes are reached.

The algorithm starts by obtaining a pointer to the root and the limiting keys L and R. See Algorithm 14.

Algorithm 14 Interval Deletion: main algorithm

```
INTERVAL-DELETE(root : node, $L$ : key, $R$ : key):
1: PROCESS-ROOT(root, $L$, $R$)
```

Processing the root

There are three stages in the algorithm. In the first stage the root node is processed (see Algorithm 15). First we remove the elements (subtrees if the root is not a leaf) that are entirely contained in the interval $[L, R]$ (line 1). Then we are all done, if the root is a leaf (lines 2–4). If root is internal, we search for the subtrees (or keys) that contain the keys L and R. If the keys are located in the same subtree node *child*, we remove all elements from this node and balance the node (lines 7–10). Note that this can cause the root node to became under-full. In the other case, the deletion path diverges and subtrees are not rooted in the same child of the root. Then we perform range deletion separately for the left and right child (lines 12–16). Now, we check whether the root node has become unbalanced and balance it (lines 18–20). This is analogous to the operation performed in ordinary top-down deletion. This is the only place where the height of the tree can decrease. We perform a tail-recursive call to PROCESS-ROOT and start reprocessing the balanced node. Otherwise, if the root node has not become unbalanced, we search the limiting subtrees once again (lines 22–23) and if the keys are in the same subtree we call PROCESS-SINGLE. Otherwise PROCESS-SEP is called. Once PROCESS-SINGLE or PROCESS-SEP is called there will be no more operations to the root node and the height of the tree cannot decrease anymore.

Processing single cut-path

In the second phase we the deletion paths do not diverge. The process is presented in Algorithm 16. If the node is leaf, we are all done (lines 1–2). Otherwise we look the indices to the subtrees that contain the subtrees with keys L and R (lines 3–4). If they are contained in the same subtree we move to process this node called *child*. First we remove the subtrees that are entirely contained in the range $[L, R]$ from the child and then we balance the child (lines 7–9). If the subtrees are not contained in the same child node, we remove the subtrees contained in the range $[L, \infty]$ ($[-\infty, R]$) from the left (right) child and balance the children (lines 11–16). After this we search the subtrees once again from the parent node. If the keys L and R are now in the same child we tail-recursively call PROCESS-SINGLE with this child (lines 19–21). Otherwise, the deletion paths diverge and we call the procedure PROCESS-SEP to handle the separated cut paths (lines 23–25).

Processing separate cut-paths

The final phase is presented in Algorithm 17. In here, the cut paths have diverged and it is not possible to rebalance the tree so that the cut paths will merge together. If the parent nodes are leaves, then we are all done (lines 1–2). Otherwise, there are two subtrees rooted at nodes *leftparent* and *rightparent*. We first descend to the children of these nodes and remove the subtrees that lie entirely in the interval $[L, R]$ (lines 4–9). After this we balance these child nodes (lines 10–11). We have to look the pointers to the child nodes once again because the balancing phase might have changed them (lines 12–15). Finally we tail-recursively call PROCESS-SEP (line 16).

Algorithm 15 Interval Deletion: processing of the root

```
PROCESS-ROOT(root : $node$, $L$ : $key$, $R$ : $key$):
DELETE-RANGE(root, $L$, $R$)
if root.leaf then
    return root
end if
i $\leftarrow$ NODE-SEARCH(root, $L$)
j $\leftarrow$ NODE-SEARCH(root, $R$)
if i = j then
    child $\leftarrow$ root.p[i]
    DELETE-RANGE(child, $L$, $R$)
    ID-BALANCE-CHILD(root, i)
else
    left $\leftarrow$ root.p[i]
    right $\leftarrow$ root.p[j]
    DELETE-RANGE(left, $L$, $\infty$)
    DELETE-RANGE(right, $-\infty$, $R$)
    ID-BALANCE-CHILDREN(root, i, j)
end if
if not root.leaf and root.n = 1 then
    root $\leftarrow$ root.p[0]
    return PROCESS-ROOT(root, $L$, $R$)
end if
i $\leftarrow$ NODE-SEARCH(root, $L$)
j $\leftarrow$ NODE-SEARCH(root, $R$)
if i = j then
    child $\leftarrow$ root.p[i]
    return PROCESS-SINGLE(child, $L$, $R$)
else
    left $\leftarrow$ root.p[i]
    right $\leftarrow$ root.p[j]
    return PROCESS-SEP(left, right, $L$, $R$)
end if
```

Algorithm 16 Interval Deletion: processing single cut-path

```
PROCESS-SINGLE(parent : node, $L$ : key, $R$ : key):
 1: if parent.leaf then
 2:     return
 3: end if
 4: i ← NODE-SEARCH(parent, $L$)
 5: j ← NODE-SEARCH(parent, $R$)
 6: if i = j then
 7:     child ← parent.p[i]
 8:     DELETE-RANGE(child, $L$, $R$)
 9:     ID-BALANCE-CHILD(parent, i)
10: else
11:     left ← parent.p[i]
12:     right ← parent.p[j]
13:     DELETE-RANGE(left, $L$, $\infty$)
14:     DELETE-RANGE(right, $-\infty$, $R$)
15:     ID-BALANCE-CHILDREN(parent, i, j)
16: end if
17: i ← NODE-SEARCH(parent, $L$)
18: j ← NODE-SEARCH(parent, $R$)
19: if i = j then
20:     child ← parent.p[i]
21:     return PROCESS-SINGLE(child, $L$, $R$)
22: else
23:     left ← parent.p[i]
24:     right ← parent.p[j]
25:     return PROCESS-SEP(left, right, $L$, $R$)
26: end if
```

Algorithm 17 Interval Deletion: processing separate cut-paths

```
PROCESS-SEP(leftparent : node, rightparent : node, $L$ : key, $R$ : key):
 1: if leftparent.leaf then
 2:     return
 3: else
 4:     i ← NODE-SEARCH(leftparent, $L$)
 5:     j ← NODE-SEARCH(rightparent, $R$)
 6:     leftchild ← leftparent.p[i]
 7:     rightchild ← rightparent.p[j]
 8:     DELETE-RANGE(leftchild, $L$, $\infty$)
 9:     DELETE-RANGE(rightchild, $-\infty$, $R$)
10:     ID-BALANCE-CHILD(leftparent, i)
11:     ID-BALANCE-CHILD(rightparent, j)
12:     i ← NODE-SEARCH(leftparent, $L$)
13:     j ← NODE-SEARCH(rightparent, $R$)
14:     leftchild ← leftparent.p[i]
15:     rightchild ← rightparent.p[j]
16:     return PROCESS-SEP(leftchild, rightchild, $L$, $R$)
17: end if
```

Removing elements from a node

Next, we describe the algorithm used to perform the removal of elements from a node. The pseudo code for this is in Algorithm 18. The procedure DELETE-RANGE is somewhat complex due to the fact that it has to operate both internal nodes and leaves. In internal nodes all subtrees contained entirely in the interval has to be removed but not the bordering subtrees that contain the interval limits.

In leaf nodes, every key that lies in the interval deletion range has to be removed. First, assume that the node where the deletion operation is performed is a leaf. First we check that the left limit is contained in the interval, if not we fix the left limit accordingly (lines 4–6). The same is done for the right limit (lines 7–9). After this we simply shift the keys towards left so that the interval is removed from the tree (lines 10–12) and update the counter (line 13).

In an internal node, if the limits are consecutive (line 15) then nothing will be removed from the node. Otherwise, there is at least one subtree that is entirely contained in the interval. Remove the routers (lines 16–18) and subtrees (lines 19–21) and update the counter (line 22).

Rebalancing the node

There are two procedures for rebalancing given in Algorithms 19 and 20. The first algorithm ID-BALANCE-CHILD balances single child and is very similar to ordinary (a, b) tree rebalancing algorithms (see Algorithms 8 and 13 in Chapter 2). The other procedure ID-BALANCE-CHILDREN balances two successive child nodes.

Rebalancing after DELETE-RANGE differs from the rebalancing done after ordinary single deletion algorithm. In an ordinary B$^+$-tree deletion a node loses at most one child during the deletion operation. Thus, it is enough to have $a + 1$ children in the parent node in order to guarantee the balance. In interval deletion, after the DELETE-RANGE has been performed the node can contain 1 or more children (if the node is internal) or 0 or more keys (if the node is a leaf). Thus, the balancing must take these special cases into account.

The procedure ID-BALANCE-CHILD is applied when there is only single node where the keys were removed. First we check whether the parent has only a single child and return immediately if so (lines 2–3). This scenario is possibly only if parent is the root. If so, the PROCESS-ROOT will remove the old root and lift this new node as the new root. Otherwise, if the node is empty we simply remove it (lines 4-5). If the node is a leaf and has less than a elements, the node is rebalanced (lines 6–8). If the node is internal and has less than $a + 2$ elements, we balance it (lines 8–9).

If the deletion paths diverge there are two child nodes that are the roots of the subtrees containing the keys L and R. Algorithm 20 handles this case. The procedure ID-BALANCE-CHILDREN first checks whether either of the children is empty (this can only occur if the nodes are leaves). If this is the case, the empty node is removed and ID-BALANCE-CHILD is called for balancing the other, possible unbalanced, node (lines 3–8). If both nodes have less than or equal to $a + 2$ elements, they are fused together and ID-BALANCE-CHILD is called to perform the balancing for the fused node (lines 9–11). If the sum of the elements is less than or equal to the node size b, then the nodes will be fused together and no further balancing is needed (lines 12–14). Otherwise the contents of the nodes are shared (lines 14–18). Note that the sharing in lines 14–17 is always possible and only one of the nodes need to be shared, because the other one is in balance anyway. This is a consequence of the fact that the sum of the elements in the two nodes is greater than b if the line 14 is reached.

Algorithm 18 Interval Deletion: deleting elements from a node

```
DELETE-RANGE(child : node, L : key, R : key):
 1: l ← NODE-SEARCH(child, L)
 2: r ← NODE-SEARCH(child, R)
 3: if child.leaf then
 4:    if child.key[l] < L then
 5:       l ← l + 1
 6:    end if
 7:    if child.key[r] ≥ R then
 8:       r ← r + 1
 9:    end if
10:    for i = r to n − 1 do
11:       child.key[l + (i − r)] ← child.key[i]
12:    end for
13:    child.n ← n − (r − l)
14: else
15:    if r − l > 1 then
16:       for i = r to n − 2 do
17:          child.key[l + 1 + (i − r)] ← child.key[i]
18:       end for
19:       for i = r to n − 1 do
20:          child.p[l + 1 + (i − r)] ← child.p[i]
21:       end for
22:       child.n ← n − (r − l − 1)
23:    end if
24: end if
```

Algorithm 19 Interval Deletion: balancing single child

```
ID-BALANCE-CHILD(parent : node, i : index):
 1: child ← parent.p[i]
 2: if parent.n = 1 then
 3:     return
 4: else if child.n = 0 then
 5:     DELETE-CHILD(parent, i)
 6: else if child.leaf and child.n < $a$ then
 7:     COMPRESS-CHILD(parent, i)
 8: else if not child.leaf and child.n < $a + 2$ then
 9:     COMPRESS-CHILD(parent, i)
10: end if
```

Algorithm 20 Interval Deletion: balancing two children

```
ID-BALANCE-CHILDREN(parent : node, i : index, j : index):
 1: child-1 ← parent.p[i]
 2: child-2 ← parent.p[j]
 3: if child-1.n = 0 then
 4:     DELETE-CHILD(parent,i)
 5:     ID-BALANCE-CHILD(parent,j)
 6: else if child-2.n = 0 then
 7:     DELETE-CHILD(parent,j)
 8:     ID-BALANCE-CHILD(parent,i)
 9: else if child-1.n + child-2.n ≤ a + 2 then
10:     FUSE-CHILD(parent, i)
11:     ID-BALANCE-CHILD(parent, i)
12: else if child-1.n + child-2.n ≤ b then
13:     FUSE-CHILD(parent, i)
14: else if child-1.n < a + 1 then
15:     SHARE-CHILD(parent, i)
16: else if child-2.n < a + 1 then
17:     SHARE-CHILD(parent, j)
18: end if
```

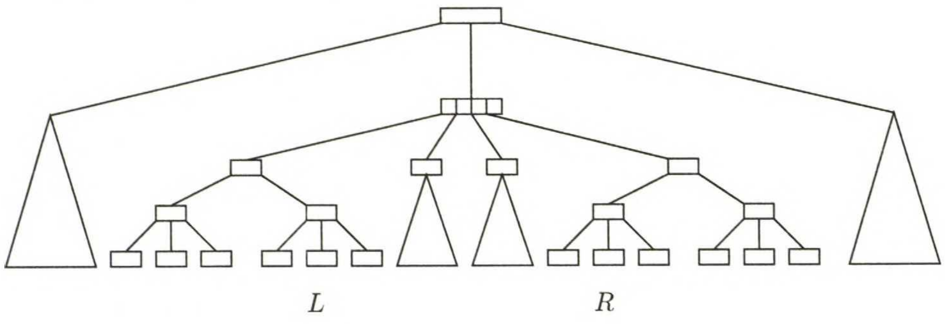

Figure 3.2: The beginning. The deletion interval is contained in a single subtree of the root thus no deletion occurs at root level.

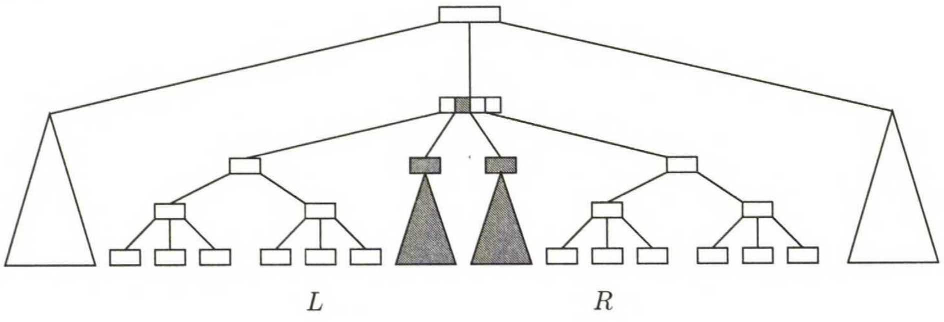

Figure 3.3: Locating *lca* and removing subtrees.

3.4 Example

An example run on a sample tree can be found from Figures 3.2–3.6. At first the root node is not changed at all (Figure 3.2). The lowest common ancestor node where the deletion begins is located at the second level of the tree (Figure 3.3). First the two middle subtrees are removed and the node is balanced. After this, the deletion proceeds to the third level of the tree (Figure 3.4). The keys to be removed are now located on the rightmost (leftmost) subtrees, thus no subtrees can be removed at this time. No balancing or deletion occurs but the algorithm descends to the fourth level of the tree (Figure 3.5). On the fourth level the rightmost (leftmost) subtrees are entirely contained in the interval and will be removed from the tree and the node is balanced. Finally the algorithm descends to the leaves containing the keys L and R and removes all keys that lie in the interval. After this, the leaves are balanced and the algorithm terminates. (Figure 3.6).

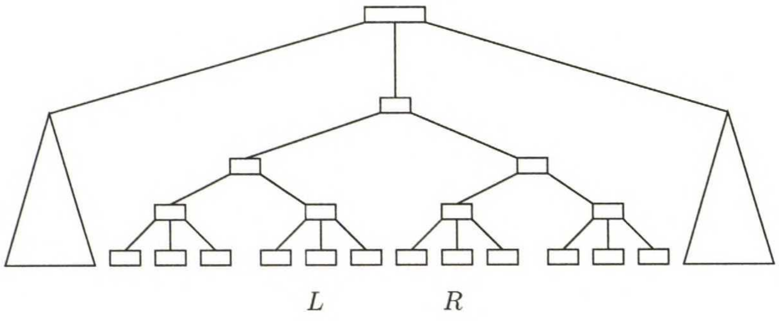

Figure 3.4: No deletion or balancing occurs since the keys are located in the leftmost (rightmost) subtrees.

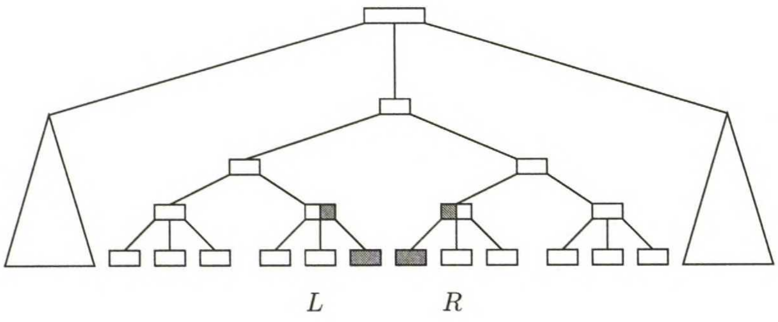

Figure 3.5: The leftmost (rightmost) subtrees are now entirely contained in the interval and will be removed. After removing, the nodes are balanced.

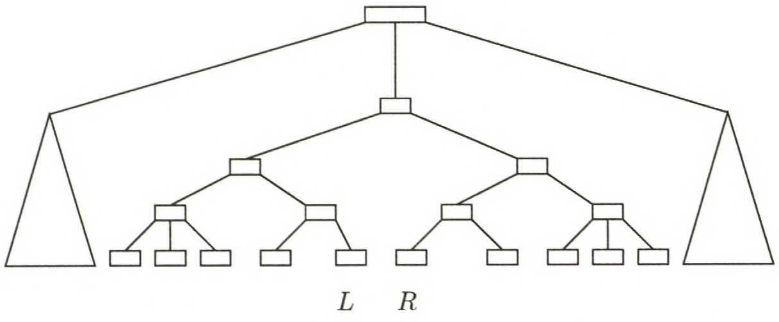

Figure 3.6: Finally the algorithm removes the keys that lie to the left (right) of the key R (L) and balance the leaves.

3.5 Correctness

In this section we discuss the correctness of the interval deletion. First we consider the single cut-path and then the separate cut-paths case.

Single cut-path

Now we consider the case where the deletion does not diverge. That is, the elements are remove from a single child node. See Figure 3.7.

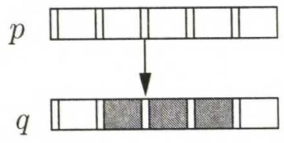

Figure 3.7: Single child deletion. Marked elements will be removed.

We prove the following lemma:

Theorem 3.1. *In the single cut-path case after removing the elements contained in the interval $[L, R]$ from node q with the procedure* DELETE-RANGE, *the node q has at least two children if the node is internal. If the node is a leaf, it has $0 \leq c(q) \leq b$ keys.*

Proof. If q is internal, only the subtrees that are entirely contained in the interval will be removed. No other subtrees can be removed. Especially, the subtrees that contain the keys cannot be removed, since they can contain both the keys that belong to the interval and the keys that do not. That is, the number of children is $2 \leq c(q) \leq b$.

If q is a leaf, the procedure DELETE-RANGE can remove no keys, all keys or between none and all. That is, the number of keys is $0 \leq c(q) \leq b$. □

Now, we prove the correctness of the balancing phase of the interval deletion algorithm in the single cut-path case. The singe cut-path processing is done in Algorithm 16.

Theorem 3.2. *In the single cut-path case after deleting elements from node q, balancing can always be done by either sharing or fusing with the procedure* ID-BALANCE-CHILD.

Proof. Call the unbalanced node q, its neighbour r and the parent p. The neighbour is obviously in balance, that is, it has $a \leq c(r) \leq b$ elements and naturally $c(q) \leq a + 1$. If q is an internal node, it must have $2 \leq c(q) \leq b$ children before balancing according to Theorem 3.1. Before balancing the parent has $a + 1 \leq c(p) \leq b$ children.

If $c(q) = 0$, it is the case that q is a leaf and it will be removed from the parent and no further balancing occurs.

Now, if $c(q) + c(r) \leq 2a + 1$, the contents of the node q and r are fused together. Call the combined node q'. It is obvious that $a + 2 \leq c(q') \leq 2a + 1$, if q is internal, and $a + 1 \leq c(q') \leq 2a + 1$ if q is leaf. Thus, node is in balance. The parent has now $a \leq c(p') \leq b$ children and is still in balance.

If $c(q)+c(r) > 2a+1$, the contents of the nodes are shared. Let q' and r' be the nodes after sharing. Because $c(q) \geq 2$ if q is internal, it is always possible to share the contents so that $a+2 \leq c(q'), c(r') \leq b$. If q is a leaf, sharing yields $a+1 \leq c(q'), c(r') \leq b$. Sharing does not affect parent's number of children.

In conclusion, balancing in a single cut-path case is always possible and will yield a balanced tree. The node where the interval deletion proceeds will have $c(q') \geq a+2$ children if it is internal. □

Separate children

Next, we consider the case of separate cut-paths and prove the correctness of this phase. This processing done in the interval deletion Algorithm 17.

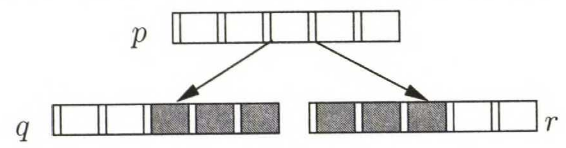

Figure 3.8: Separate children deletion. Marked elements will be removed.

Theorem 3.3. *In the separate-cut path case removing the range $[L, \infty]$ from node q and $[-\infty, R]$ from node r will leave the nodes $1 \leq c(q), c(r) \leq b$ children if the nodes are internal or $0 \leq c(q), c(r) \leq n$ keys if the nodes are leaves.*

Proof. If q (r) is internal, then the subtree containing the key L (R) is rooted at q (r). Now, the deletion cannot remove the subtree containing the key L (R), thus at least one subtree must remain after deletion.

If q (r) is a leaf, the deletion can remove no keys, all keys or something in between. □

The correctness of balancing after separate cut-paths is proven below.

Theorem 3.4. *In the separate cut-path case after deleting elements from nodes q and r balancing can always be done by either sharing or fusing with the procedure* ID-BALANCE-CHILDREN.

Proof. First, consider the case where q and r are internal nodes and let p be the parent node. Due to the interval deletion top-down balancing parent node has $a+2 \leq c(p) \leq b$ children.

First, if after deletion, one or both of the nodes q and r are empty, they are immediately removed and no further balancing occurs. According to Theorem 3.3, this can happen only if q and r are leaves.

Otherwise, if q and r are internal, we have $1 \leq c(q), c(r) \leq b$ and the parent has $a+2 \leq c(p) \leq b$ children.

If $c(q)+c(r) \leq 2a+1$ the nodes are fused together. If $c(q)+c(r) > a+2$ the combined node is in balance and the balancing stops. Otherwise, we need to rebalance the combined node with a neighbouring node once more. This is analogous to the processing done in the single cut-path case.

If $c(q)+c(r) > 2a+1$ and if $c(q) \leq a+1$ or $c(r) \leq a+1$, we can perform sharing so that both nodes have $c(q), c(r) \geq a+1$ elements. Note that the sharing is performed only with one of the nodes q and r, not both because if $c(q)+c(r) > 2a+1$ it cannot be the case that both $c(q), c(r) \geq a$. □

3.6 Amortised Analysis

Hoffman and Mehlhorn proved in [15] that the amortised cost of balancing in interval deletion problem by using tree splits and merges (see Section 3.2) for (a, b) trees has the amortised bound $O(\log m)$ where m denotes the length of the interval. Here we give a similar result for the top-down interval deletion algorithm of this thesis.

First we state a fact about the height of the subtree where the interval deletion modifies the tree.

Lemma 3.1. *Assume that the number of keys to be deleted with interval deletion is m. Then the actual deletion operations occur only in nodes whose height is less than or equal to $\log_a m$.*

Proof. By contradiction. Assume that the actual deletion operation removes a subtree q whose height is $h > \log_a m$. The subtree q has at least a^h leaves. By substitution we have $m = a^h > a^{\log_a m} = m$, which is impossible. □

Now, we are ready to formulate and give a proof of the amortised number of balancing operations when interval deletions are present.

Theorem 3.5. *Assume that i insertions, d deletions, and k interval deletions of size m_j, $1 \leq j \leq k$ are performed to an initially empty (a, b) tree where $b \geq 2a + 3$. The rebalancing operations needed during an insertion or deletion in this sequence have amortised $O(1)$ cost and the rebalancing operations needed after interval deletions have $O(\log m)$ cost.*

Proof. This proof is similar to that of Theorem 2.3 in Section 2.4. We use the following potential function for non-root nodes.

$$\Phi(u) = \begin{cases} 3, & c(u) \leq a \\ 1, & c(u) = a + 1 \\ 0, & a + 1 < c(u) < b - 1 \\ 2, & c(u) = b - 1 \\ 4, & c(u) = b \end{cases}$$

Naturally, for the root node the variable a is replaced with constant 2. The potential of the tree is the sum of the potential of all of its nodes.

The arguments given in Section 2.4 for normal insertion and deletion are still valid. Here we consider only the interval deletion.

Interval deletion

According to Lemma 3.1 the deletion operations will not occur on nodes higher than $h = O(\log m)$ where m is the number of keys in the deletion interval.

In the worst case, the deletion path diverges immediately so that there are $2h$ nodes where elements are deleted. After removing, each of the nodes may have less than a elements and needs to be balanced. The potential increase of a single node is at most 3 units. Thus, the total increase in the potential of the tree is at most $6h = O(\log m)$.

Rebalancing after interval deletion

In rebalancing we have two cases to be considered: single and separate cut-paths (see the Figures 3.7 and 3.8). In single cut path, we have given the node q and its parent p. Balancing occurs when $c(q) \leq a+1$. The parent has $a+1 \leq c(p) \leq b$ children.

In separate cut paths case, we have given the nodes q, r and the parent p. Balancing occurs if $c(q) \leq a+1$ or $c(r) \leq a+1$.

Fusing

- Single cut-path: Let q be the unbalanced node and p its parent. Now, we have $c(q) \leq a$. Call the node q' after fusing. Theorem 3.2 implies that after the operation we have $a+2 \leq c(q') \leq 2a+1 < b-1$ in the node q'. The potential is decreased by at least 3 units. Removing the router and a pointer from the parent may increase the parent's potential by at most 2 units. Thus, the total potential decrease is at least 1 unit.

- Separate cut-paths: Let q and r be the nodes. If fusing is to be applied $2 \leq c(q)+c(r) \leq 2a+1$. The balancing stops, if the combined node, say q', has $a+2 \leq c(q') < b-1$ elements. The potential decrease is at least 3 units. Removing a router and pointer from the parent may increase parent's potential by at most 2 units. Thus the total decrease is at least 1 unit.

  If $c(q')$ is not in balance, we need to perform fusing or sharing once more. If fusing is performed, it is analogous to the single cut-path case presented above. Thus, the total potential decrease is at least 1 unit.

Sharing

Sharing does not affect the potential of the parent node at all.

- Single cut-path: Let q be the unbalanced node before sharing and q' be it after sharing. Obviously $c(q) \leq a$ and according to Theorem 3.2 after sharing, we have $a+1 \leq c(q) < b-1$. Thus the potential is decrease by at least 2 units.

- Separate cut-paths: In separate cut-paths, sharing is applied only once. That is, if we have nodes q and r. Only one of them is shared and the other is already in balance. Call the node after sharing q'. According to Theorem 3.4, we have $a+1 \leq c(q') \leq b$ and thus the total decrease of the potential is at least 2 units.

Conclusion

The interval deletion part and the Theorem 2.3 imply that a sequence of intermixed insertion, deletion and interval deletion perform $O(i+d+\sum_{j=1}^{k}\log m_i)$ balancing operations. □

Chapter 4

The Experiments

4.1 Introduction

The experiments were carried in order to study the complexity of the top-down interval deletion algorithm. The goal was to study the number of balancing operations performed with respect to the deletion interval size and the memory/disk behaviour in a environment similar of a buffer pool in a relational database management system.

4.2 The Framework

A simulator environment was constructed and the experiments were carried within this environment. The simulator consists of instrumented algorithms, a buffer pool simulator, operation generation, a test execution module and the experiment execution system.

Test is a sequence of operations search, insert, delete and interval-delete operations performed to an empty B-tree. A test produces *log* which includes various informations described below. A sequence of tests with varying a parameter is called an *experiment*. After an experiment has been carried out, it is *analysed*.

The algorithms implemented for this testing system are normal top-down (a, b) tree search, insert and delete algorithms described in Section 2.3. Insertion and deletion algorithms were instrumented to log primitive operations for further analysis. The insertion algorithm logs every *split* operation that is performed. Deletion logs every balancing operation (*share* or *fuse*) that is performed during the execution of the algorithm. The interval deletion algorithm of Section 3.3 was implemented and instrumented to log balancing operations.

In order to study the memory behaviour of the algorithms a buffer pool simulator was constructed. The buffer pool simulator is somewhat similar to that used in Relational Database Management Systems (RDBMS). That is, the algorithms must *fix* pages before they can reference to them and *unfix* when the pages are no longer needed. There is a two-level hierarchy in the memory simulator: a disk and main memory. The main memory is of fixed size which is a parameter to the experiment. The disk is assumed to be infinitely large.

The buffer pool simulator counts the number of memory read/write and disk read/write operations and logs them after the experiment.

The operation generation module is related to an experiment. The module has a *common* and *specific* parts. The common part is common to all experiments carried in the simulator. The specific part contains variables related to a specific test. The common variables are the following:

a, b	Size of the (a, b) tree used in the test
num-pages	number of pages in the main memory buffer pool
seed1, *seed2*	seeds for the pseudo-random number generator

The specific variables are defined when the actual experiments are described later on this chapter. An experiment is formed by giving the above parameters and the specific part to each test. The independent variable is varied and the observed variables (memory references, balancing operations, ...) are measured and logged for further analysis. To make the tests completely deterministic even though randomness is desired a hand-written random number generator [42] implemented for [41] was used. The seeds are stored so that the tests can be easily repeated, which was desired for the debugging and, in some cases, for further analysis.

For statistical purposes, each test is repeated fixed number of times to make the statistical analysis possible. After the experiment has been completed the log file is analysed and interpreted by constructing plots and doing statistical analysis. The statistical methods used are described in the Section 4.3.

The testing framework is implemented with PLT MzScheme implementation for the *Scheme programming language* [23]. The implementation uses mostly standard Scheme with some MzScheme extensions. It would be rather straightforward to port the framework to some other Scheme implementation, should that be desired. Some AWK and Shell scripts were used to generate experiment data, that is, the individual tests. Statistical analysis and collecting the data for plotting was done with a PERL script. Plots were generated with GNUPLOT.

The environment where the experiments were carried was Intel Pentium 4 2.00 GHz, 1 GB RAM, with Linux 2.6.11.8 kernel and Debian GNU/Linux 3.1 distribution with GNU C library implementation version libc6 2.3.2.ds1-21. Debian packaged MzScheme interpreter version 209 was used. GNU AWK version 3.1.4. The simulation environment was completely deterministic and all measured qualities were fully simulated so the actual hardware and software platforms do not affect the results at all.

4.3 Statistical Methods

The goal of the experiments was to study and compare the various resource utilisations of the given algorithms. When comparing two or more algorithms it is important to perform some statistical analysis in order to gain knowledge on the statistical validity of the results.

The tests are constructed so that one parameter is varied, others are kept constant and some value is measured. A trivial approach would be to make one test per one parameter value, but this would easily give wrong conclusions: there is a high probability that the measured quality, or random variable, is subject to random fluctuation so that if only one test is performed the results are incorrect.

Thus, in order to gain some statistical validity, each test is repeated for $n = 40$ times. That is, for example, we measure two different algorithms for same performance measure and obtain the random variables x and y. We repeated the test, so that we obtain the datasets $(x_1, x_2, \ldots, x_{40})$ and $(y_1, y_2, \ldots, y_{40})$. Then we calculate the averages $\overline{x}$ and $\overline{y}$ for both of these vectors.

Now, the statistical method to gain some trust on the validity of the results is to do confidence interval analysis. That is, given a dataset, we want to know with some probability where the actual value x or y is. We construct 95% confidence intervals. That is, an interval for the random variable, where the actual value is located with probability 0.95. Before we can construct the confidence intervals, the distribution of the random variables must be known. Unfortunately, it is not known for the random variables x and y. But due to the Central Limit Theorem of probability, for the average of 40 independent repetitions should be normally distributed, no matter what the distribution for an individual random variable is.

To construct the 95% confidence interval for the expected (average) value when the variance is unknown, we use Student t distribution with $\nu = n-1 = 39$ degrees of freedom $P(|T| \geq t_{\alpha/2}) = \alpha$:

$$\overline{x} - t_{\alpha/2}\frac{s}{\sqrt{n}} \leq \mu \leq \overline{x} + t_{\alpha/2}\frac{s}{\sqrt{n}}, \quad \alpha = 0.05$$

The parameter s is the estimator for population standard deviation defined by

$$s^2 = \frac{1}{n-1}\sum_{i=1}^{n}(x_i - \overline{x})^2.$$

The parameter t is obtained from the Student t distribution table. For $\nu = 39$ degrees of freedom with 95% confidence, $t_{\alpha/2} \approx 2.023$.

If the confidence intervals for the algorithms overlap, then it cannot be statistically said which of the algorithms performs better. In each experiment described in the next sections it will be always indicated whether the results differ so that the confidence intervals do not overlap. That is, the difference is statistically meaningful.

4.4 Size of the Deletion Interval

The goal of this experiment was to study how the size m of the deletion interval affects on the number of balancing and memory/disk operations performed. The algorithms that were compared were the normal (a, b) tree top-down deletion and interval deletion (A-B-INTERVAL-DELETE, Algorithm 14). In addition to the common parameters presented in Section 4.2 there were two specific parameters: number of initial insertions and the size of the interval. The number of initial insertions n was kept constant during the experiment and the size of the interval was varied.

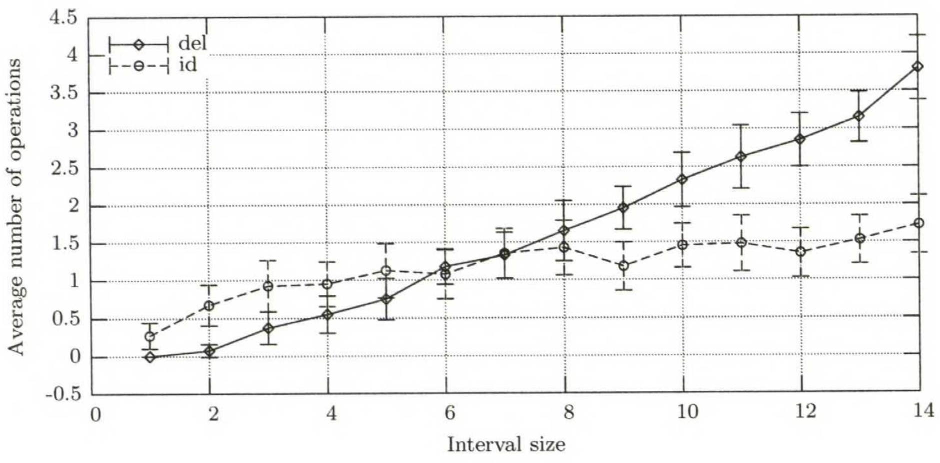

Figure 4.1: Number of balancing operations (interval size 1–15).

The following parameters were used to conduct the experiment:

a	2
b	7
num-pages	10
n	1000
m	$\{1, 2, \ldots, 15\}$, $\{50, 100, 150, \ldots, 1000\}$

Naturally, for each value of m, 40 random number seeds were generated so that each test could be repeated.

The measured quantities were total number of balancing operations (share and fuse operations, Algorithms 9 and 10) and number of memory and disk references. Figure 4.1 presents the average number of balancing operations in $n = 40$ repeated tests with small intervals. Larger intervals are presented in Figure 4.2. The same plot with just interval deletion operations is pictured in Figure 4.3. All of these figures the y-axis error-bars describe the 95% confidence intervals.

From Figure 4.1 it is clear that for very small values of m, the traditional deletion algorithm performs better than interval deletion. This is obvious because the balancing condition of interval deletion differs that of normal deletion. Thus, interval deletion has to do some extra work.

In Figures 4.2 and 4.3 we see the behaviour of the two algorithms with larger deletion interval sizes. From Figure 4.2 it is clear that the asymptotical behaviour of the normal deletion algorithm is clearly linear with respect to the deletion interval size m whereas it seems that the number of balancing operations needed in interval deletion is almost constant. In Figure 4.3 we have the same plot as in Figure 4.2 but the normal deletion line omitted. From Figure 4.3 we see that at first the number of operations increase until we reach interval sizes about the half of the tree size. After this the number of balancing operations is about constant until it drops to the height of the tree when we reach the maximum interval deletion size.

Figure 4.4 and 4.5 present the number of memory read and write operations

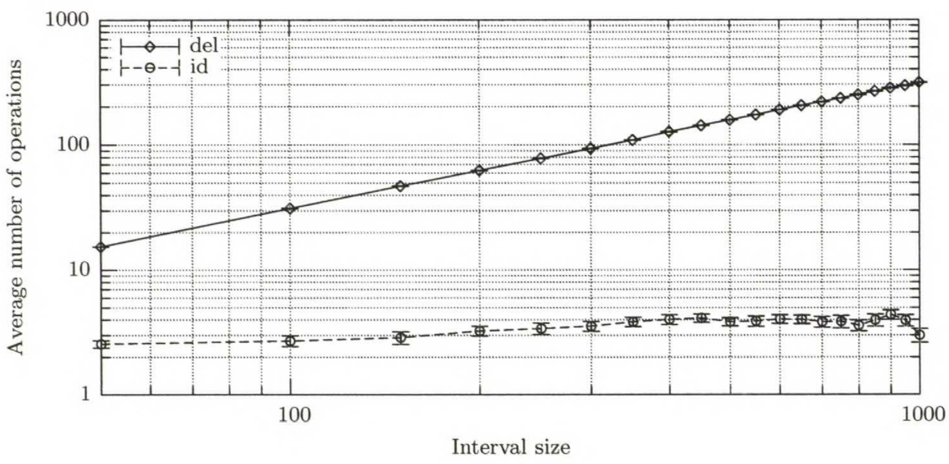

Figure 4.2: Number of balancing operations (interval size 50–1000).

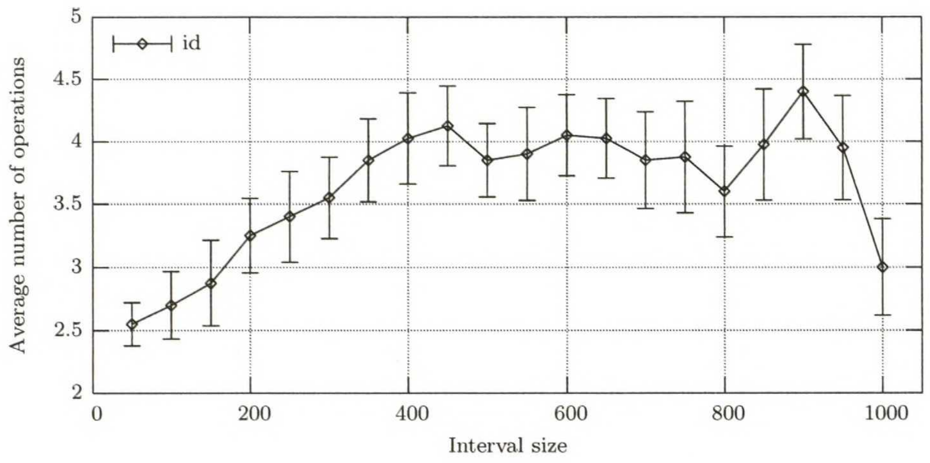

Figure 4.3: Number of balancing operations (interval size 50–1000, ID only).

performed during the experiment. From the figures it is clear that the number of memory operations performed by the normal deletion is linear and by the interval deletion is sub-linear. Similar results can be seen in Figures 4.6 and 4.7 when measuring the number of disk operations performed.

From the figures one can conclude that the behaviour of the normal deletion algorithm is linear and interval deletion algorithm is sub-linear with respect to the interval deletion size with various performance measures. It is not seen here, however, whether the number of operations performed in interval deletion is in fact logarithmic with respect to the interval size as the theory of Section 3.6 proves.

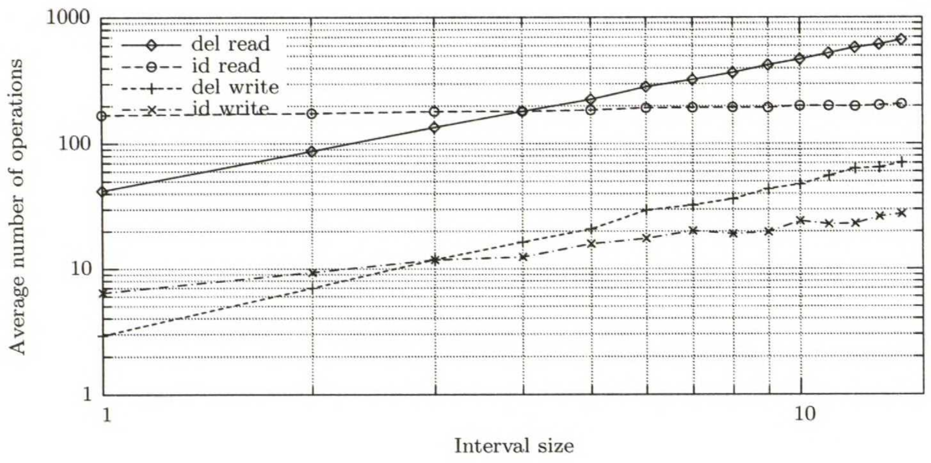

Figure 4.4: Number of memory operations (interval size 1–15).

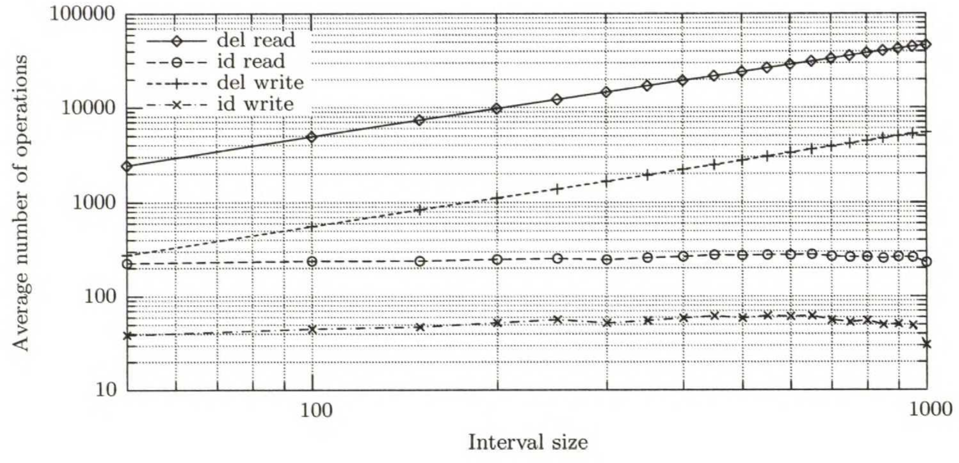

Figure 4.5: Number of memory operations (interval size 50–1000).

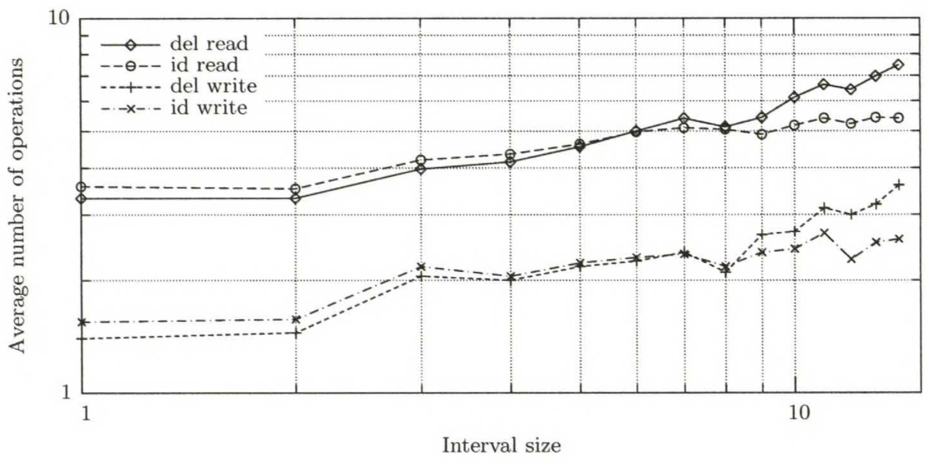

Figure 4.6: Number of disk operations (interval size 1–15).

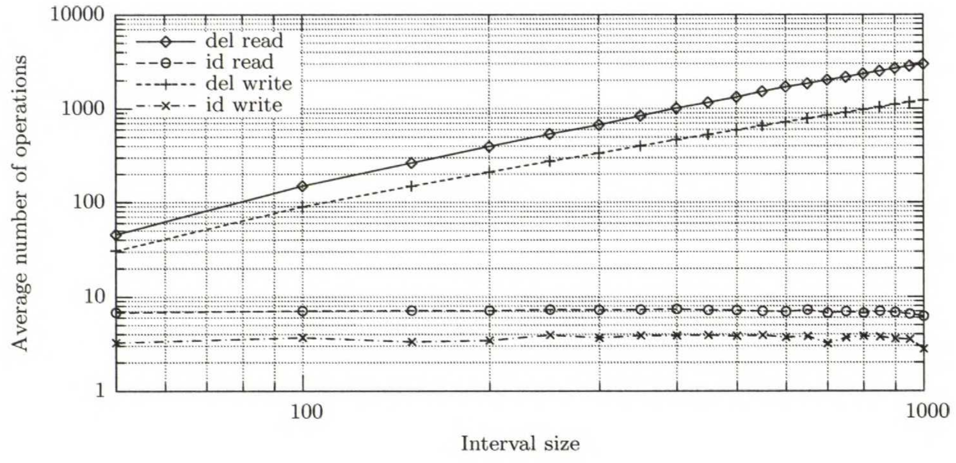

Figure 4.7: Number of disk operations (interval size 50–1000).

Chapter 5

Conclusions

In this thesis we described and surveyed some of the most common variants of the B-tree structure. To gain some historical perspective, we described the original B-trees, the most commonly used variant B$^+$-trees and (a, b) trees which are a especially suitable for theoretical analysis. We considered two common balancing strategies: bottom-up and top-down and discussed their pros and cons.

We extended the normal dictionary abstract data type to include a special case of bulk deletion operation called interval deletion. In interval deletion a key-range is removed from a B-tree so that the tree remains balanced and a proper B-tree. We surveyed the current approaches presented in the literature and presented a novel approach where the interval deletion is performed with a top-down balanced single-pass algorithm. We proved that in a sequence of intermixed insertion, deletion and interval deletion operations, a $O(\log m)$ balancing operations is performed when the number of balancing operations is amortised over the length of the operation sequence.

In the experimental study, we tried to find empirical data to validate the theoretical results obtained and to compare the interval deletion algorithm with the normal top-down deletion algorithm. We found that the normal deletion algorithm works linearly with respect to the interval size, whereas the top-down interval deletion algorithm perform a sub-linear amount of work.

We conclude from the theory and experiments show that with intervals larger than the node size the interval deletion algorithm performs better than the ordinary deletion algorithm.

In this thesis we omitted concurrency issues altogether. In a real database management system concurrency is a very important issue that cannot be neglected. We merely stated that the top-down algorithm is better for concurrency control since it requires only constant amount of locks whereas bottom-up might require locks proportional to the height of the tree. Naturally, concurrency could also be improved with strategies that involve neither top-down or bottom-up balancing but special constructs, like B$^{\text{link}}$-trees, with which it is possible to lock only single node at a time.

For further research I would suggest that the top-down interval deletion algorithm were extended to take the concurrency and memory management into account. Both the theoretical model and the experimental environment should be developed so that the concurrency could be studied further.

The interval deletion algorithm could be modified to use bottom-up balancing (similar or that of [7]) and B$^{\text{link}}$-tree style "lazy" balancing. These modified variants could be compared to the original top-down balancing presented in this thesis in order to gain better understanding of the behaviour of the various balancing schemes.

Theoretical studying and experiments are always idealisations of the real world situations. Thus, in order to gain better understanding and better results a real-life implementation of the interval deletion algorithm could be devised. For example the indexing structures of some open source database management system, like POSTGRESQL or MYSQL, could be extended to include the interval deletion operation. Naturally this would include a lot of extra work into the algorithms presented in this thesis and to various parts of the relational database management system. E.g., the RDBMS query optimiser should recognise queries that can be reduced into interval deletion operations. Unfortunately, MYSQL has bottom-up balanced B$^+$-trees and POSTGRESQL has B$^{\text{link}}$-trees, so the algorithms presented in this thesis cannot be directly "plugged in" even into the indexing parts of these systems.

Bibliography

[1] H. Abelson, G. J. Sussman, and J. Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, second edition, 1996.

[2] G. M. Adelson-Velsky and E. M. Landis. An Algorithm for the Organization of Information. *Soviet Mathematics*, 3:1259–1263, 1962. English translation.

[3] P. Agarwal, L. Arge, O. Procopiuc, and J. Vitter. A Framework for Index Bulk Loading and Dynamization. In *ICALP '01: Proceedings of the 28th International Colloquium on Automata, Languages and Programming*, pp. 115–127. Springer-Verlag, London, UK, 2001.

[4] L. Arge, K. Hinrichs, J. Vahrenhold, and J. Vitter. Efficient Bulk Operations on Dynamic R-trees. In *ALENEX '99: Proceedings of the 1st Workshop on Algorithm Engineering and Experimentation, Lecture Notes in Computer Science 1619*, pp. 328–348. Springer-Verlag, 1999.

[5] R. Bayer and E. McCreight. Organization and Maintenance of Large Ordered Indexes. *Acta Informatica*, 1(3):173–178, 1972.

[6] M. J. Carey, D. J. DeWitt, J. E. Richardson, and E. J. Shekita. Object and File Management in the EXODUS Extensible Database System. In *VLDB'86: Proceedings of the 12th International Conference on Very Large Data Bases*, pp. 91–100. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1986.

[7] M. J. Carey, D. J. DeWitt, J. E. Richardson, and E. J. Shekita. *Storage management for objects in EXODUS, Object-Oriented Concepts, Databases and Applications.*. Addison-Wesley, 1989.

[8] D. Comer. Ubiquitous B-Tree. *ACM Computing Surveys*, 11(2):121–137, 1979.

[9] J. V. den Bercken, B. Seeger, and P. Widmayer. A Generic Approach to Bulk Loading Multidimensional Index Structures. In *VLDB '97: Proceedings of the 23rd International Conference on Very Large Data Bases*, pp. 406–415. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1997.

[10] K. W. Deschler and E. A. Rundensteiner. B+ Retake: Sustaining High Volume Inserts Into Large Data Pages. In *Proceedings of the 4th ACM International Workshop on Data Warehousing and OLAP*, pp. 56–63. ACM Press, 2001.

[11] R. Fenk, A. Kawakami, V. Markl, R. Bayer, and S. Osaki. Bulk Loading a Data Warehouse Built upon a UB-tree. In *Proceeding of IDEAS Conference*. 2000.

[12] L. J. Guibas and R. Sedgewick. A Dichromatic Framework for Balanced Trees. In *In 19th Annual Symposium on Foundations of Computer Science*, pp. 8–21. IEEE Computer Society.

[13] A. Gärtner, A. Kemper, et al. Efficient Bulk Deletes in Relational Databases. In *Proceedings of the 17th International Conference on Data Engineer*, pp. 183–192. IEEE Computer Society, 2001.

[14] S. Hanke and E. Soisalon-Soininen. Group Updates for Red-Black Trees. In *Proceedings of the 4th Italian Conference on Algorithms and Complexity, Lecture Notes in Computer Science 1767*, pp. 253–262. Springer-Verlag, 2000.

[15] K. Hoffmann, K. Mehlhorn, P. Rosenstiehl, and R. E. Tarjan. Sorting Jordan Sequences in Linear Time Using Level-Linked Search Trees. *Information and Control*, 68:170–184, 1986.

[16] S. Huddleston and K. Mehlhorn. Robust Balancing in B-trees. In *5th GI-Conference on Theoretical Informatics, Lecture Notes in Computer Science 104*, pp. 234–244. 1981.

[17] S. Huddleston and K. Mehlhorn. A New Data Structure for Representing Sorted Lists. *Acta Informatica*, 17:157–184, 1982.

[18] L. Jacobsen, K. S. Larsen, and M. N. Nielsen. On the Existence and Construction of Non-extreme (a, b)-trees. *Information Processing Letters*, 84(2):69–73, 2002.

[19] J. Jannink. Implementing Deletion in B^+-trees. *SIGMOD Record*, 24(1):33–38, 1995.

[20] C. Jermaine, A. Datta, and E. Omiecinski. A Novel Index Supporting High Volume Data Warehouse Insertion. In *VLDB '99: Proceedings of the 25th International Conference on Very Large Data Bases*, pp. 235–246. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1999.

[21] T. Johnson and D. Sasha. The Performance of Current B-tree Algorithms. *ACM Transactions Database Systems*, 18(1):51–101, 1993.

[22] B. Kantor and P. Lapsley. RFC 977: Network News Transfer Protocol. 1986.

[23] R. Kelsey, W. Klinger, and J. Rees. Revised^5 Report on the Algorithmic Language Scheme. *Higher-Order and Symbolic Computation*, 26(11):895–901, 1998.

[24] D. E. Knuth. *Sorting and Searching*, volume 3 of *The Art of Computer Programming*. Addison-Wesley, Reading, Massachusetts, first edition, 1973.

[25] D. E. Knuth. *Sorting and Searching*, volume 3 of *The Art of Computer Programming*. Addison-Wesley, Reading, Massachusetts, second edition, 1998.

[26] S. D. Lang, J. R. Driscoll, and J. H. Jou. Improving the Differential File Technique via Batch Operations for Tree Structured File Organizations. In *Proceedings of the Second International Conference on Data Engineering*, pp. 524–532. IEEE Computer Society, Washington, DC, USA, 1986.

[27] V. Lanin and D. Shasha. A Symmetric Concurrent B-tree Algorithm. In *Proceedings of 1986 Fall Joint Computer Conference*, pp. 380–389. IEEE Computer Society, 1986.

[28] K. S. Larsen. Relaxed Multi-way Trees With Group Updates. *Journal of Computer and System Sciences*, 66:657–670, 2003.

[29] K. S. Larsen and R. Fagerberg. B-trees with relaxed balance. In *IPPS '95: Proceedings of the 9th International Symposium on Parallel Processing*, pp. 196–202. IEEE Computer Society, Washington, DC, USA, 1995.

[30] K. S. Larsen, E. Soisalon-Soininen, and P. Widmayer. Relaxed Balance through Standard Rotations. In *WADS '97: Proceedings of the 5th International Workshop on Algorithms and Data Structures*, pp. 450–461. Springer-Verlag, London, UK, 1997.

[31] P. L. Lehman and S. B. Yao. Efficient Locking for Concurrent Operations on B-trees. *ACM Transactions on Database Systems (TODS)*, 6(4):650–670, 1981.

[32] T. Lilja. The Cost of Balancing in top-down (a, b) trees. In *AIT '05: Proceedings of the Conference on Algorithmic Information Theory in Vaasa, Finland*. 2005.

[33] R. Maelbrancke and H. Olivié. Optimizing Jan Jannink's Implementation of B$^+$-tree Deletion. *SIGMOD Record*, 24(3):5–7, 1995.

[34] L. Malmi and E. Soisalon-Soininen. Group Updates for Relaxed Height-balanced Trees. In *PODS '99: Proceedings of the Eighteenth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, pp. 358–367. ACM Press, New York, NY, USA, 1999.

[35] K. Mehlhorn. *Data Structures and Algorithms 1: Sorting and Searching*. Springer-Verlag, 1986.

[36] C. Mohan. An Efficient Method for Performing Record Deletions and Updates Using Index Scans. In *VLDB '02: Proceedings of the 28th International Conference on Very Large Data Bases*, pp. 940–949. VLDB, 2002.

[37] K. Oksanen. *Memory Reference Locality in Binary Search Trees*. Master's thesis, Helsinki University of Technology, Finland, 1995.

[38] K. Pollari-Malmi. *Batch Updates and Concurrency Control in B-trees*. Ph.D. thesis, Helsinki University of Technology, Finland, 2002.

[39] J. Rao and K. A. Ross. Making B$^+$-trees Cache Conscious in Main Memory. *SIGMOD Record*, 29(2):475–486, 2000.

[40] Y. Sagiv. Concurrent Operations on B*-trees with Overtaking. In *PODS '85: Proceedings of the Fourth ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, pp. 28–37. ACM Press, New York, NY, USA, 1985.

[41] R. Saikkonen. *Group Insertion in* AVL *Trees*. Master's thesis, Helsinki University of Technology, Finland, 2004.

[42] B. Schneier. *Applied Cryptography*. Wiley, second edition, 1996.

[43] R. E. Tarjan. Amortized Computational Complexity. *SIAM Journal of Algorithms and Discrete Methods*, 6(2), 1985.

[44] T. Ylönen. *An Algorithm for Full-Text Indexing*. Master's thesis, Helsinki University of Technology, Finland, 1992.

[45] C. Zou and B. Salzberg. On-line Reorganization of Sparsely-populated B$^+$-trees. *SIGMOD Record*, 25(2):115–124, 1996.