Universiteit Leiden
Opleiding Informatica

An algorithm for balancing

a binary search tree

Name: Ivo Muusse

Date: 29/01/2017

1st supervisor: Dr. H.J. Hoogeboom
2nd supervisor: Dr. R. van Vliet

BACHELOR THESIS

Leiden Institute of Advanced Computer Science (LIACS)
Leiden University
Niels Bohrweg 1
2333 CA Leiden
The Netherlands

An algorithm for balancing a binary search tree

Ivo Muusse

Abstract

The binary search tree is a well known and widely used data structure in computer science. It is the most used data structure for data storage and retrieval. Therefore it is important to keep researching in this area and hopefully find new properties and applications. In this thesis we present an algorithm to globally balance a binary search tree in a more dynamic way so that the tree stays valid during the process. It does this by repeatedly replacing each node by their underlying median node. The algorithm will be discussed and analyzed whereafter it will be compared with other algorithms.

Acknowledgements

There are many people to be thankful to. First of all my friends, who were writing their own thesis as well. We sat together in the computer rooms and provided each other with helpful knowledge in times of need. Secondly the student association: De Leidsche Flesch. When working on the thesis they provided us with the much needed coffee to keep us focused. I also want to thank my family for the extra morale they gave me. And lastly, many thanks to my supervisors, who were always there for me and gave me many insights.

Contents

List of Algorithms

Chapter 1

Introduction

A tree is a widely used data structure in computer science and is described in [Win60]. Its derivative the binary search tree [Hib62] is perfect for storing and retrieving data. This binary search tree is built from nodes, which are data structures holding a key and two references that can point to other nodes. Keys are items such as a number or a name which can be linearly ordered. The first reference is known as the left child and has to point to a node with a smaller key. The second reference is known as the right child and has to point to a node with a larger key. An example of a binary search tree is shown in Figure 1.1.

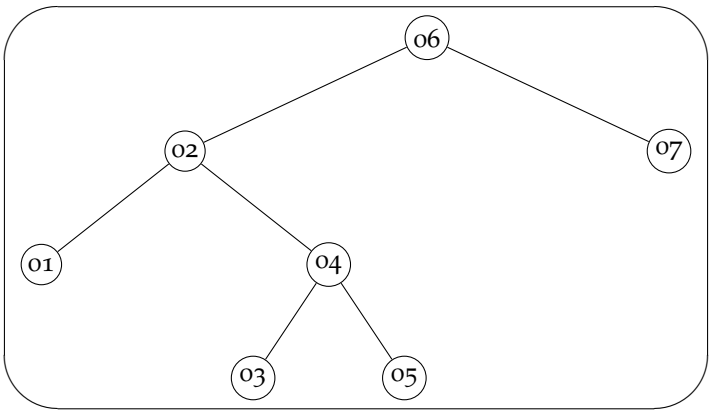


Figure 1.1: A binary search tree with integers as keys

This reference based construction is dynamic and therefore only reserves the memory it needs. The structure is suited for fast lookup, insertion and removal of data. With this in mind it is no surprise that the binary search tree is used in many search applications such as the map and the set data structures. The only problem is that a fast lookup time is not always guaranteed. It is possible that by inserting and removing nodes the tree will end up as a list leaving a worst case search complexity of $\mathcal{O}(N)$. To prevent this it is possible to rearrange the nodes in the tree so that for every node the minimum and maximum height of all routes in its left and right subtrees do not differ more than one. This results in a balanced tree with a worst case search

complexity of $\mathcal{O}(\log N)$. An example of a balanced binary search tree is given in Figure 1.2.

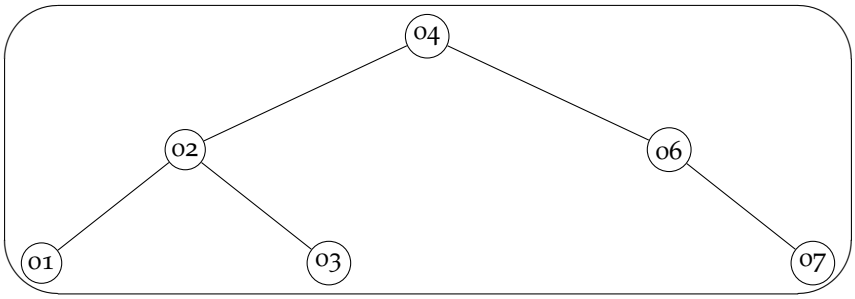


Figure 1.2: A balanced binary search tree

1.1 Problem

Since balanced trees are desired for fast searching, inserting and removal of data, there is already a lot of research done in finding algorithms which can convert unbalanced binary search trees to balanced ones. However, most current strategies for balance a unbalanced binary search tree completely demolish the tree before rebuilding it in balanced form. During the process the tree is in an invalid state and can't be used for any executions. This is not practical when trees are very large, because then it takes a long time to do the rebalancing. And then there is that some trees might be less static. In their dynamic appearance they can get many insertions and deletions between the reads. Some algorithms balance trees after each step with would be more useful if they did this when all insertions and deletions were done. In this thesis we will investigate how the rebalancing of nodes is done and propose a new algorithm where the tree can be used during the balancing and with only balances the tree at once.

1.2 Thesis Overview

The thesis is organised as follows. In Chapter 2 definitions are given to understand the rules of balancing. Chapter 3 shows an overview of existing algorithms followed by a proposition of the new algorithm in Chapter 4. We evaluate this algorithm in Chapter 5 and see how it compares to existing ones in Chapter 6. Chapter 7 then concludes.

Chapter 2

Definitions

Before we can focus on the rebalancing of a binary search tree, it should first be clear what balancing is and how it works. Therefore we first need to know what conditions need to be true to make a tree balanced. There are two kinds of balanced states. The first one is height-balanced, where the search time is limited to $\mathcal{O}(\log N)$. This tree is useful when all nodes are equally important so that fast look-up time is guaranteed. Sometimes a selection of nodes may be more important than others. And that is where a weight-balanced tree [NR73] comes in. The more important the node the higher the weight, so that those nodes are expected to be found closer to the root since the difference of weights in the left and right tree remain within some factor α of each other [HY11]. In our thesis we our focus is on height-balancing.

We know from Knuth [Knu71] that if no route in a tree node differs more than one in length from any other route or when the tree is empty the tree is height-balanced. For our thesis we the tree to be more strict and make therefor use of another interpretation, as is formulated in Definition 2.9. But to get there we first define other definitions.

Definition 2.1. A node is *height-balanced* if the maximum route in the left and right subtree do not differ more than one in length.

Definition 2.2. A binary search tree is *height-balanced* if

1. the tree is empty, or
2. all its nodes are height-balanced.

An example of a height-balanced tree can be found in Figure 2.1. For every node in the tree the maximum path length of the left and right subtree do not differ more than one. For example in node *o4* the path length is two in the left subtree and three in the right subtree.

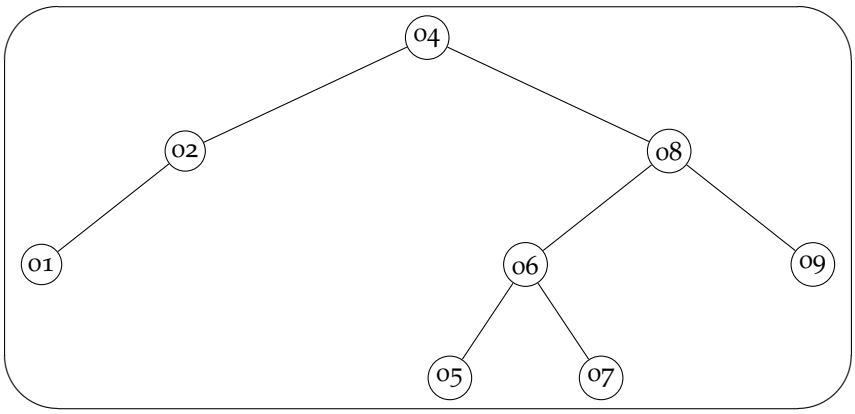


Figure 2.1: A height-balanced binary search tree

Definition 2.3. A node is *strictly height-balanced* if the maximum and minimum route in both left and right subtree do not differ more than one in length.

Definition 2.4. A binary search tree is *strictly height-balanced* if

1. the tree is empty, or
2. all its nodes are strictly height-balanced.

Definition 2.5. If a binary search tree is strictly height-balanced it is also height-balanced.

A strictly height-balanced tree is more precisely. Only the lowest level in the tree can be incomplete. It is therefor automatically height-balanced. The minimum-height stands in this case for the maximum-height in the second subtree. An example can be found in Figure 2.2. Now in node *04* the maximum path length of its subtrees is three and its minimum is two. This is different than in the example of height-balanced in Figure 2.1 where the lowest two levels were incomplete.

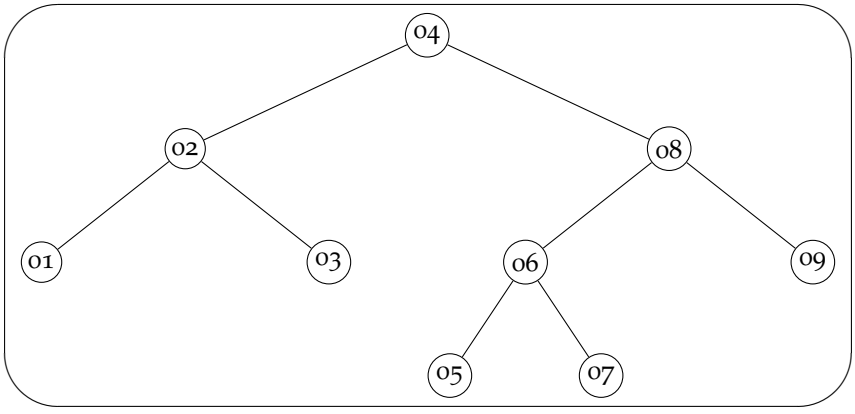


Figure 2.2: A strictly height-balanced binary search tree

An even more precise way of balancing is making the tree median-balanced, as is formulated in Definition 2.6 and Definition 2.7. It is almost the same as strictly height-balanced but now for all nodes also the number of

nodes in the left and right subtree does not differ more than one.

Definition 2.6. A node in a binary search tree is a *median-balanced* if the number of nodes in the left and right subtree differ not more than one.

Definition 2.7. A binary search tree is *median-balanced* if

1. the tree is empty, or
2. all its nodes are median-balanced.

An example of a median-balanced tree can be found in Figure 2.3. This example differs from the two above by that all nodes are now median nodes. There can be multiple solutions since when we have an even number of nodes two nodes can are medians. For example we can put node *03* as right child of node *02* and node *04* then as right child of node *03*. This would still result in a median-balanced tree. The difference in this is that they can be left or right median-balanced.

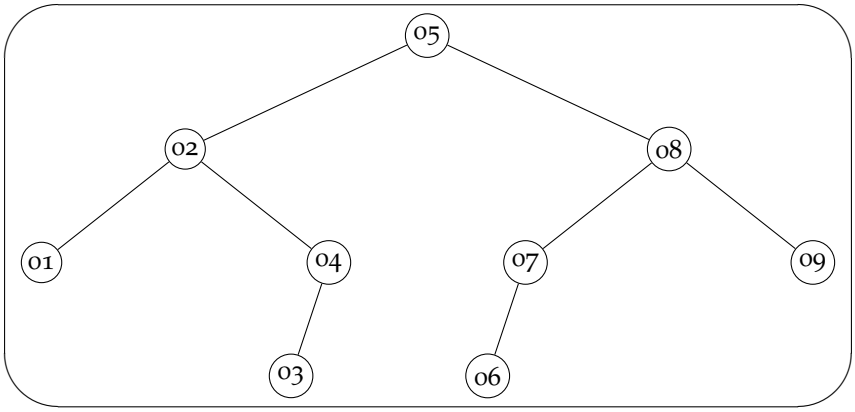


Figure 2.3: A median-balanced binary search tree

A median-balanced tree is also strictly height-balanced, because when the last two levels would be incomplete some nodes would not be median-balanced and therefor the tree would not be median-balanced.

Definition 2.8. If a binary search tree is median-balanced it is also strictly height-balanced.

In a strictly height-balanced tree some nodes might therefor be median-nodes as well. We can therefor redefine strictly height-balanced from Definition 2.4 to strictly height-balanced as is given Definition 2.9.

Definition 2.9. A binary search tree is *strictly height-balanced* if

1. the tree is empty, or
2. all its nodes are strictly height-balanced or median-balanced.

Chapter 3

Related Work

Already many people have proposed an algorithm to deliver a strictly height-balance or also said balanced binary search tree. There already exist all kind of strategies which can be very different from each other. In this chapter we discuss the differences between those algorithms.

3.1 Storing references in an array

The first group of algorithms globally balances the tree by first converting it into a sorted array where each element corresponds with a node in the tree. This can be easily done by an inorder traversal of the tree. The array is then used to keep connections to every node so that none of those references is lost when making adjustments to the nodes of the tree. By using this array the tree can then be easily rearranged to a balanced one.

3.1.1 Martin and Ness

One of the first to use this method were Martin and Ness [MN72]. They create an imaginary blueprint of how the tree is going to be. Then they fill this tree by making an inorder traversal where each time the smallest remaining node from the array by is put. This is done by the main recursive function *Grown*, which is shown in Algorithm 1. In this, n is the number of nodes and the function *Next* returns the smallest returning node. The children of this node are set to the outcome of both *Grown* calls in line 7 and 9. It then returns this node in line 10. The function *Grown* itself returns then this smallest node. This all together gives a complexity of $\mathcal{O}(N)$.

Algorithm 1 Martin and Ness

```
1: function GROW(n)
2:     if n = 0 then
3:         return null
4:     if n = 1 then
5:         return Next()
6:     if n>1 then
7:         Grow(⌊(n−1)/2⌋)
8:         Next()
9:         Grow(⌈(n−1)/2⌉)
10:        return node
```

3.1.2 Chang and Iyengar

Chang and Iyengar [CI84] differ from Martin and Ness by making the same imaginary blueprint of a tree but then fill this by making a preorder traversal instead of an inorder traversal where at each point the corresponding median node from the array is placed. Therefore the complexity is the same. This algorithm is shown in Algorithm 2. The variables *low* and *high*, respectively the lower and upper bound, are used to do the tree traversal. The first time the function *grow* is called the variable *low* starts at 1 and *high* gets to be the same as the number of nodes in tree. In the function *next* the median node between the corresponding bounds is placed. It then returns this median in line 11.

Algorithm 2 Chang and Iyengar

```
1: function GROW(low, high)
2:     if low>high then
3:         return null
4:     if low = high then
5:         return Next()
6:     if low<high then
7:         mid = ⌊(low+high)/2⌋
8:         Next()
9:         Grow(low, mid − 1)
10:        Grow(mid + 1, high)
11:        return node
```

3.1.3 Moitra and Iyengar

Moitra and Iyengar [MI86] also found a way to rebuild the tree from a sorted array. Their solution is more suitable for parallelization. For each node in the array they found a way to determine the children of the node. This can be computed in series by one single core or separately where for each node a different core can be used. This solution is shown in Algorithm 3. The number of nodes N is set to $N = 2^n - 1$ where$n$ is the height of the tree. The *lson* and *rson* are array's which hold pointers to the left and right children and the *link* holds pointers to the nodes itself. Note that this only works for trees where in balanced state each level is filled. The time complexity without extra cores is $\mathcal{O}(N)$.

Algorithm 3 Moitra and Iyengar

```
1: function C-Level-Struct
2:     for i ← 1, n do
3:         for j ← 1, 2^(i−1) do
4:             K ← 2^(n−i) + (j − 1) ∗ 2^(n−i+1)
5:             if odd(k) then
6:                 lson(link(k)) ← null
7:                 rson(link(k)) ← null
8:             else if even(k) then
9:                 K1 ← 2^(n−i−1) + (2 ∗ j − 2) ∗ 2^(n−i)
10:                K2 ← 2^(n−i−1) + (2 ∗ j − 1) ∗ 2^(n−i)
11:                lson(link(k)) ← link(k1)
12:                rson(link(k)) ← link(k2)
```

3.1.4 Haq, Cheng and Iyengar

Haq, Cheng and Iyengar [HCI88] later improved the algorithm from Moitra and Iyengar by making it useful for balancing any arbitrary binary search tree where the number of nodes is not only $N = 2^n - 1$ but can also be $N < 2^n - 1$. They conclude that with N cores the complexity is brought back to $\mathcal{O}(1)$.

3.1.5 Muthusdari and Suresh

It is also possible to do this in a completely different way, like Musthusdari and Suresh [MS14] did. They use an array constructed from a preorder traversal. This array is then converted to a second array so that when this is inserted in an empty tree, with at most one left or right rotation afterwards, a balanced tree is created. The second array is made by first taking all the even positions in groups of 3. Then reverse the elements in these groups and put them in the second array. After this, the same is done for all the odd positions. Then if the number of nodes is larger than 14 one rotation is done to make the tree balanced.

3.2 Rearranging the tree by making adjustments

A second strategy is to rearrange the binary search tree by making adjustments in the tree itself. The tree then undergoes multiple transformations to convert the tree to a balanced state. This can be left or right rotations but also splitting and rejoining parts of the tree. To do this no secondary array is needed.

3.2.1 Bentley

Bentley's solution [Ben75] is to rearrange the nodes so that all nodes are median nodes. It is then guaranteed that the tree is balanced. He does this by finding the median which are then set as the root. All nodes smaller become its left subtree and all nodes greater become its right subtree. Then the same is done recursively for both children. This solution is shown in Algorithm 4. The variable A stands for the tree which is then split in three trees: P, *greater*, and *less*. The complexity of this algorithm is $\mathcal{O}(N \log(N))$.

Algorithm 4 Bentley

```
1: function OPTIMIZE(A)
2:     if A = null then
3:         return null
4:     P ← median(A)
5:     split(A, P, less, greater)
6:     join(A Optimize(less), Optimize(greater))
7:     return P
```

3.2.2 Day

Day [Day76] on the other hand converts the tree first to a backbone. Then he rotates the tree in balanced form by another series of rotations. The rotations are self-determined and therefore independent from the structure of the original tree. All rotations are done in constant time and therefore the complexity of the algorithm is $\mathcal{O}(N)$. The only restriction is that the tree has to be threaded so that the successor and predecessor nodes can be efficiently found.

3.2.3 Stout and Warren

Stout and Warren [SW86] later improved Day's algorithm by using the advantages of the binary search tree. Because the tree doesn't have to be threaded anymore, both phases can be simplified. In the first phase only right rotations are done to make a backbone. Then in the second phase, also show in Algorithm 5, the backbone is rotated back with a constant series of rotation to a balanced tree. The second phase is divided into two parts: Vine-to-tree and Compress. Vine-to-tree computes all places where series of rotations which have to be executed. Then Compress does this by doing left rotations on the backbone or what is left of it. In the first series in line 3 all leaves are prepared and set in order to ensure that the tree will be perfectly balanced. Then as long as the remaining size of the backbone is larger than 1, new series of rotations are done to divide all routes till the tree is completely balanced.

Algorithm 5 Stout and Warren

```
1:  function VINE-TO-TREE(root, size)
2:      leaves ← size + 1 − 2 ∗ log_2(size + 1)
3:      Compress(root, leaves)
4:      size ← size − leaves
5:      while size>1 do
6:          Compress(root, size/2)
7:          size ← size/2
8:
9:  function COMPRESS(root, count)
10:     node ← root
11:     for i ← 1, count do
12:         child ← rson(node)
13:         rson(node) ← rson(child)
14:         node ← rson(node)
15:         rson(child) ← lson(node)
16:         lson(node) ← child
```

3.3 Rebalance after every modification

There also exist ways to balance the tree during insertion and deletion. The costs for these algorithms are higher, but they guarantee a height-balanced tree at any moment. It is then not necessary to rebalance the tree for reading. Note that this is does not guarantee a strictly balanced tree.

3.3.1 Adel'son-Vel'skii and Landis

An example of this is the algorithm of Adel'son-Vel'skii and Landis [AVL62]. They modified the tree so that to every node a balance factor is added. The factor stands for the difference in maximum height of the two subtrees. These factors are set correctly after each inserion and deletion. When in a node the factor in its left subtree differs more than one with the factor in its right subtree or visa versa, rebalancing is done to restore this property of the tree. Rebalancing can be done by one or more tree rotations.

Chapter 4

Proposition of an algorithm

In this chapter we propose an algorithm to globally balance a tree. The algorithm is based on Bentley's solution [Ben75]. His idea to rebalance the tree is to rearrange the nodes so that all nodes are median nodes. We do the same if we find a node with is not strictly height-balanced by finding the underlying median for each subtree, which we will then move upwards to the root of the subtree. Bringing nodes upwards is a common technique, which is mainly used in Sleator and Tarjan's splay tree [ST85]. We replace the rotation technique with the Stephenson's split and join technique [Ste80] used to do root insertions. Our most important modifications are that add we add the number of nodes to every node so that we can use this number instead of keys to find the median node and that we now split the tree on the median. The algorithm is described by both examples and pseudo code.

4.1 Preliminaries

In the tree used for testing our algorithm each node contains a lson, rson, weight and a data element. Lson and rson are pointers to its left and right subtree. Data is the element where information about the node is stored. It contains the key value. The weight is the number of nodes of the subtree at a node. Notice that the weight element is not common in a binary search tree and is specially added for our algorithm. Therefore in our binary search tree functions like insert and delete are modified to keep the weights valid.

4.2 Median-balancing

We know from the definitions in Chapter 2 that a tree is strictly height-balanced if all nodes are strictly height-balanced or median-balanced. Our solution is to check for each node if it is height-balanced and then

if this is not the case we will make this node median-balanced. We do this by replacing the node by its underlying median. We name the algorithm **Algorithm A** and explain it by two parts.

4.2.1 Split and join

The first part of Algorithm A, shown in Algorithm 6, is the balance step. It replaces the root by its underlying median. The algorithm does this transformation by splitting the tree in the middle, resulting in the median, a left tree with all elements with a smaller value and a right tree with all elements with a larger value. It joins these parts together into a new tree with the median as root, the left tree as lson and the right tree as rson.

Algorithm 6 Algorithm A: Split and join

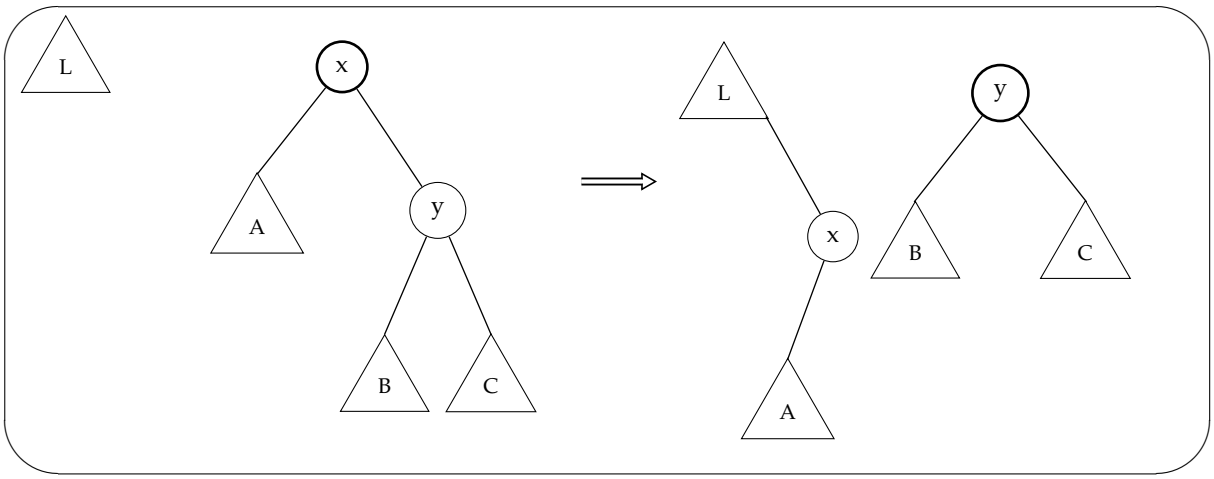
```
1:  function REBALANCE(root)
2:      ltree, lsubtree, rtree, rsubtree ← null
3:      if weight(lson(root)) < weight(rson(root)) then
4:          put_left ← weight(root) − 1 / 2
5:      else
6:          put_left ← weight(root) / 2
7:
8:      while weight(lson(root)) ≠ put_left do
9:          if weight(lson(root)) < put_left then
10:             if ltee = null then
11:                 ltree ← root
12:             else
13:                 rson(lsubtree) ← root
14:             lsubtree ← root
15:             weight(root) ← put_left
16:             put_left ← put_left−weight(lson(root)) − 1
17:             root ← rson(root)
18:         else
19:             if rtee = null then
20:                 rtree ← root
21:             else
22:                 lson(rsubtree) ← root
23:             rsubtree ← root
24:             weight(root) ← weight(root) - put_left − 1
25:             root ← lson(root)
26:     if ltree = null then
27:         ltree ← lson(root)
28:     else
29:         rson(lsubtree) ← lson(root)
30:     if rtree = null then
31:         rtree ← rson(root)
32:     else
33:         lson(rsubtree) ← rson(root)
34:     lson(root) ← ltree
35:     rson(root) ← rtree
36:     weight(root) ← weight(ltree) + weight(rtree) + 1
37:     return root
```

The algorithm starts by creating four pointers in line 2 named: *ltree*, *rtree*, *lsubtree* and *rsubtree*. *Ltree* and *rtree* will point to the root of the left and right tree, *lsubtree* will points to the node with the largest value in the left tree and *rsubtree* will points the node with the smallest value in the right tree. Then in line 3–6 the variable *put_left* is declared. This variable stands for the number of nodes to be added to the left tree. Since the left tree starts empty we have to determine its expected size. This size is the size of the tree divided by two. Then, when the size of the left tree is the same as the expected size, we stop. At this point, the current node is the median. We check in line 3 if our first step is to the left or right so that we will split on the closest median.

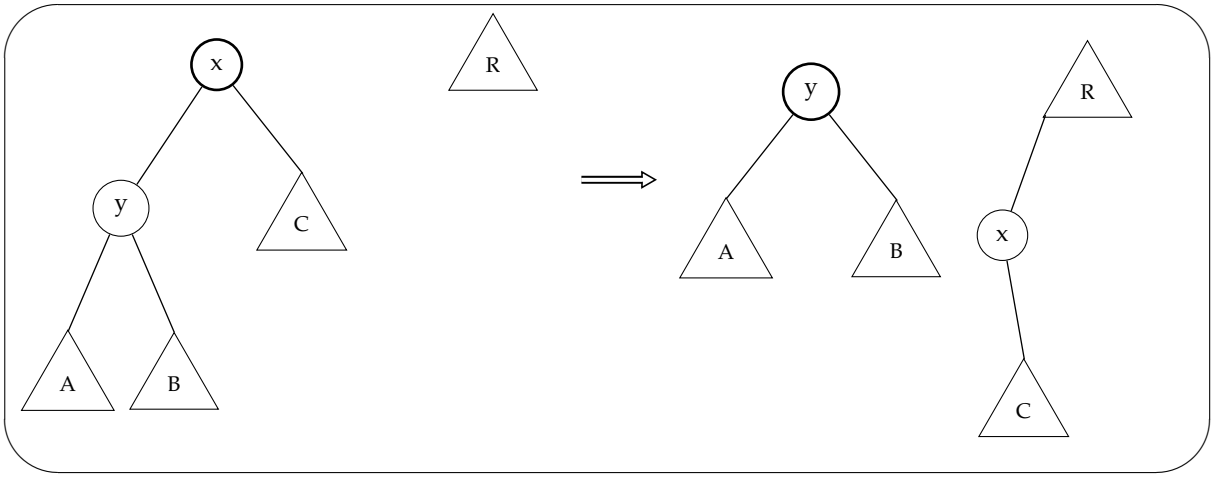
When everything is set up, we start splitting the tree in line 8–33. We do the splitting by walking through the tree until we find a node where the weight of the lson is the same as *put_left*, so that the selected node is the closest median. While walking, we add nodes to the left and right tree and adjust the weights of the nodes, so that they stay valid. There are three possibilities:

1. The weight of lson is less than *put_left* (line 10–17)
2. The weight of lson is greater than *put_left* (line 19–25)
3. The weight of lson is equal to *put_left* (line 26–33)

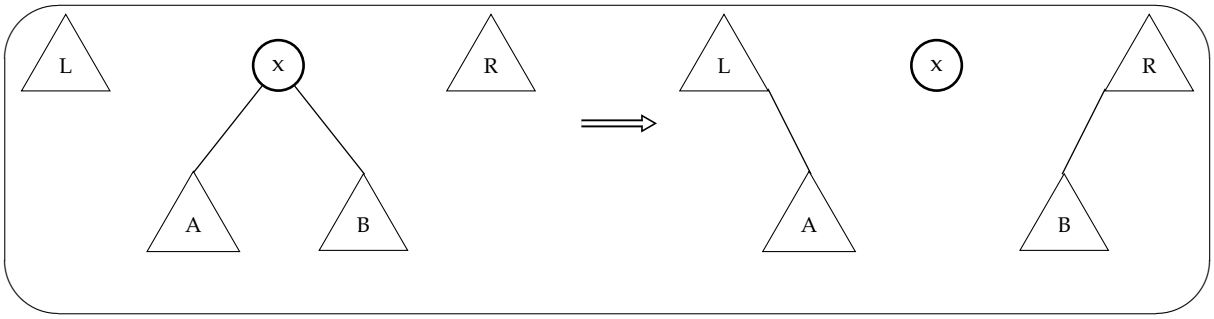
In option 1, we add the node to the rson of the greatest node in the left tree. Its weight is set to *put_left* since we know that exactly *put_left* will be added. We then decrease *put_left* by the number of nodes we added to the left tree and move to the rson. In option 2, the node is added to the left son of the smallest node in the right tree. We decrease the weight of the node by *put_left* plus one extra for the median. Since we don't add nodes to the left tree we don't adjust *put_left*. And lastly in option 3, there are enough nodes in lson to complete the left tree. This means that the current node is the median. Therefore we add its lson to the left tree and its right son to the right tree. These three steps of splitting are shown in Figure 4.1.



(a) The number of nodes in A is less than *put_left*



(b) The number of nodes in A is greater than *put_left*



(c) The number of nodes in A equals *put_left*

Figure 4.1: The three actions during the splitting phase. L and R are respectively the left and right subtree. Also in each figure the node with the bold edge is the current root of the tree

Once the median is found and the tree is correctly split, the trees are joined together in line 34–37. The median becomes the new root with the left tree as lson and the right tree as rson. The result of splitting and joining this way is the same as one obtained by repeatedly rotating the median with its parent until the median is the root but requires less pointer changes.

4.2.2 Recursive balance

The second part of Algorithm A is shown in Algorithm 7. In this part we make sure that every node in the tree is balanced. When a node is found for which this is not the case '*rebalance*' will be called to fix this.

Algorithm 7 Algorithm A: Solve

```
1: function SOLVE(root)
2:     if root = null then
3:         return root
4:     if balanced(weight(lson(root), weight(rson(root)) = false then
5:         root ← rebalance(root)
6:     lson(root) ← solve(lson(root))
7:     rson(root) ← solve(rson(root))
8:     return root
```

'*Solve*' executes Algorithm 6 if necessary for the root, followed by doing the same recursively for its lson and rson until these are *null*. Because of the preorder traversal, it first balances the node itself before it balances the children. Then at the end a balanced tree is guaranteed. The check if a node is strictly height-balanced takes place at line 4. The function '*balanced*' takes the number of nodes in the lson and rson as input and returns true if these are correctly divided between these subtrees. This function is shown in Algorithm 8.

Algorithm 8 Algorithm A: Balanced

```
1: function BALANCED(a, b)
2:     return ceil(log2(max(a, b) + 1)) ≤ floor(log2(min(a, b) + 1)) + 1
```

It does this by pretending that the subtree its rson and lson are already in balanced. Then by comparing the minimum and maximum height in line 2 of both subtrees it is checked that no two routes differ more than one in length. For the largest subtree the last level gets completed with *ceil* and the smallest subtree gets emptied at the last level with *floor*. By comparing these heights of both subtrees we can then determine if the difference is more than one or not. The complete C++ code of Algorithm A can be found at [Muu16].

4.3 Technique

Algorithm A uses a split and join technique to balance the tree. In every balance step the tree is torn open and glued back together. A example of splitting is shown in Figure 4.2. Visualizing this shows us what really is done. We start in Figure 4.2a searching for the median M. Then when found, we draw a dashed vertical line through this median. The tree is then split on this line resulting in two new trees with the L nodes as left tree and the R nodes as right tree to separate the median. The left tree contains the nodes of α_1, L1, α_2, L2, α_3 in symmetric order and a right tree with β_1, R1, β_2, R2, β_3 in symmetric order as well. Once M is then placed to the top and the left and right tree are added as children, a balance step is completed. The tree after the balance step is shown in Figure 4.2b.

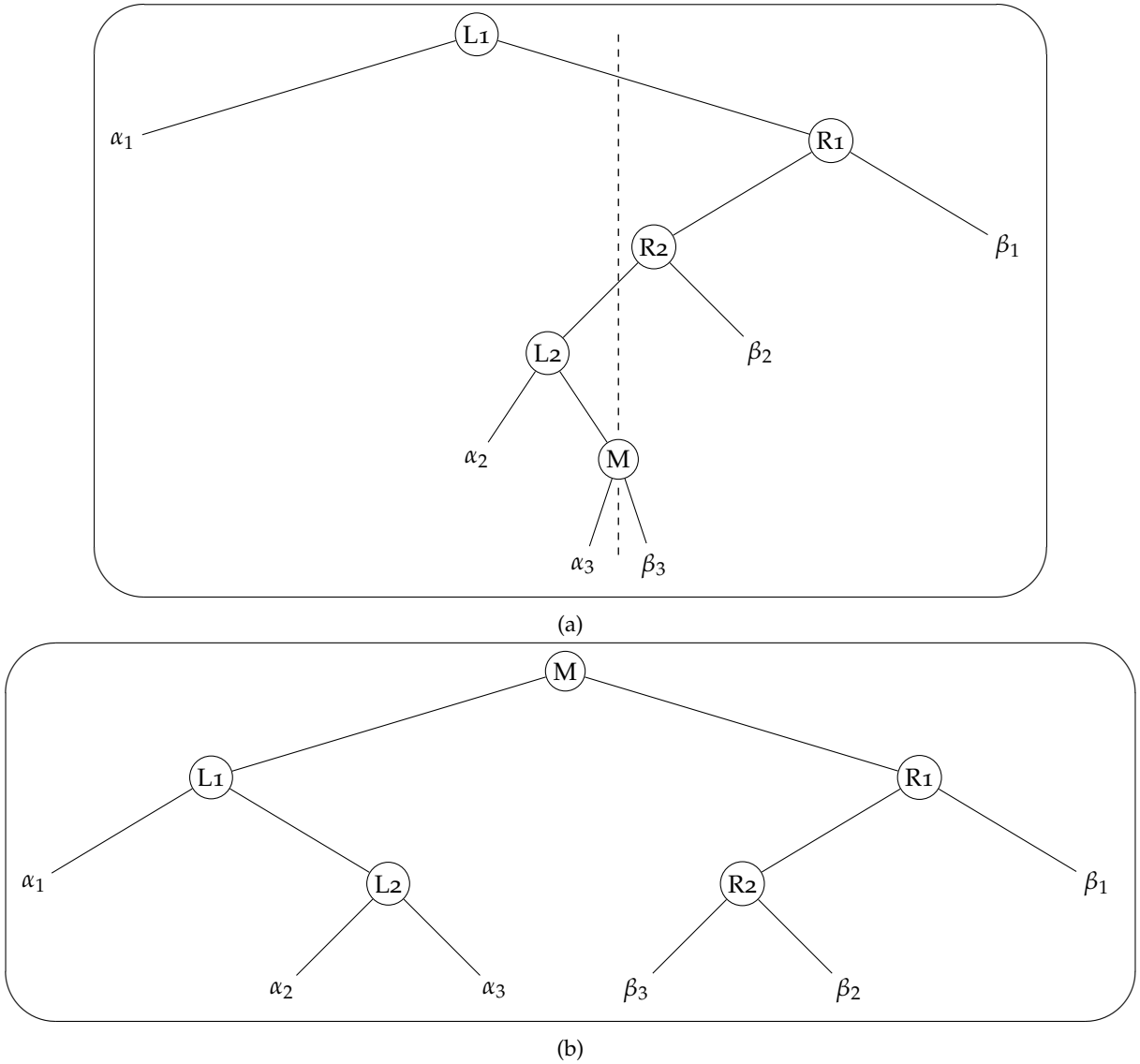
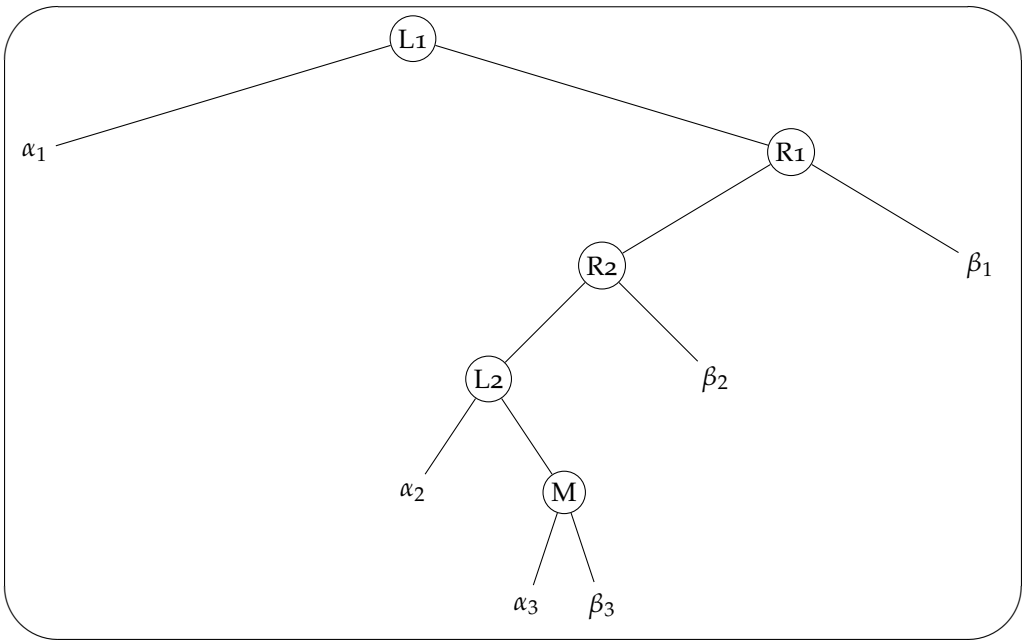


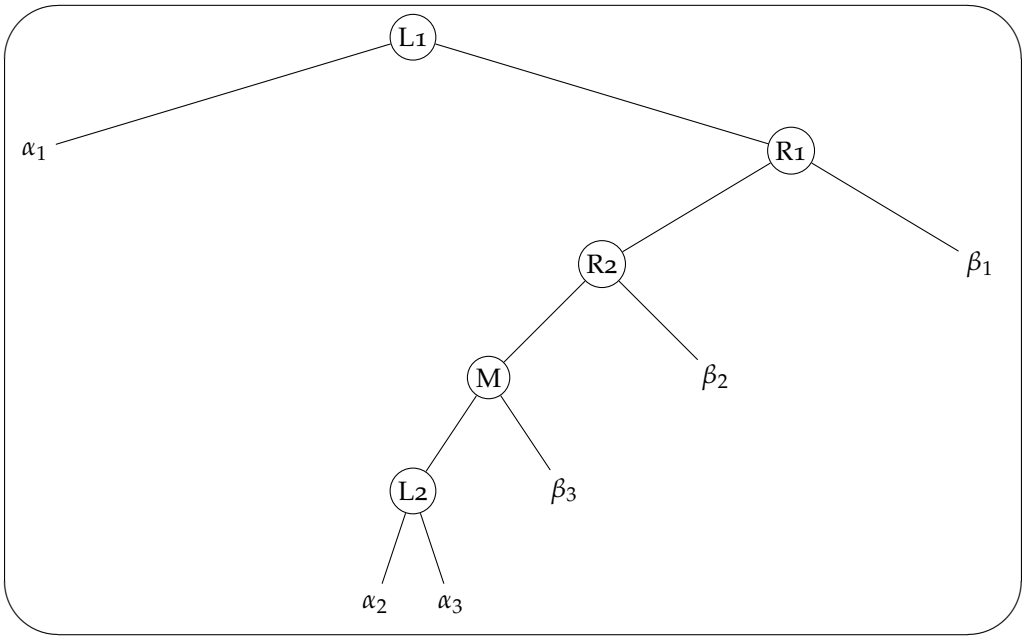
Figure 4.2: M is brought to the root by split and join

Another way to accomplish this is by bringing the median M up with only parent rotations. This is shown in Figure 4.3. Doing only parent rotations gives the same result but the costs are higher. For every rotation at least three pointers have to be adjusted while for split and join this is at most one per node with only the exception of the median node.

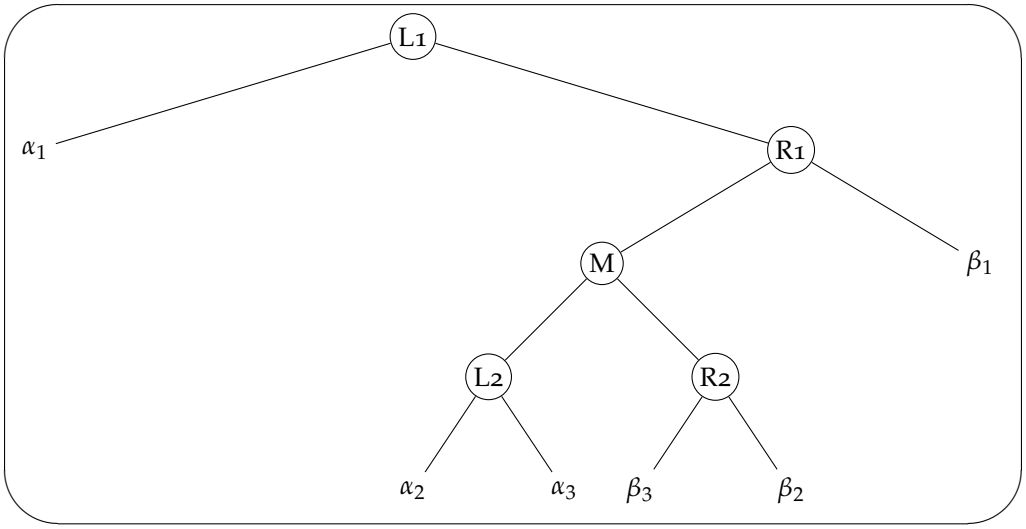
Whenever a node is rotated, its parent is added to one of its children. When this is a left rotation it is added to the left child and when this is a right rotation it is added to the right child. The difference is then that the left and right subtree are build bottom-up instead of the top-down. Note that doing only root rotations lead, to a different tree than one that would be obtained in a splay tree, because no Zig-Zig or Zag-Zag operation is executed or in other terms that no two levels are rotated together.


L1
α_1
R1
R2
β_1
L2
β_2
α_2
M
α_3
β_3

(a)


L1
α_1
R1
R2
β_1
M
β_2
L2
β_3
α_2
α_3

(b)


L1
α_1
R1
M
β_1
L2
R2
α_2
α_3
β_3
β_2

(c)

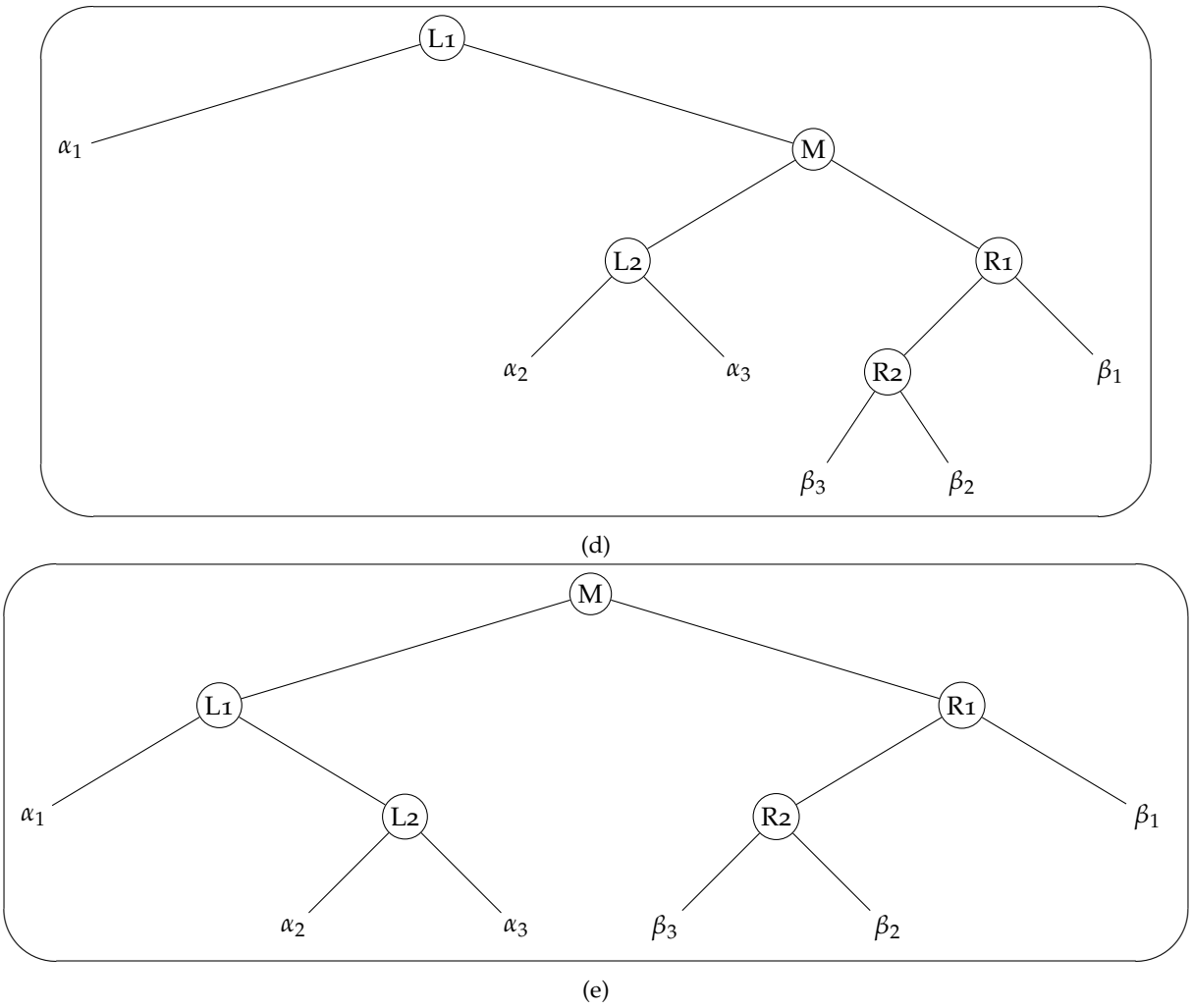


Figure 4.3: M is brought to the root by parent rotations

4.4 Examples

We demonstrate how this balancing is done by visualizing examples of balance steps. The same is done as in Figure 4.2, but now the nodes are filled with actual numbers so that we can simulate the process of balancing. There will be three examples. Each example concentrates itself on a particular part of the process so that all possible combinations of splitting will be shown.

In the first example, shown in Figure 4.4, the tree starts as a backbone consisting of 7 nodes in Figure 4.4a. Since root node 01 is in an unbalanced state we need to replace it with the median node 04 to follow our algorithm. This means that the tree will be split on node 04. To do this we draw the vertical dashed line which crosses on that node. All nodes left of the line will become the left tree and all nodes right of the line will become the right tree. The left and right tree are then added as children of the median to finalize the balance step. Then in Figure 4.4b we do the same for the children of node 04. In here the medians 02 and 06

are brought up to the top. Since we are in the second layer the dashed lines are not all the way to the root, but only to the root of the subtree we are looking at. This technique is repeated for all the children until the tree is completely balanced in Figure 4.4c. Remember that only a balance step is executed on a node when this node is in an unbalanced state. Shown in this first example are only right traversals.

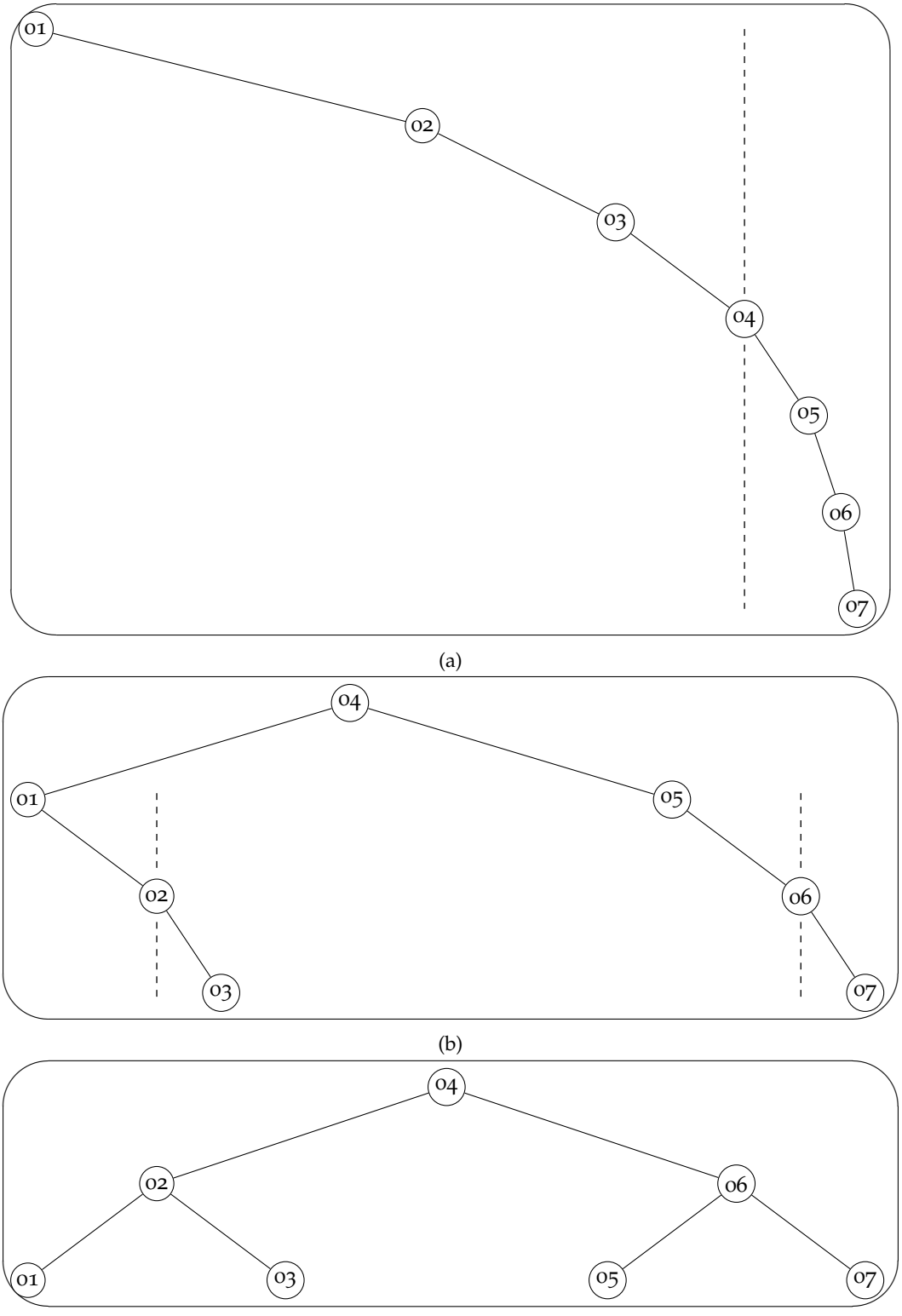


Figure 4.4: The transformation of a backbone into a balanced tree

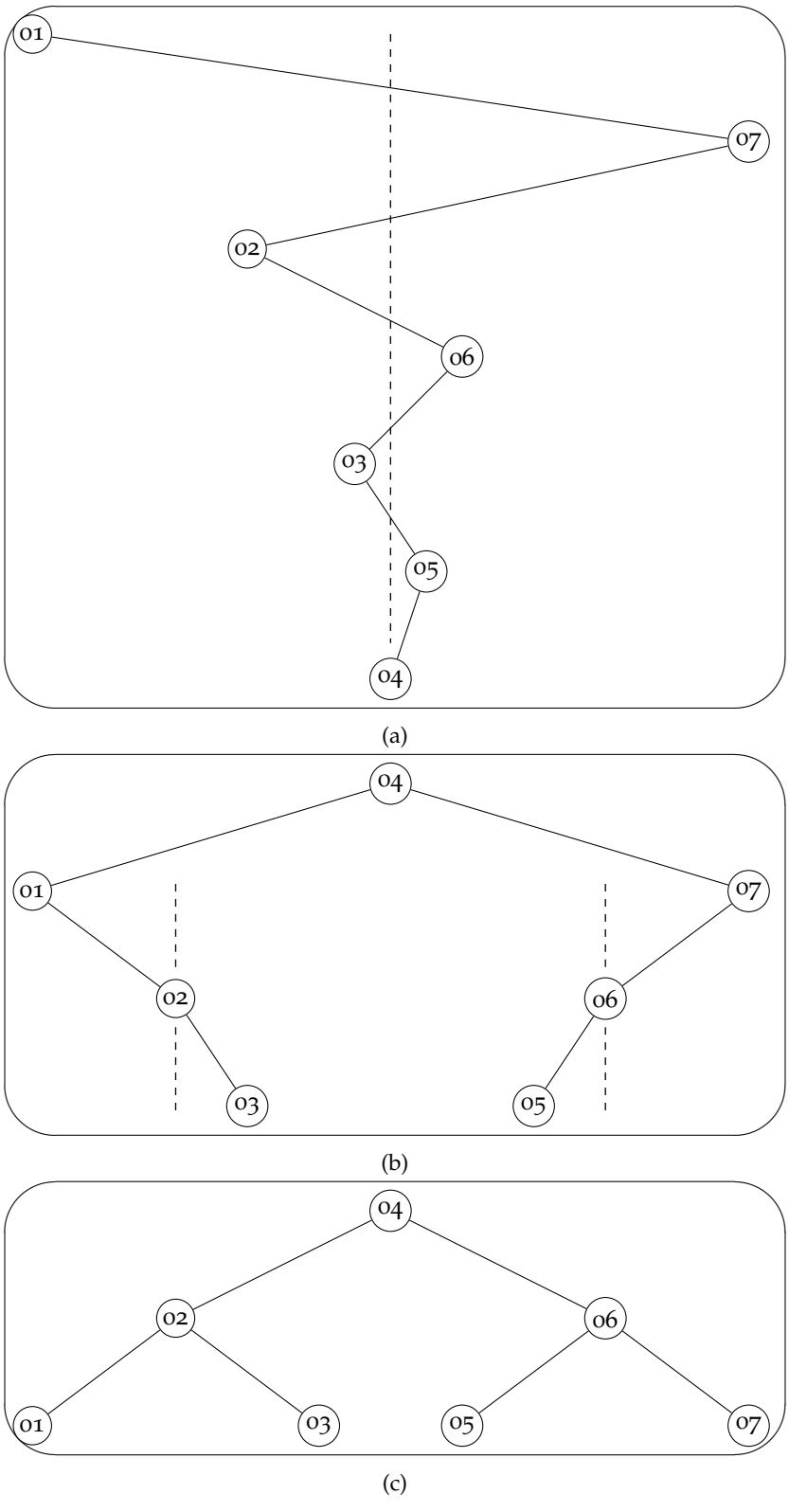
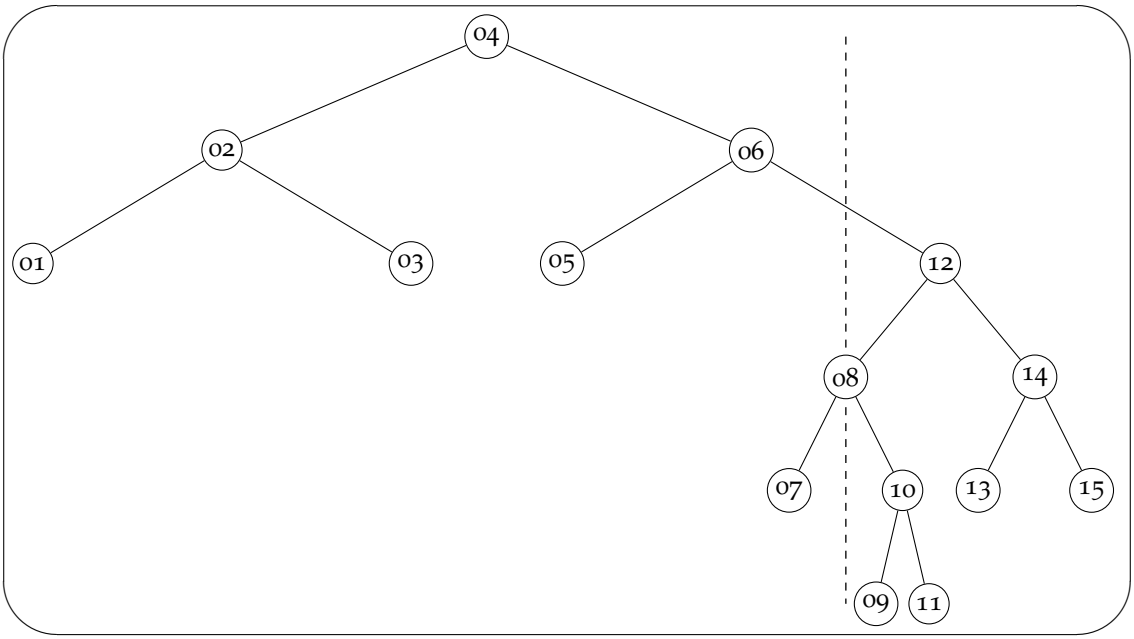


Figure 4.5: The transformation of a V formed tree into a balanced tree

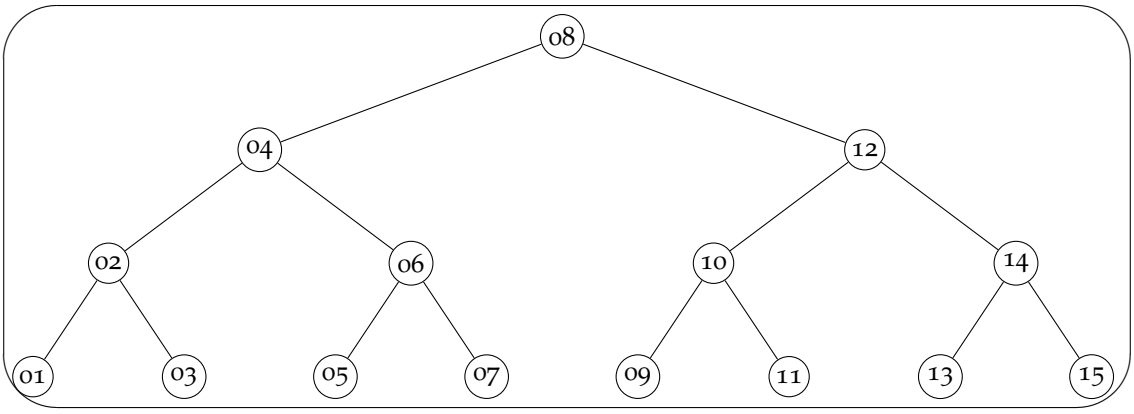
In the second example, which is shown in Figure 4.5, we do the opposite. The starting tree has the same number of nodes but now has the median as last node. To split this tree we need to do maximal possible number of moves in balance step. To do this we cut all edges which cross the dashed line. When we cut an

edge a node becomes disconnected, therefore we have to add it to the corresponding left or right tree. The result after one balance step is shown in Figure 4.5b. This example shows us right and left traversals from Figure 4.1 alternately by each other.

The last example is shown in Figure 4.6. This tree is larger than the previous trees and gives therefore a more general view of how trees are balanced. The tree consist of 15 nodes and starts in an unbalanced state. To balance this tree we need only one balance step. We start drawing the dashed vertical line on the median 08. To split here we cut all lines that cross the dashed line and connect the disconnected nodes to their corresponding tree. This means that node 07 is added to node 06 and node 10 is added to node 12 to finalize the right and left tree. Then after joining these trees together we get a balanced tree in Figure 4.6b. For the balance we made use of both left and right traversals.



(a)



(b)

Figure 4.6: The transformation of a unbalanced binary search tree into a balanced one

Chapter 5

Evaluation of the algorithm

In this chapter we show how strong the algorithm from Chapter 4 is by analysing its complexity in both space and time. We examine and describe the best, worst and average cases by giving formulas and examples of corresponding trees. Also the algorithms advantages and disadvantages will be discussed to reveal its characteristics.

5.1 Complexity

5.1.1 Best-case

The best case of our solution is an already balanced tree. In each node is checked if balancing has to be done. This check is done by looking at the number of nodes in the rson and lson. In a balanced tree the minimum and maximum height of the all rsons and rsons do not differ more than one, so that no balancing step has to be done. Therefore all nodes are only watched once. The complexity of the best case is thus $\mathcal{O}(N)$.

5.1.2 Worst-case

In the worst-case, it is exactly the other way around. In this case, we have to do a balance step for each node in the tree. It is possible that the tree is a linked list with the median as bottom node in the tree whereby all nodes have to be checked. This makes the maximal possible cost of a balance step $\mathcal{O}(N)$. Then in the next level we get two times the cost of $\frac{N}{2}$. This continues to four times $\frac{N}{4}$ in the third level till $\log_2(N+1)$ times $\frac{N}{\log_2(N+1)}$ at the last level. This sums up each of the $\log_2(N+1)$ levels to N as is shown in Figure 5.1.

The tree in this figure represents thus the worst-case. The cost of rebalancing is therefore in the worst-case $\mathcal{O}(N \log(N))$.

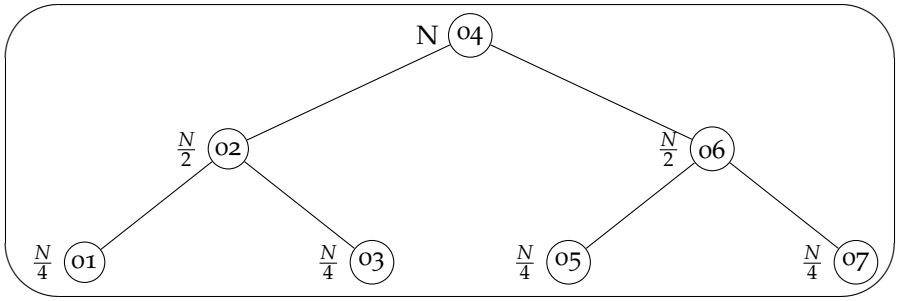


Figure 5.1: The number of nodes which have to be checked in a tree

However in practice, it is not possible to have such a worst-case situation where a balance step splits the tree in a root with two sub-trees where in each step the median is also the last node in the linked list. Every tree with the median at the bottom, results in new tree with a root with two backbones as left and right subtree after a balance step. Since the median is at position $\frac{N+1}{2}$ we should ask ourselves is the worst-case complexity is really $\mathcal{O}(N \log(N))$.

We can do this by proving that algorithm A is a member of $\Omega(N \log(N))$ by finding an example in this order of magnitude. If we take the tree from above where we try to balance a list with the median at the bottom, shown in Figure 5.2, we should get an example which should get close to the actual worst case because for every node a balance step has to be done. Since backbones will be recursively split and joined in smaller backbones we can find a recurrence relation which determines the number of node visits to balance a backbone. We then convert this recurrence relation, by prove of induction, to a formula and add the tree size to it. This formula can then be used to compute the total number of node visits to balance a binary search trees, which is in a list form with the median at the bottom.

So first we set up a recursive relation for balancing a backbone. In a complete tree we have N or $2^h - 1$ nodes where h is then the height of the tree. The median in a backbone can then be found at position $\frac{N+1}{2}$ or 2^{h-1}. After a balance step the tree is split and joined together in two smaller backbones with $h = h - 1$. The recursive relation is then as follows:

$$B(0) = 0$$
$$B(h) = 2 * B(h-1) + 2^{h-1}$$

To convert this recursive relation to a formula we need to adjust $B(h)$. We repeatedly replace $B(h-1)$ with itself until we find a pattern. This is shown with the symbol i. Then when the pattern is found this symbol is replaced so that $B(h-i)$ becomes $B(0)$ of which we know is zero, since an empty tree is already balanced.

$$
\begin{aligned}
B(h) &= 2 * B(h-1) + 2^{h-1} \\
&= 2 * (2 * B(h-2) + 2^{h-2}) + 2^{h-1} \\
&= 2 * (2 * (2 * B(h-3) + 2^{h-3}) + 2^{h-2}) + 2^{h-1} \\
&= 2^3 * B(h-3) + 3 * 2^{h-1} \\
&\vdots \\
&= 2^i * B(h-i) + i * 2^{h-1} \qquad (i = h) \\
&= 2^h * B(0) + h * 2^{h-1} \\
&= 2^h * 0 + h * 2^{h-1} \\
&= h * 2^{h-1}
\end{aligned}
$$

$B(h)$ gives the sum of the natural numbers less than or equal to number h. We show that the formula $h * 2^{h-1}$ is true for each natural number h by proving the basis and inductive step.

Basis: The statement holds for $h = 0$.

$$
\begin{aligned}
B(0) &= 0 * 2^{0-1} \\
&= 0
\end{aligned}
$$

Inductive step: $h * 2^{h-1}$ is derivable from $B(h)$.

$$
\begin{aligned}
B(h) &= 2 * B(h-1) + 2^{h-1} \\
&= 2 * (h-1) * 2^{h-2} + 2^{h-1} \qquad (\textit{induction hypothesis}) \\
&= (h-1) * 2^{h-1} + 2^{h-1} \\
&= h * 2^{h-1}
\end{aligned}
$$

Now that we know and have proved what the amount of work is to balance a backbone the only thing that remains is adding the number of nodes in the tree to this formula. This is because in our example we started with a list in which the median is at the bottom so that all nodes have to be visited to do a balance step. In a complete tree this is exactly $2^h - 1$ nodes. The formula for the two resulting backbones are then filled with $h - 1$, since the tree is split in half.

$$\begin{aligned} W(h) &= 2 * (h-1) * 2^{h-2} + 2^h - 1 \\ &= (h-1) * 2^{h-1} + 2 * 2^{h-1} - 1 \\ &= (h+1) * 2^{h-1} - 1 \end{aligned}$$

Replacing h with $\log_2(N+1)$ gives us our complexity in N. This $\frac{(N+1)*(\log_2(N+1)+1)}{2} - 1$ is member of $\Omega(N\log(N))$. Since we have found an upper bound in $\mathcal{O}(N\log(N))$ and a lower bound in $\Omega(N\log(N))$, we now know that the tree in worst case is solved in $\Theta(N\log(N))$. Such a tree can be built by first inserting $\lfloor\frac{N-1}{2}\rfloor$ elements in ascending order followed by all other $\lfloor\frac{N+2}{2}\rfloor$ nodes in descending order. This tree can be found in Figure 5.2.

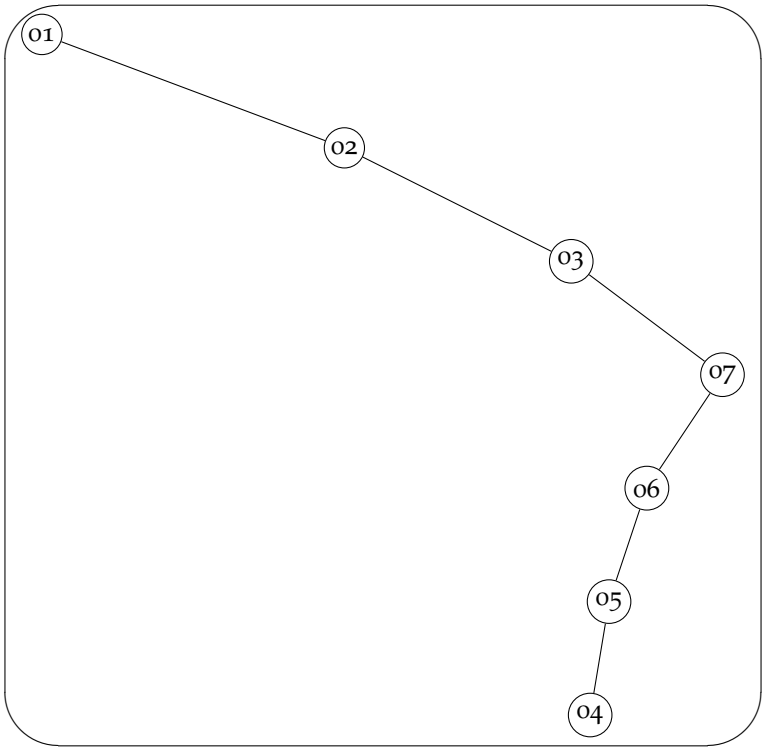


Figure 5.2: A worst case complexity situation

5.1.3 Average-case

A much more accurate measurement of the algorithm's performance is its average-case. The occurrence of a worst-case tree is rare in random generated balanced tree and can therefore be misleading. An experiment is done to determine the complexity of the average-case. For all trees with N between 1 and 1100, we generate 1000 random. We choice this range so that all vertexes all clearly visible. Generating these trees and balancing them takes around one minute on our test computer. The hardware specifications of this computer can be

found in Appendix A. For each tree size we determine the average number of node visits needed to balance the tree and divide them by the size of the tree to see what its complexity is. The result are shown in Figure 5.3

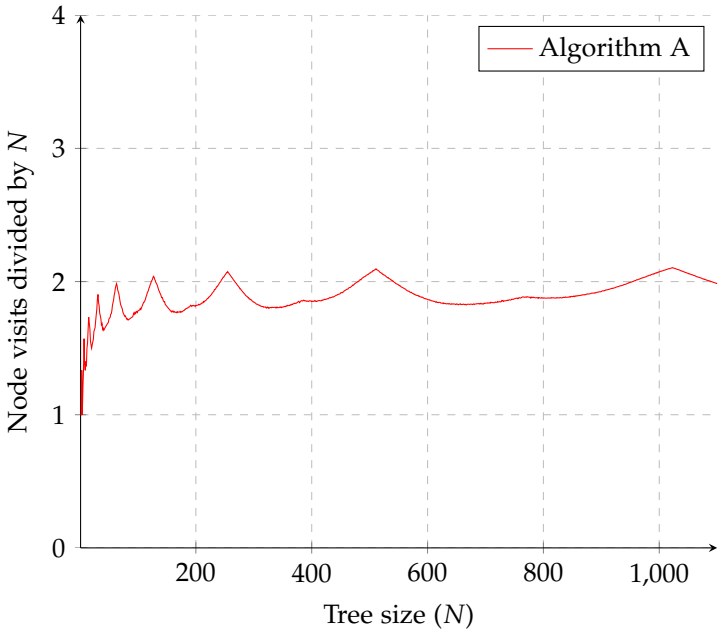


Figure 5.3: Average node visits to balance a random binary search tree by Algorithm A

It seems like the complexity is bounded by a horizontal asymptote just above the $2N$. From this it is almost possible to assume that the complexity of the average case is $\mathcal{O}(N)$ which is way better than our earlier found complexity of $\mathcal{O}(N \log(N))$. An explanation for the indentations might be that the more imperfect the tree, the more nodes can be the used as root. The chance that a node is than already balanced is greater.

When we try to determine the average number of node visits for a random tree, we can look to all possible permutations of the insertion order. The same tree may have multiple insertion orders. This means that some trees have a higher chance to occur when it is randomly generated. By analysing the chances of the different trees we should be able to determine the average node visits in the same way Knuth [Knu73] did for the average height of a random generated tree.

5.1.4 Space

Besides the fixed amount of memory which is required in our algorithm, we need to add N extra integers for the weights of the nodes. These can be kept inside the nodes as is done in our algorithm or stored in a separate array.

5.2 Advantages

An unique advantage of this algorithm is that the user can pause or stop the balancing process to do some searching, inserting or deleting. It is guaranteed that the tree is still in valid state without an increase in height. Nowadays trees are so large that balancing may take hours or even days. In this algorithm it is possible to do some look-up or adjustments between the balancing without waiting until it is finished or making a copy of the tree.

It is also possible to do some parallelization. When the algorithm splits the tree in a left and right tree, it can immediately start rebalancing those trees in parallel. The weights are already set correctly and thus already hold the value of its expected weight. It can be assumed that the left and right tree don't change and only grow. When we directly rebalance all the left and right trees when created, we only have to pay attention that sometimes the left and right tree are not yet complete and we sometimes have to wait for the next element. This is most the height of a balanced tree. When we add this to the maximum possible height we receive a complexity of $\mathcal{O}(N)$. This is only the case when we have N cores at our disposal so that every left and right is can be balanced at the same time. With a lower number of cores we still speed up the process but stay in the complexity of $\mathcal{O}(N \log(N))$.

Lastly, the algorithm can both be used for height or weight balancing. In our algorithm the weight for all nodes is 1 so that the tree is height and weight balanced. However, it is possible that some nodes are more important and should therefore have a greater weight.

5.3 Disadvantages

The main disadvantage is the time complexity. There exist algorithms that balance a tree in $\mathcal{O}(N)$ time which is faster than our $\mathcal{O}(N \log(N))$. Also, despite the fact that our extra space complexity is limited, it is not necessary for balancing a tree.

Chapter 6

Comparison with other algorithms

In this chapter we show how the algorithm from Chapter 4 behaves between the already existing solutions. Pushpa and Vinod [PV07] preceded us by giving a general comparison between the best known algorithms. They also name the advantages and disadvantages. These algorithms can be found in Chapter 3 under the names: Algorithm 1, 3 and 5. To elaborate and extend their work we will use some of the same algorithms to compare them with our found algorithm. The algorithms the ones from Martin and Ness [MN72] and Stout and Warren [SW86]. Both strategies globally balance a binary search tree in $\mathcal{O}(N)$ time, which is faster than our worst-case. But is it possible that we can still compete with those algorithms with a particular set of binary search trees?

6.1 Measurements

With our tests we want to compare the amount of work which is needed to balance a set of trees. We do two tests. In the first one we check how well the algorithms perform on just inserted trees. We assume that we have no prior knowledge and therefore use random trees. In the second test we used balanced trees and modified them by inserting and deleting nodes to simulate real use of trees where they are sometimes rebalanced.

All tests are done on trees which have exactly N nodes after balancing, with N between 1 and 1100 and can be found in [Muu16]. We choice this range so that all vertexes all clearly visible. We do every test 1000 times to get a good average. Generating these trees and balancing them takes around one minute on our test computer. The hardware specifications of this computer can be found in Appendix A. For the measurement to compare the performance we check the number of node visits. This is exactly $2N$ for the algorithm of Martin and Ness: N visits for the tree traversal to link an array to the tree and another N visits to build

the balanced tree from the array. For Stout and Warren, we count the nodes that were visited during the construction of the backbone and the nodes that were visited during the process of balancing this backbone. For algorithm A, this is the amount of nodes visited during all splits plus the amount of nodes where no split is done.

6.1.1 Balancing random trees

For simulating newly inserted trees, we use random trees. We assume that we don't know anything about the insertion order of the elements. Only the tree size is given. In Figure 6.1 the node visits of the three algorithms are shown per tree. Note that the plot of Algorithm A is the same as in Figure 5.3.

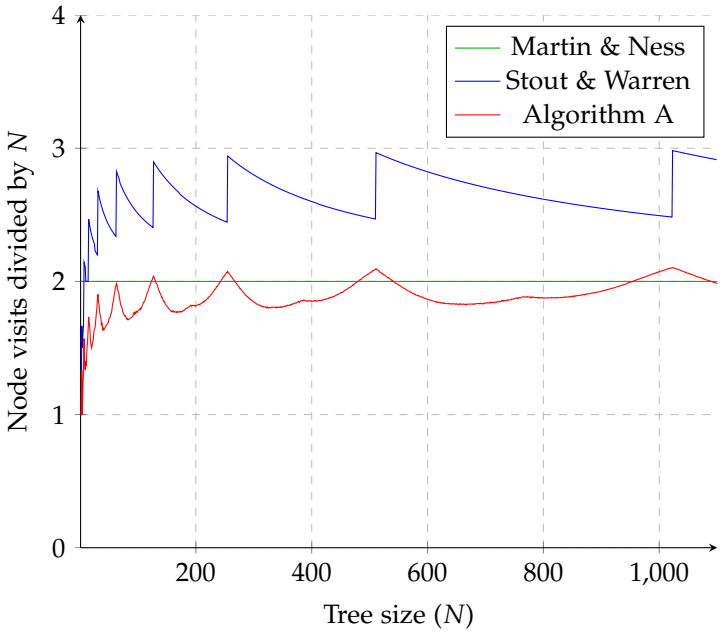


Figure 6.1: Average node visits to balance a random binary search tree

It can easily be observed that for Stout and Warren the line drops after every power of two. The explanation for this is that for every greater power of two the height of the tree increases resulting in the fact that an extra set of constant rotations has to be done to balance the tree. We also see that Algorithm A has the best average performance.

6.1.2 Modifications on a balanced trees

In practise, trees are not just random. It is possible that they are balanced once in a while. When there are not too many insertions and deletions done it keeps some of its balanced form. Since Algorithm A works better when a tree is more balanced we expect it to perform better than on random trees. For the other two

algorithms, we don't expect a significant difference because they always balance the full tree regardless of its shape. To test this, we create balanced trees on which modifications like insertion and deletion are done. Then, we analyse the average amount of node visits needed to balance those trees.

Insertions in a balanced tree

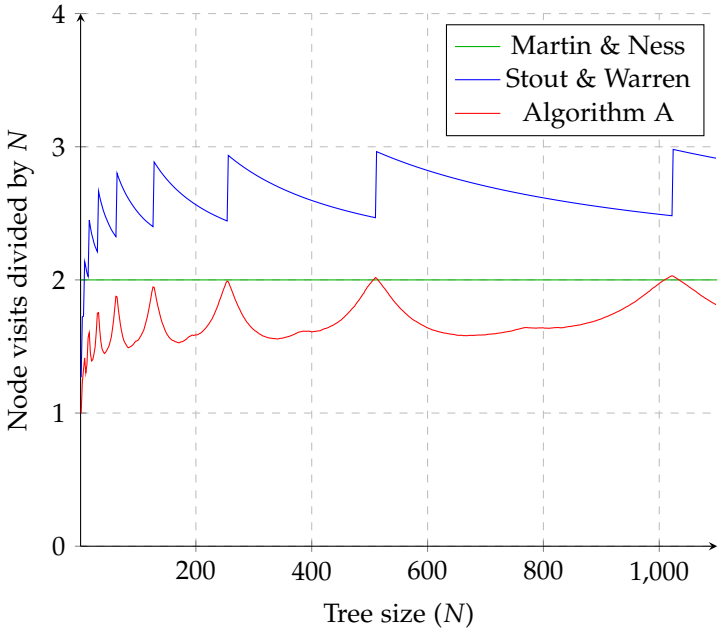


Figure 6.2: Average node visits to balance a balanced binary search tree of size $\frac{N}{2}$ where $\frac{N}{2}$ nodes are randomly inserted

Deletions in a balanced tree

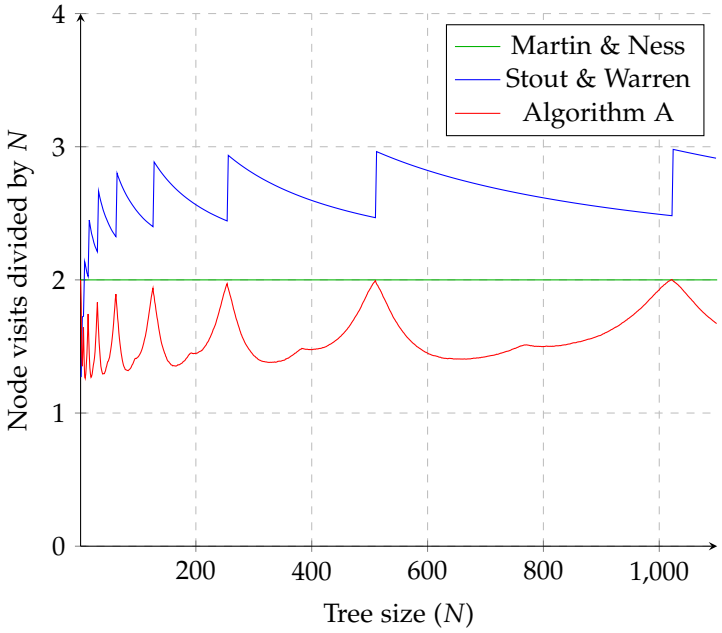


Figure 6.3: Average node visits to balance a balanced binary search tree of size $2N$ where N nodes are randomly deleted

Insertions and deletions in a balanced tree

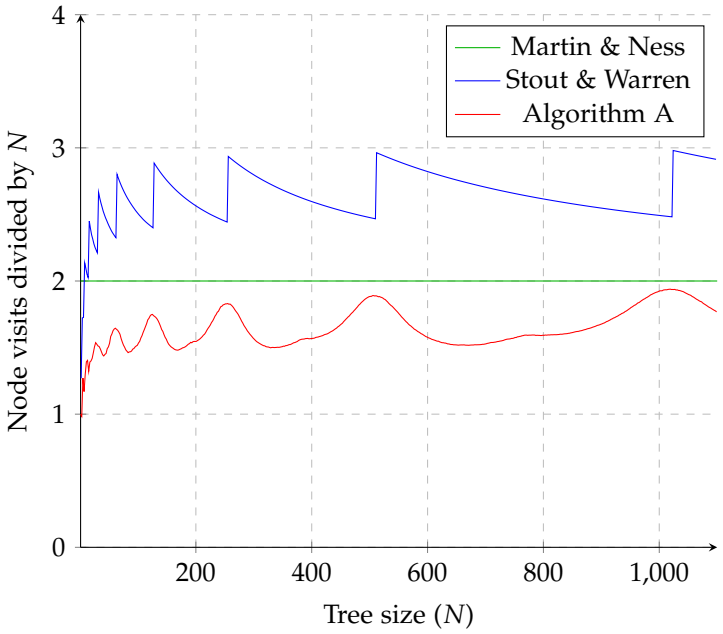


Figure 6.4: Average node visits to balance a balanced binary search tree of size N where $\frac{N}{2}$ nodes are randomly deleted and $\frac{N}{2}$ nodes are randomly inserted

As expected, only Algorithm A proves to be different in the given situations. Both for insertions and deletions in a balanced tree, the algorithm performs better than on a random generated tree of the same size. This is because there is a good chance that the deletions or insertions are equally divided, so that in many nodes, no rebalancing has to be done. Also, the average maximum height after insertions or deletions is less than in the randomly generated tree. After doubling the size of a balanced tree, the maximum possible height is $\frac{N}{2} + \log_2(\frac{N}{2} + 1)$. After bisecting, the maximum possible height is $\log_2(2N + 1)$. Both maximum heights are much smaller than a possible N in a random generated tree. This is logical because then all routes to find medians in balance steps are smaller. Therefore, the performance of the algorithm after deletions is also better than after insertions since $\log_2(2N + 1)$ is less than $\frac{N}{2} + \log_2(\frac{N}{2} + 1)$.

The best performance is seen when both insertions and deletions are performed. As shown in Figure 6.4, the plot of algorithm A is now completely under the $2N$-line. Even though the local maxima are slightly higher the local minima, is lowered considerably. This is due the fact that the insertions and deletions may cancel each other out so that the tree stays more balanced.

Chapter 7

Conclusions

The goal of this thesis was to find an algorithm to globally balance a binary search tree which is competitive and keeps the tree valid during the process. Shown is an algorithm with a worst-case of $\mathcal{O}(N \log(N))$. However, the worst-case situation seems to be very rare in occurrence. By doing experiments with this algorithm as well as some well-known algorithms, we found out that it is comparable in performance to the other algorithms and even performs better in the average-case. The best performance is seen when we try to balance balanced binary search tree on which insertions and deletions are done.

The algorithm works by replacing each unbalanced node by its underlying median. Between these steps it is possible to pause or stop the algorithm so that it leave an opportunity to do some new searching or modifications. It is then guaranteed that the height is smaller or the same as before balancing.

For future work, we can explore the possibilities of the proposed algorithm in weight-balanced trees or B-trees. Since these data structures are very similar to a binary search tree, it should not take much time to convert the algorithm. Also for weight-balanced trees the weight is already implemented. Another area of research is the parallelization. In the algorithm the tree is split multiple times. Another thread should then already be able to continue in the two newly created trees which will later be joined together.

Appendices

Appendix A

Hardware specifications of the test computer

```
*-core
       description: Motherboard
       physical id: 0
     *-memory:0
          description: System memory
          physical id: 3
          size: 3029MiB
     *-cpu
          product: AMD Athlon(tm) 64 Processor 3000+
          vendor: Advanced Micro Devices [AMD]
          physical id: 5
          bus info: cpu@0
          version: 15.15.0
          size: 1800MHz
          capacity: 1800MHz
          width: 64 bits
          capabilities: fpu fpu_exception wp vme de pse tsc msr pae mce cx8 apic sep mtrr
               pge mca cmov pat pse36 clflush mmx fxsr sse sse2 syscall nx mmxext
              fxsr_opt x86-64 3dnowext 3dnow pni lahf_lm cpufreq
        *-cache:0
             description: L1 cache
             physical id: 0
```

```
          size: 128KiB
       *-cache:1
            description: L2 cache
            physical id: 1
            size: 512KiB
    *-memory:1 UNCLAIMED
         description: Memory controller
         product: CK804 Memory Controller
         vendor: NVIDIA Corporation
         physical id: 0
         bus info: pci@0000:00:00.0
         version: a3
         width: 32 bits
         clock: 66MHz (15.2ns)
         capabilities: bus_master cap_list
         configuration: latency=0
```

Bibliography

[AVL62] G.M. Adel'son-Vel'skii and E.M. Landis. Algorithm for the organization of information. *Soviet Mathematics Doklady*, 3(4), 1962.

[Ben75] J.L. Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):509–517, 1975.

[CI84] H. Chang and S.S. Iyengar. Efficient algorithms to globally balance a binary search tree. *Communications of the ACM*, 27(7):695–702, 1984.

[Day76] A.C. Day. Balancing a binary tree. *The Computer Journal*, 19(4):360–361, 1976.

[HCI88] E. Haq, Y. Cheng, and S.S. Iyengar. New algorithms for balancing binary search trees. In *Proceedings of IEEE Southeastcon' 88*, pages 378–382, 1988.

[Hib62] T. Hibbard. Some combinatorial properties of certain trees with applications to searching and sorting. *Journal of Association of Computing Machinery*, 9(1):13–28, 1962.

[HY11] Y. Hirai and K. Yamamoto. Balancing weight-balanced trees. *The Singapore Family Physician*, 21(3):287–307, 2011.

[Knu71] D.E. Knuth. Optimal binary search trees. *Acta Informatica*, 1(1):14–25, 1971.

[Knu73] D.E. Knuth. *The Art of Computer Programming*, volume 3. Addison-Wesley, 1973.

[MI86] A. Moitra and S.S. Iyengar. Derivation of the parallel algorithm for balancing binary trees. *IEEE Transactions on Software Engineering*, 12(3):442–449, 1986.

[MN72] W.A. Martin and D.N. Ness. Optimizing binary trees grown with a sorting algorithm. *Communications of the ACM*, 15(2):88–93, 1972.

[MS14] S. Muthusundari and R.M. Suresh. A sorting based algorithm for the construction of balanced search tree automatically for smaller elements and with minimum of one rotation for greater elements from BST. *Indian journal of computer computer science and engineering*, 4(4):297–303, 2014.

[Muu16] I.J. Muusse. Used code for algorithms and experiments. `http://www.ivo.jelletalstra.nl/scriptie/code/`, 2016.

[NR73] J. Nievergelt and E. M. Reingold. Binary trees of bounded balance. *SIAM journal on Computing*, 2(1):137–142, 1973.

[PV07] S. Pushpa and P. Vinod. Binary search tree balancing: a critical study. *IJCSNS International Journal of Computer Science and Network Security*, 7(8):237–243, 2007.

[ST85] D.D. Sleator and R.E. Tarjan. Self-adjusting binary search trees. *Journal of Association for Computing Machinery*, 32(3):652–686, 1985.

[Ste80] C.J. Stephenson. A method for constructing binary search trees by making insertions at the root. *International Journal of Computer and Information Sciences*, 9(15):15–29, 1980.

[SW86] Q.F. Stout and B.L. Warren. Tree rebalancing in optimal time and space. *Communications of the ACM*, 29(9):902–908, 1986.

[Win60] P.F. Windley. Trees forests and rearranging. *The Computer Journal*, 3(2):84–88, 1960.