Bachelor Thesis

Search Terms

Ahmet Aspir
ahmet.aspir@student.uibk.ac.at

3 June 2019

Supervisor: Dr. Vincent van Oostrom

Eidesstattliche Erklärung

Ich erkläre hiermit an Eides statt durch meine eigenhändige Unterschrift, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe. Alle Stellen, die wörtlich oder inhaltlich den angegebenen Quellen entnommen wurden, sind als solche kenntlich gemacht.
Ich erkläre mich mit der Archivierung der vorliegenden Bachelorarbeit einverstanden.

Datum	Unterschrift

Abstract

The aim of this project is to implement search trees, i.e., trees satisfying the AVL property and the operations on them (search, insert, delete), by means of term rewriting. That is, search trees should be implemented as search terms and the operations on them should be implemented by a finite set of term rewrite rules. Since term rewrite rules act locally, i.e., each term rewrite step modifies only a small part of a term, it is challenging to decompose the operations into appropriate term rewrite rules (allowing for parallel execution), and to show that rewriting with them preserves/creates the AVL-property, at the same time. If time permits, it should be investigated whether extant term rewrite tools (for proving termination) can be of use for showing properties to be preserved/created, and for showing the complexity of the operations.

Contents

1 Introduction **1**
1.1 Motivation . . . 1
1.2 Objective . . . 1

2 Preliminaries **2**
2.1 Abstract Rewriting . . . 2
2.2 Terms . . . 3
2.3 Term Rewriting . . . 3
2.4 Modeling . . . 4
2.5 Search Trees . . . 5
2.5.1 Binary Search Tree . . . 6
2.5.2 AVL Tree . . . 6
2.6 Search Terms . . . 14

3 TRS Implementation **15**
3.1 Search . . . 16
3.2 Insert . . . 16
3.3 Delete . . . 17
3.4 Trinode-Rotation . . . 20
3.4.1 Case 1 . . . 20
3.4.2 Case 2 . . . 22
3.4.3 Case 3 . . . 23
3.4.4 Case 4 . . . 24
3.5 Sequential Implementation . . . 26
3.6 Parallel Implementation . . . 27
3.6.1 Search . . . 28
3.6.2 Insert . . . 28
3.6.3 Delete . . . 28
3.6.4 Trinode . . . 31
3.6.5 Arithmetic . . . 33
3.6.6 Collision Handling . . . 33
3.7 Performance Issues of Arithmetic . . . 33
3.8 Termination . . . 34
3.9 Complexity . . . 34
3.10 Correctness . . . 36
3.11 Rewriting Strategy and Optimization . . . 38
3.12 Parallel Execution . . . 38

3.13 TRS Statistics 38

4 Tools for Development **39**
4.1 TRS Engine 39
4.2 Parsing Terms 40
4.3 Validation of Terms 40
4.4 Drawing Trees 40
4.5 Termination 41
4.6 Complexity 41

5 Running the TRS **42**
5.1 Testing 42
5.1.1 Result 42
5.2 Runtime Measurements 43

6 Conclusion **45**

Bibliography **46**

A Calculations **48**
A.1 Imbalances of Case 2 48
A.2 Imbalances of Case 3 48
A.2.1 First Value 48
A.2.2 Second Value 49
A.2.3 Third Value 50
A.3 Imbalances of Case 4 50
A.3.1 First Value 50
A.3.2 Second Value 51
A.3.3 Third Value 51

B TRS Codes **53**
B.1 Parallel AVL 53
B.1.1 Search 53
B.1.2 Insert 53
B.1.3 Delete 54
B.1.4 Trinode 57
B.1.5 Collision Handling 57
B.1.6 Arithmetic 58
B.2 Sequential AVL 58

1 Introduction

1.1 Motivation

Term rewriting [18] is a Turing-complete model of computation and one of the main aspects of it is the analysis of termination.

We want to know which existing procedures, algorithms, structures, etc. can be modeled by a term rewrite system at their natural level of abstraction. It also serves as a case study, to see if term rewrite systems are suited for the implementation of search trees.

Modeling rules that are based on pattern matching fit very well for a problem that evolves around structures such as trees, thus meeting the criterion for a good test subject.

If implemented correctly, term rewrite systems are also able to perform multiple rewrite steps at once, fully utilizing a parallel implementation.

1.2 Objective

In this thesis we have three main objectives:

1. Implement search trees and the operations on them as a term rewrite system with a finite set of rules.

2. Extend the implementation so the parallel execution of operations is possible.

3. Proof termination for the implemented term rewrite systems.

We will focus on one specific search tree, namely the AVL (Adelson-Velsky and Landis) tree. The operations modeled by the TRS should not have worse runtime-complexity than the conventional operations on an AVL tree.

Once implemented, termination for the term rewrite systems needs to be proven. We want any operations to yield an AVL tree, meaning if a rewrite sequence yields a normal form of a term, this term should represent an equivalent AVL tree.

Ideally, termination should be shown with existing termination tools. Other goals are to extend modeling methods and to show the strengths and weaknesses of certain tools such as the Tyrolian Termination Tool, also known as $\mathsf{T_TT_2}$ [9].

2 Preliminaries

The idea of term rewriting is to operate with a set of *rewrite rules* by applying pattern matching to certain terms. More general for rewrite systems: Find a pattern and apply a rule out of the set of rules to obtain a new or modified expression being whether a term, graph, string or so on.

Before getting to the main topic, the formal definition of the used concepts needs to be addressed. The relevant parts of term rewriting will be mentioned and a brief introduction to the data structures with their operations will also be shown. For the following definitions, these sources have been used:

- Course-Material: Term Rewrite Systems [10]

- Course-Material: Algorithms and Data Structures [8].

2.1 Abstract Rewriting

To define the general properties of rewrite systems, we first look at *abstract rewrite systems* (ARS) [10, Chapter 1].

Definition 2.1. An abstract rewrite system $\mathcal{A}$ has a set A and a relation $\to$, in short: $\mathcal{A} = \langle A, \to \rangle$. For elements $a, b \in A$: $a \to b$ denotes a *rewrite step.*

With the formal definition of an abstract rewrite system, some more properties on them are now defined.

Definition 2.2. With respect to previously defined ARS $\mathcal{A}$: The sequence $(a_0, ..., a_n)$ is called a *finite rewrite sequence* if $a_i \to a_{i+1}$ for some $n \in \mathbb{N}$ and $0 \leq i < n$.

Definition 2.3. The sequence (a_i) is called an *infinite rewrite sequence* if $\forall i \in \mathbb{N}$: $a_i \to a_{i+1}$.

Definition 2.4. If it is not possible to build an infinite rewrite sequence starting from $a \in A$, then this particular element is *terminating.* An ARS $\mathcal{A} = \langle A, \to \rangle$ is terminating if $\forall x \in A$: x is terminating.

Definition 2.5. For $\mathcal{A} = \langle A, \to \rangle$: An element $a \in A$ is in *normal form* if $a \to b$ does not exist where $b \in A$.

2.2 Terms

Here, we want to formally introduce *terms* and concepts surrounding them [10, Chapter 2, 2.1].

Definition 2.6. The set $\mathcal{F}$ of function symbols is called *signature*. The arity of a function symbol $f \in \mathcal{F}$ is n where $n \in \mathbb{N}$. This means that each function symbol f takes a certain number n of arguments and if $n = 0$, the function symbol is called a *constant*.

Definition 2.7. A set of *variables* $\mathcal{V}$ and the set of function symbols $\mathcal{F}$ build the set of *terms* $\mathcal{T}(\mathcal{F}, \mathcal{V})$. It is important to note that $\mathcal{F} \cap \mathcal{V} = \varnothing$. Terms in $\mathcal{T}(\mathcal{F}, \mathcal{V})$ are:

- Variables
- Constants
- If a function symbol $f \in \mathcal{F}$ with n arguments where $n \in \mathbb{N} \setminus \{0\}$ and $t_1, ..., t_n$ are terms, then $f(t_1, ..., t_n)$ is a term.

Definition 2.8. The mapping σ from $\mathcal{V}$ to $\mathcal{T}(\mathcal{F}, \mathcal{V})$ is called *substitution*. The result of the substitution σ of a term t is $t\sigma$. The *domain* is defined $\mathcal{D}\text{om}(\sigma) = \{x \mid \sigma(x) \neq x\}$. Applying substitution is done by the following definition:

$$t\sigma = \begin{cases} \sigma(t) & \text{if } t \text{ is a variable} \\ f(t_1\sigma, ..., t_n\sigma) & \text{if } t = f(t_1, ..., t_n) \end{cases}$$

The expression $t\sigma$ is called an *instance* of t.

Example 2.9. If σ is expressed as a set, then it would look like this:

$$\sigma = \{x_1 \mapsto t_1, ..., x_n \mapsto t_n\}.$$

For Definition 2.8, consider following: $\mathcal{D}\text{om}(\sigma) = \{x_1, ..., x_n\}$ and $t_i = x_i\sigma$ for $1 \leq i < n$.

Example 2.10. The substitution $\sigma = \{x \mapsto n(x, z, y), y \mapsto n(y, z, x)\}$ has $\mathcal{D}\text{om}(\sigma) = \{x, y\}$. Applying substitution to the term $t = n(x, z, y)$ results in $t\sigma = n(n(x, z, y), z, n(y, z, x))$.

Definition 2.11. A term s *matches* a term t if t is an instance of s.

2.3 Term Rewriting

Term rewrite systems induce abstract rewrite systems [10, Chapter 3].
A concept called *context* will also be introduced which is very intuitive, but still needs to be formally defined.

Definition 2.12. The set $\mathcal{T}(\mathcal{F}, \mathcal{V})$ is extended by a special symbol $\Box$ called *hole* which is a constant. This leads to $\mathcal{T}(\mathcal{F} \cup \{\Box\}, \mathcal{V})$. Let $C \in \mathcal{T}(\mathcal{F} \cup \{\Box\}, \mathcal{V})$, so that C has exactly one hole. Context $C[t]$ means that the hole in C is replaced by the term t.

Example 2.13. For $C = n(\Box, y, z)$ and a context $C[n(a, b, c)]$ the hole filling yields: $C[n(a, b, c)] = n(\Box, y, z)[n(a, b, c)] = n(n(a, b, c), y, z)$.

Definition 2.14. An *equation* $s \approx t$ consists of two terms $s, t \in \mathcal{T}(\mathcal{F}, \mathcal{V})$.

Definition 2.15. An *equational system* $ES(\mathcal{F}, \mathcal{E})$ has a signature $\mathcal{F}$ and a set of equations $\mathcal{E}$ between terms in $\mathcal{T}(\mathcal{F}, \mathcal{V})$.

Definition 2.16. For the relation $s \rightarrow_{\mathcal{E}} t$ on terms in an equational system $\mathcal{E}$, following properties must hold:

- $\exists e$: $e \in \mathcal{E} \wedge e = \ell \approx r$
- $\exists \sigma, \exists C$: $s = C[\ell\sigma] \wedge t = C[r\sigma]$

where σ is a substitution.

Definition 2.17. An equation $\ell \approx r$ is called a *rewrite rule* if:

- ℓ is not a variable and
- variables that appear in r also appear in ℓ, meaning $\mathcal{V}\text{ar}(r) \subseteq \mathcal{V}\text{ar}(\ell)$.

Rewrite rules will be written $\ell \rightarrow r$ instead of $\ell \approx r$. A *term rewrite system* (TRS) is an equational system where all equations are rewrite rules.

Example 2.18. The following TRS $\mathcal{R}$ expresses associativity of terms that represent nodes.
Rules:

- $n(v, w, n(x, y, z)) \rightarrow n(n(v, w, x), y, z)$
- $n(n(v, w, x), y, z) \rightarrow n(v, w, n(x, y, z))$

Here $\mathcal{V}\text{ar}(\mathcal{R}) = \{v, w, x, y, z\}$ and $\mathcal{F}(\mathcal{R}) = \{n\}$.

2.4 Modeling

In this section, we want to state what the modeling of a TRS means and what properties are desired.

When modeling external structures by TRS, we should be able to transform the term of a TRS back to its original form and vice versa at all times, in other words, the original structure needs to be preserved.

The graph in Figure 2.1 summarizes every important aspect of modeling.

$$\begin{tikzcd}
T \arrow[r, shift right, "\mathcal{T}_{RR}(T)"'] \arrow[d, "\mathcal{M}"'] & S \arrow[l, shift right, "\mathcal{R}_{RR}(S)"'] \arrow[d, dotted, "\mathcal{M}_{TRS}"] \\
\mathcal{M}(T) \arrow[r, shift right, "\mathcal{T}_{RR}(\mathcal{M}(T))"'] & \mathcal{M}_{TRS}(S) \arrow[l, shift right, "\mathcal{R}_{RR}(\mathcal{M}_{TRS}(S))"']
\end{tikzcd}$$

Figure 2.1: Modeling

The expressions in the graph above are explained below.

- $\mathcal{T}_{RR}(X)$ is the transformation of a structure X into a term x.
- $\mathcal{R}_{RR}(y)$ is the transformation of a term y into a structure Y.
- T is a tree structure and S is the term for T in a TRS.
- $\mathcal{M}$ represents operations on a tree.
- $\mathcal{M}_{TRS}$ represents many small rewriting steps locally that model $\mathcal{M}$.

2.5 Search Trees

Search trees are data structures that enable, if balanced, better runtime-complexities, in regards to their update-operations.

For a search operation and the number of elements n, special forms of binary search trees can achieve a runtime-complexity of $\mathcal{O}(\log n)$ [8, Proposition 10.2].

One special search tree is a balanced binary search tree called AVL tree. This particular tree is self-balancing and is defined by a certain property. To maintain this property, a set of actions is defined.

Definition 2.19. For a set of *nodes* N and *edges* E, we call a graph a tree if:

- the edges are directed,
- the nodes are connected in an undirected-acyclic way,
- exactly one unique path from the *root* node to an arbitrary node exists.

A tree consists of nodes and a node has up to c *child* nodes, a single *parent* node and holds a certain value. Every tree has exactly one root which is a node without a parent. A node with empty value and without any children is called a *leaf.* An *internal* node is a node with at least one child that is not a leaf.

2.5.1 Binary Search Tree

Definition 2.20. A binary search tree is a binary tree [8, Sections 7, 7.3, 10.1] which means every node has at most two children. For a set S with the order $\geq$ and any node m that holds the value $v \in S$ and $\ell, r \in S$:

- A node with value ℓ is in the left subtree of m if: $\ell \leq v$.
- A node with value r is in the right subtree of m if: $r \geq v$.

Definition 2.21. The *inorder* list of a binary search tree is built by taking the values of the tree in inorder traversal. The order $\geq$ on this list must hold.

2.5.2 AVL Tree

An AVL tree is a binary search tree which tries to maintain balancedness.

Definition 2.22. For a node m, the *imbalance* is $b(m) = \mathit{height}(\ell) - \mathit{height}(r)$, where *height* is the height of the specified subtree and ℓ and r are the left and right subtrees respectively. If a node has m has $b(m) = 0$, it is called *balanced*. If all the nodes of an AVL tree have an imbalance of 0, we call this tree balanced as well. The *balancedness* $\mathcal{B}$ is defined as following, with N being the set of nodes of a tree:

$$\mathcal{B} = \begin{cases} \mathit{true} & \text{if } \forall m \in N\colon\ b(m) \in \{-1, 1, 0\} \\ \mathit{false} & \text{if } \exists m \in N\colon\ b(m) \notin \{-1, 1, 0\}. \end{cases}$$

If $\mathcal{B}$ is *false*, then the tree needs to be restructured with respect to its inorder list so it is *true* again. A binary search tree T is called an AVL tree, if $\mathcal{B}$ for T is *true*.

Operations and Algorithms

The main focus in this part will be on the operations and modifications of an AVL tree. We will first describe the operations *search*, *insert* and *delete*, which are the same ones used on binary search trees [8, 10.1.2 Update Operations].

In the second part, we will show how to deal with an unbalanced AVL tree, which can be a result of either an insert or a delete. To restore the balancedness, we resort to the restructure operation for an AVL tree, where three nodes and their subtrees are involved. This restructuring is also known as *trinode-rotation*.

Definition 2.23. For the *search* operation, the starting point is the root. Let x be the numeric value that is being searched for and v the value of the root-node. If $x \geq v$ and $x \neq v$, the right node is visited, otherwise the left node is visited. At the visited node, the same operation is applied. This process is repeated until either the target node is reached with $x = v$, or a leaf is reached. Reaching a leaf means the search was not successful.

We want to describe the complexity of the search operation and for this, we assume a balanced AVL tree.

When searching, at each node one subtree becomes irrelevant. This happens for all visited nodes.

Let k be the steps to reach the target node and let n be the total number of nodes. At each step, the number of elements to take into consideration is being halved. We can say that the target node is found after k steps which yields $n \cdot (\frac{1}{2})^k = 1 \Rightarrow \log_2 n = k$.

Omitting the base of $\log$, we speak of logarithmic complexity, hence the runtime-complexity of this operation is $\mathcal{O}(\log n)$.

Since AVL trees are not always necessarily balanced, another way of describing the runtime-complexity can be found here [8, Proposition 10.2].

Definition 2.24. In order to *insert*, a similar procedure as in Definition 2.23 is used until a leaf is reached. Once a leaf is hit, this leaf becomes a new node holding the value that was inserted. The new node has two leaves as children.

The runtime-complexity of this operation is the same as in Definition 2.23, hence $\mathcal{O}(\log n)$.

Definition 2.25. For *delete*, the first task is to find the correct node which is, again, a very similar procedure to the search in Definition 2.23. Once the correct node that needs to be deleted is found, three scenarios have to be considered:

1. The node has two leaves as children.
2. The node has one leaf and one non-leaf as children.
3. The node has two non-leaf nodes as children.

For scenario 1 the node can simply be replaced by a leaf and for scenario 2 the node can be replaced by its non-leaf child node.

Regarding scenario 3 either the biggest inorder predecessor or the smallest inorder successor needs to found. In this case, we chose the successor as a fitting replacement and it is found by searching the leftmost node in the right subtree of the node that is being deleted.

For this operation, we also get a complexity of $\mathcal{O}(\log n)$, as again, the routine in Definition 2.23 is deployed.

Example 2.26. We want to demonstrate scenario 3 from Definition 2.25. The node with value 6 is being deleted and its inorder successor 7 takes its place.

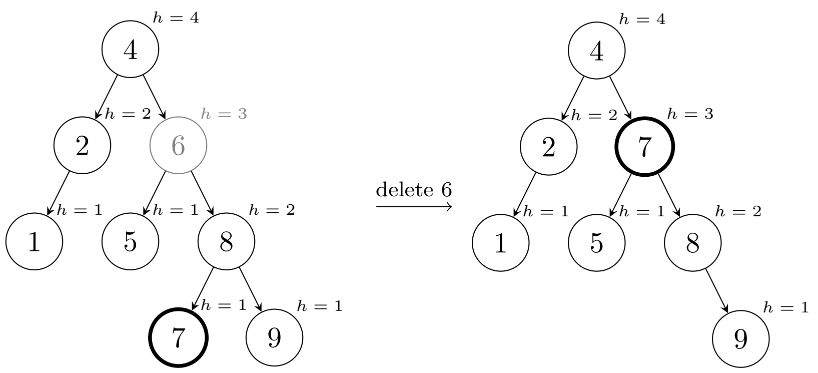

Figure 2.2: Deleting a node with two internal nodes as children

Definition 2.27. The insert and delete operations yield a node p which is described as following:

- **Insert**: $insert(p)$, where p is the node that was inserted.
- **Delete**: $p = delete(x)$, where $delete(x)$ returns the parent of the deleted node.

With the p for either deleting or inserting, some ancestor node z can have an imbalance that makes $\mathcal{B}$ *false*. By marking three nodes and initiating a restructuring, $\mathcal{B}$ can become *true* again.
This is done in four steps by assigning labels to specific nodes that are related to p.

To fix the height-property, the procedure below can be applied.

1. Let z be the first unbalanced node along the path from p to the root.
2. Let y (an ancestor of p) be the child of z that has a greater height than its sibling.
3. Let x (also an ancestor of p, possibly $x = p$) the child of y that has a greater height than its sibling.
4. $restructure(x)$

The subtree with the labeled nodes is now balanced.

Figure 2.3: Marking Algorithm [8, Figure 10.8]

The algorithm for $restructure(x)$ can be seen in Figure 2.4 and is able to restore the

AVL property. It is important to mention that a local fix may not necessarily make $\mathcal{B}$ hold again. Every unbalanced node needs to be restructured.

restructure(x):

Input: Position x with parent-node y and grandparent-node z in a Tree T.
Output: T after a trinode-rotation.

1. Let (a, b, c) be an inorder-list of positions x, y, and z and let (x_0, x_1, x_2, x_3) be an inorder-list of the four subtrees with roots x, y, and z.
2. Replace the subtree with root z with the subtree with root b.
3. Let a be the left child of b. Subtrees x_0 and x_1 need to be the left and the right subtree of a.
4. Let c be the right child of b. Subtrees x_2 and x_3 need to be the left and the right subtree of c.

Figure 2.4: Trinode-Rotation Algorithm [8, Code Fragment 10.6]

When applying the restructure algorithm, four possible scenarios are considered. These affect the marking of each node, but the results are always subtrees with fixed imbalances. The four cases can be found in Figures 2.5, 2.6, 2.7, 2.8 which show different structures where the node with the label z always has an imbalance that violates $\mathcal{B}$.

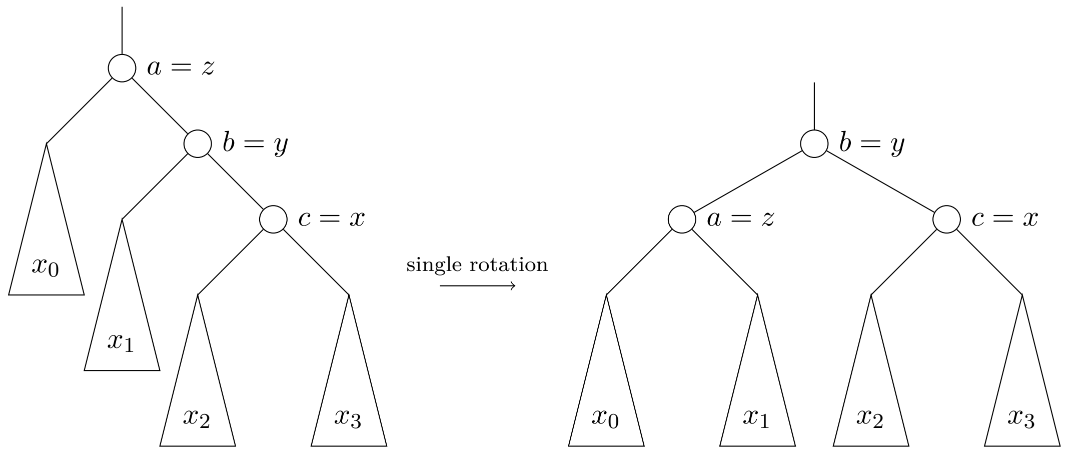

Figure 2.5: Trinode-rotation single rotation - Case 1

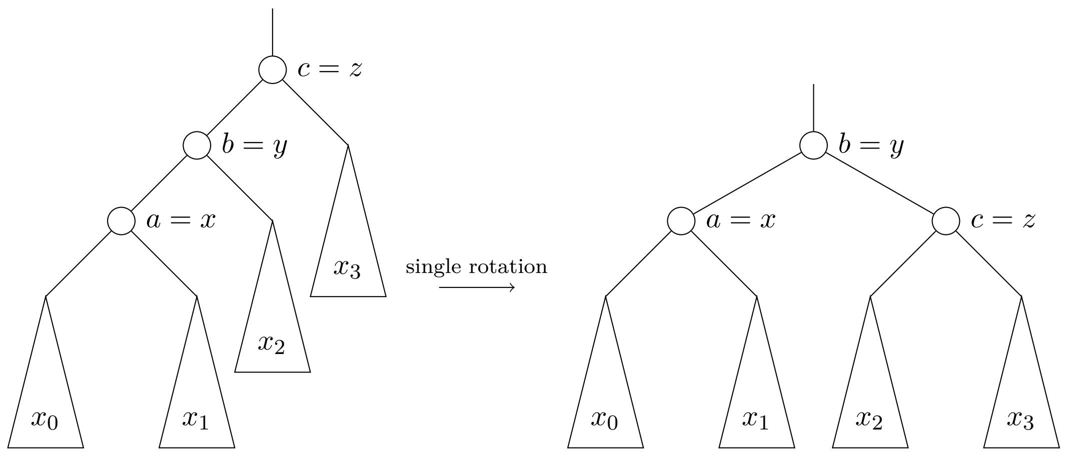

Figure 2.6: Trinode-rotation single rotation - Case 2

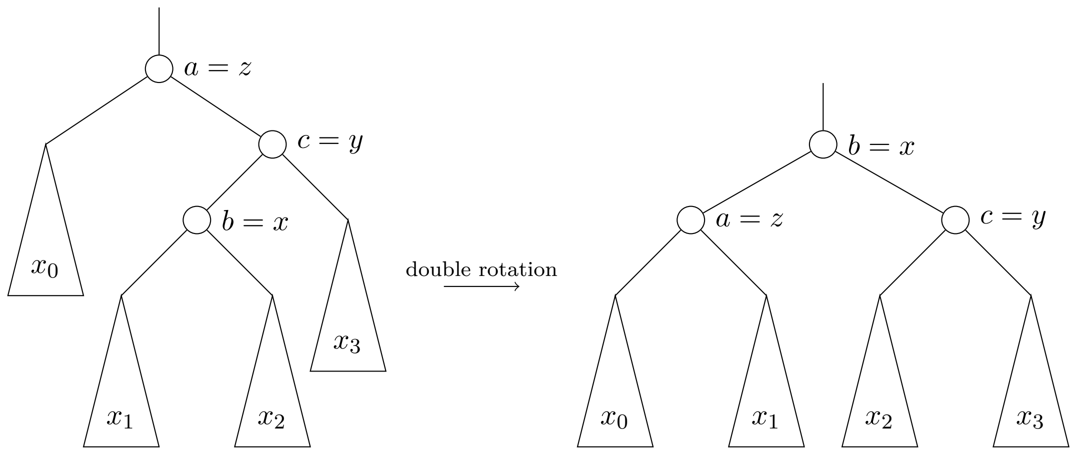

Figure 2.7: Trinode-rotation double rotation - Case 3

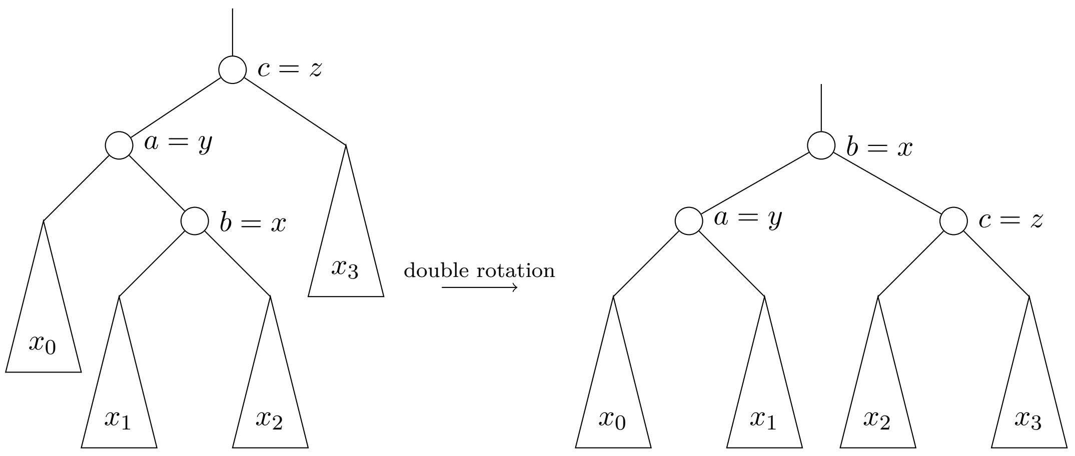

Figure 2.8: Trinode-rotation double rotation - Case 4

When $b = y$ (e.g. Figure 2.6), the transformation is called *single rotation*. The best way to imagine this is to think of y being rotated over z, a single node.

Let us consider the illustration in Figure 2.9. The node b and its subtree x_2 are being rotated over the marking point •. The subtree x_2 becomes a subtree of c.

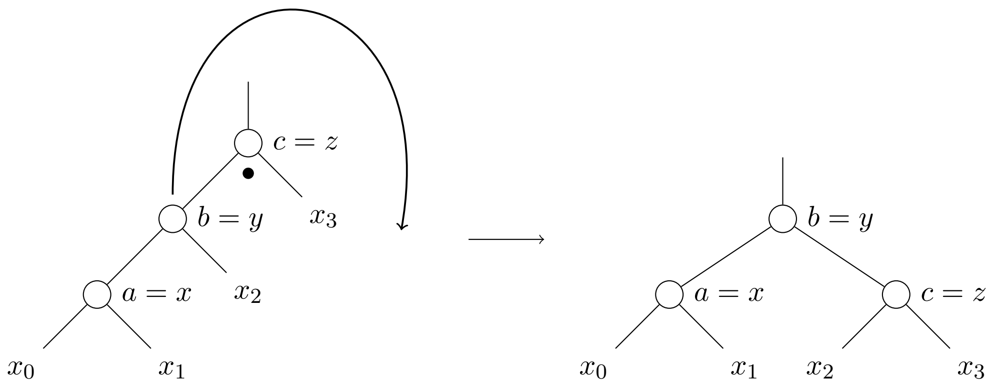

Figure 2.9: Single rotation

When $b = x$ (e.g. Figure 2.7), the rotation is called *double rotation*. Here one can think of x being rotated over y and then z, two nodes so to say, hence double rotation.

Now, the difference in the double rotation is that we perform the action twice at different nodes and in different directions.

First off, the node b and its subtrees x_1 and x_2 are being rotated over $\bullet$. Again, x_2 becomes a subtree of the node c. Then, b and its subtree x_1 are being rotated over $\bullet$, so x_1 becomes a subtree of a.

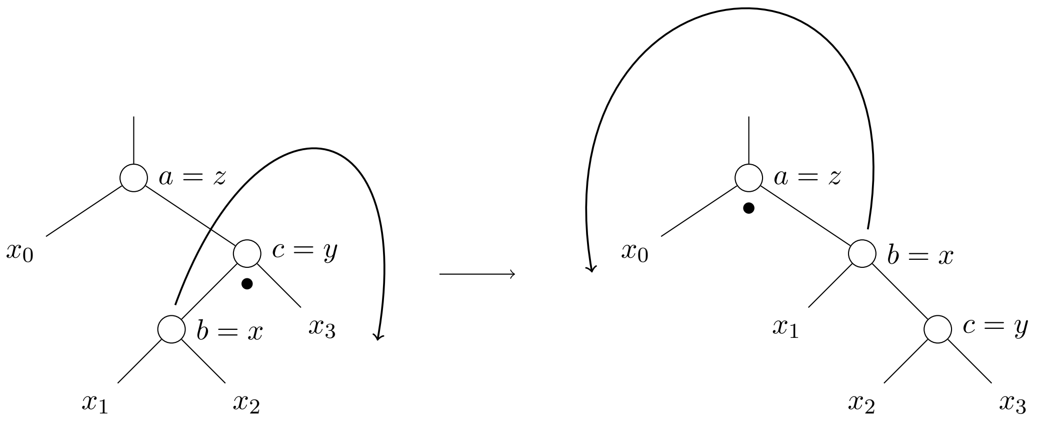

Figure 2.10: Double rotation - Part 1

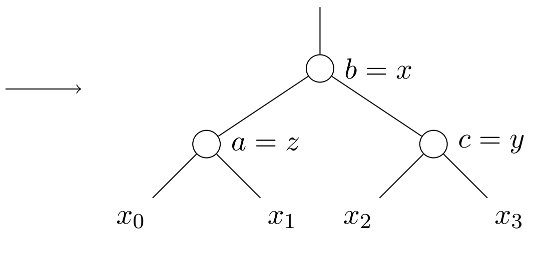

Figure 2.11: Double rotation - Part 2

Example 2.28. We have an AVL tree with 7 nodes as seen on the left side in Figure 2.12. The node with the value 1 is deleted and a violation of $\mathcal{B}$ is created by the nodes holding the values 2 and 6, where h designates the height of a node.

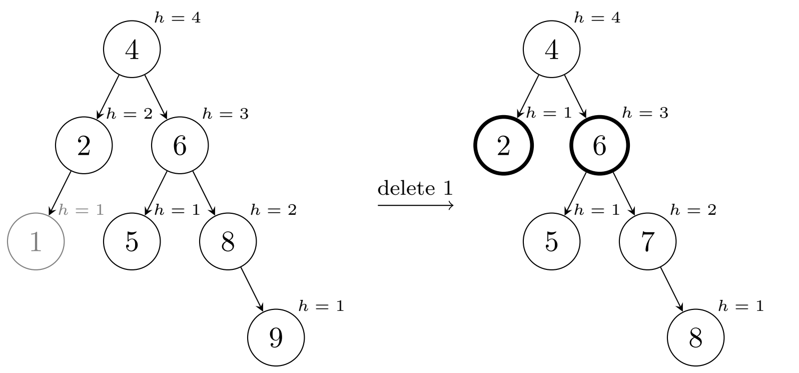

Figure 2.12: Height property violation of an AVL tree

To check if $\mathcal{B} = \mathit{true}$, we need to traverse the tree and check for every node n if $b(n) \in \{-1, 1, 0\}$. Looking at the root *root*, we apply: $b(\mathit{root}) = \mathit{height}(\ell) - \mathit{height}(r) = 1 - 3 = -2$. Now, with $\mathcal{B}$ being *false*, a swift application of restructuring can restore $\mathcal{B}$ and make it *true* again.

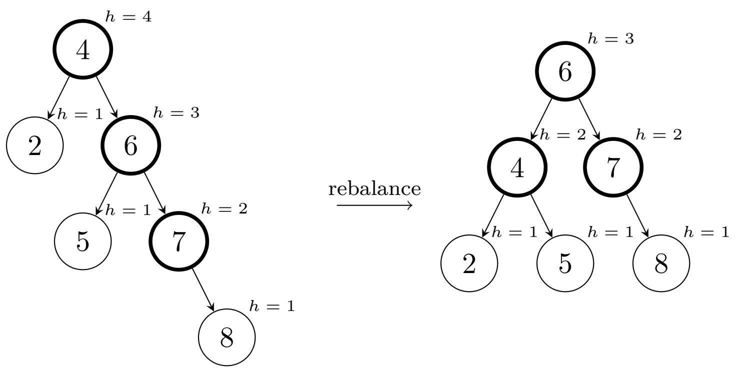

Figure 2.13: Balancing of an AVL tree

After the restructuring, the inorder list is still the same and $\mathcal{B}$ now holds.

This technique will be the basis for designing the TRS that models the operations. With the main operations of the AVL tree defined, we can start modeling a TRS transforming the existing procedures into terms and rules.

2.6 Search Terms

In this section, we will show the link between terms and trees.

Definition 2.29. We refer to *search terms* as the term representation of a search tree.

Definition 2.30. Let x be a search term, ℓ and r the functions that yield the left and right node respectively. Also, let v be the function that gives the current value of the node and let $\bullet$ be a leaf. Lastly the function @ concatenates elements to a tuple.

For the function I that yields the inorder list of search term, we get:

$$I(x) = \begin{cases} I(\ell(x)) \mathbin{@} v(x) \mathbin{@} I(r(x)) & \text{if } \ell(x) \neq \bullet \wedge r(x) \neq \bullet \\ v(x) \mathbin{@} I(r(x)) & \text{if } \ell(x) = \bullet \wedge r(x) \neq \bullet \\ I(\ell(x)) \mathbin{@} v(x) & \text{if } \ell(x) \neq \bullet \wedge r(x) = \bullet \\ v(x) & \text{if } \ell(x) = \bullet \wedge r(x) = \bullet \end{cases}$$

Proposition 2.31. *The application of the node function n to an inorder list L behaves in an associative manner with regards to the inorder lists of the resulting terms. If p and q are different search terms that were created by applying n to L, then $I(p) = I(q)$.*

Suppose our term for the nodes of a tree looks like this: $n(x, y, z)$. The first and second argument of n denote other nodes and the second one is the value of the node.

Let $L = (a, b, c, d, e)$ be an inorder list of a search tree. We can apply the function n to L in any shape, as long as we apply it correctly. So, the only possible combinations are $s = n(n(a, b, c), d, e)$ and $t = n(a, b, n(c, d, e))$.

Even though $s \neq t$, we can see that $I(s) = I(t)$, because the function I takes always takes the leftmost value of a term. Visually speaking, the inorder list of a search term can be obtained by extracting all values from left to right.

Due to the associativity stated in Proposition 2.31, it is possible that different representations of trees as terms, can still have the same inorder list.

If a trinode-rotation does not change the order of the elements in the search term, then we can say that the rotation preservers the inorder list I of a search term.

3 TRS Implementation

In this section, we will describe the implementation process and parts of the TRS. As a syntax for the TRS, the *TPDB* [3] format was chosen, which is commonly used by termination tools. See the folder *TRS-Codes*[1] for the complete TRS implementations.

The first goal is to achieve a simple implementation which is able to handle every operation correctly and efficiently. Once reached, it is also desired to allow simultaneous execution of operations.

We also want the TRS to model the operations so that the original tree is preserved, as shown in Figure 2.1 and described in Section 2.4. The AVL tree needs to be able to be constructed using the search term.

In regards to terms, the key idea is using the imbalance as part of the node term so any recomputation of it, probably caused by operations, can be done locally.

Definition 3.1. Natural numbers are represented by the function symbol s and the constant 0. The application of s n-times to 0, represents the natural number n.

Example 3.2. The term $s(s(0))$ represents the natural number 2 and $-(s(s(0)))$ is -2.

Definition 3.3. A node n expressed by a term is $n(d, \ell, v, r)$ where ℓ and r are nodes themselves and v is a value e.g. $v \in \mathbb{N}$. The first argument d represents the imbalance from Definition 2.22.

Example 3.4. We want to show search terms for given AVL trees. In this example, instead of sequences of the function symbol s, we will write the natural number. The heights will be used to get the imbalances at each node and the leaves will be represented by $\bullet$.

For the AVL trees shown in Figure 2.12, we get following search terms from left to right respectively:

$n(-1, n(+1, n(0, \bullet, 1, \bullet), 2, \bullet), 4, n(-1, n(0, \bullet, 5, \bullet), 6, n(-1, \bullet, 8, n(0, \bullet, 9, \bullet)))) \xrightarrow{delete\ 1}$
$n(-2, n(0, \bullet, 2, \bullet), 4, n(-1, n(0, \bullet, 5, \bullet), 6, n(-1, \bullet, 8, n(0, \bullet, 9, \bullet))))$.

For the ones in Figure 2.13, we get:

$n(-2, n(0, \bullet, 2, \bullet), 4, n(-1, n(0, \bullet, 5, \bullet), 6, n(-1, \bullet, 8, n(0, \bullet, 9, \bullet)))) \xrightarrow{rebalance}$
$n(0, n(0, n(0, \bullet, 2, \bullet), 4, n(0, \bullet, 5, \bullet)), 6, n(-1, \bullet, 7, n(0, \bullet, 8, \bullet)))$

An operation can cause an imbalance to become ± 2. Based on that, we only allow imbalance updates to be values in the range of $[-2, 2]$. With respect to this constraint, the TRS will be modeled.

[1] `https://git.uibk.ac.at/csat1763/SearchTerms/tree/master/TRS-Codes`

We will first look at the operations, then the trinode-rotation, and then how they are implemented as sequential and parallel term rewrite systems.

3.1 Search

Searching is implemented by translating the procedure in Definition 2.23 directly into terms and rewrite rules. Deploying a search either yields *false* or an entry for the value. The entry consists of a boolean indicating the positive result and a mapping for the value. This is for the key-value pairs commonly used in search trees. Only one search at a time can be deployed.

Example 3.5. We show what a positive search yields. Suppose we are searching the value 10. If 10 is in the search term, the result will be: $\mathit{entry}(\mathit{true}, \mathit{getValueFor}(10))$. The mapping $\mathit{getValueFor}(x)$ needs to be an explicit rewrite rule for every value.

3.2 Insert

The concept of inserting is described in Definition 2.24, which translates well into a TRS. Once a node is inserted, it is mandatory to update the imbalances of the involved nodes, being the ones that are in the path from the inserted node to the root. The node that is being inserted is wrapped with $\upuparrows^+$ to indicate that a node has been added so the parent node can update its imbalance. When the parent sees this marking, it checks whether it comes from the left or right subtree and acts accordingly. If the term looks like $n(d_0, \upuparrows^+(x_0), x_a, x_1)$, then $d_0 = d_0 + 1$ is done, and if $n(d_0, x_0, x_a, \upuparrows^+(x_1))$, $d_0 = d_0 - 1$ is done. Following the update of the imbalance, if $d_0 = \pm 1$ then the propagation is continued by wrapping the current node in $\upuparrows^+$ leading to $\upuparrows^+(n(d_0, x_0, x_a, x_1))$. If a node updates its imbalance to $d_0 = 0$, the propagation can be stopped, because the height of the parent node is not affected.

In case the imbalance of a node is updated to $+2$ or -2, due to a propagation, a trinode-rotation is executed. The propagation-symbol is held back until the result of the rotation is present. Depending on the imbalance of the highest node term in the result of the rotation, the propagation is either canceled if the imbalance is 0, or continued otherwise.

Example 3.6. We demonstrate how an insert is dealt with. After inserting the node, the propagation is started and passed from child to parent. In this process the imbalances are updated accordingly. Here, the criterion to stop the propagation is met by performing a trinode-rotation, since the result of the rotation is an imbalance of 0.

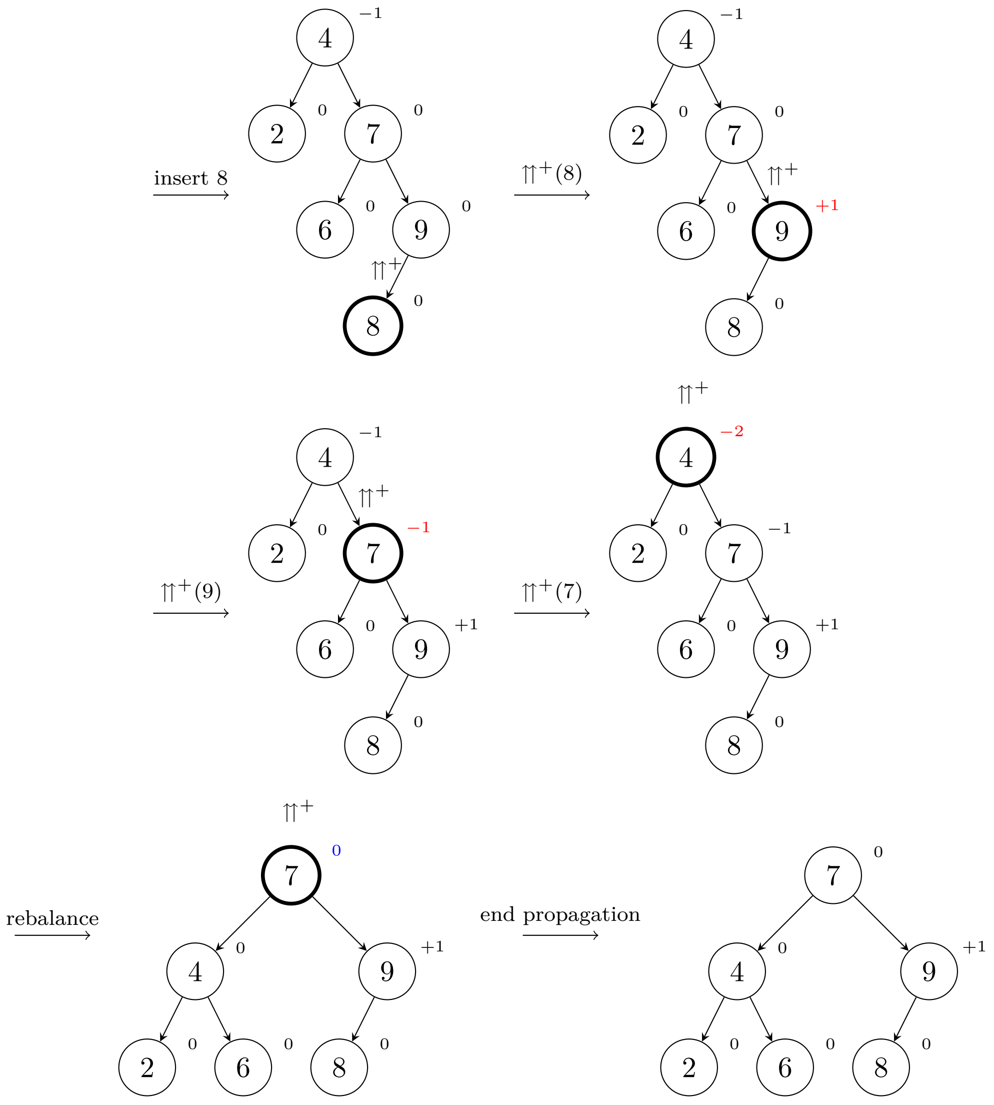

Figure 3.1: Insert with propagation $\upuparrows^{+}$ and stop condition

In this case, we meet two criteria for the cancellation of the propagation, namely reaching the root and the update of an imbalance to 0.

3.3 Delete

Deleting happens as mentioned in Definition 2.25 and the rewrite rules easily follow from that definition as well.

Similar to insert, every delete needs a propagation too. Here, a new leaf that replaces either the deleted node or the inorder successor of the deleted node, will be the initial propagation point.

The rules take different cases into consideration and depending on the structure, replacing the target-node by a leaf or its non-leaf child happens in one rewrite step. If the inorder successor needs to be found in the right subtree, however, the target-node-value is replaced by a placeholder-constant and remains as the value of the node. The right subtree of the target-node is being reconstructed in a way so the inorder successor of the target is not in it anymore. Once the inorder successor is transformed into a leaf, a double propagation will be issued at this starting point, with one being the replacement value for the placeholder at the initially deleted node and the other the imbalance update propagation. Since in this case deleting the target-node means swapping its value with the one of its inorder successor and actually deleting the latter, propagation is started from the inorder successor's old place.

The node of the initial propagation is wrapped in $\upuparrows^-$. Now if $n(d_0, \upuparrows^-(x_0), x_a, x_1)$ occurs, the imbalance becomes $d_0 = d_0 - 1$ otherwise for $n(d_0, x_0, x_a, \upuparrows^-(x_1))$ the imbalance becomes $d_0 = d_0 + 1$. The node that acted upon the propagation wraps itself with the symbol becoming $\upuparrows^-(n(d_0, x_0, x_a, x_1))$ and propagation is continued until a node changes its imbalance to $d_0 = \pm 1$. The problematic case is the change to $d_0 = 0$ because that means that the height of that node shrank by one which continues propagation. This also means that every trinode-rotation that yields a balanced node with an imbalance of 0 also continues the propagation.

Example 3.7. For the AVL tree below, we delete 11 and start a propagation from the leaf that replaces 11. We abstract the subtree x which has the same height as the node 13.

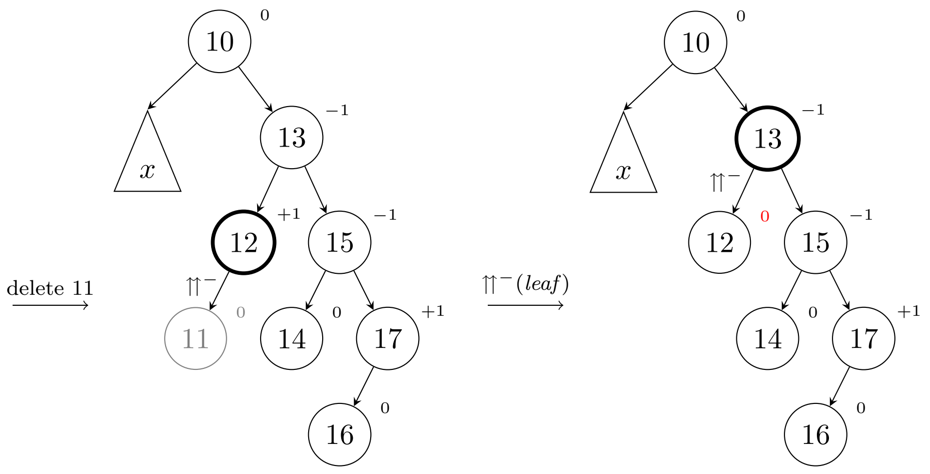

Figure 3.2: Delete with propagation $\upuparrows^-$ and stop condition - Part 1

We can see that the imbalances are updated accordingly leading to an unbalancedness at the node 13. When an imbalance of -2 or $+2$ is reached, a trinode-rotation will be performed. Depending on the result of the rotation and the imbalance of the highest node term, the propagation is continued. In this case the imbalance of 0 does not stop the propagation.

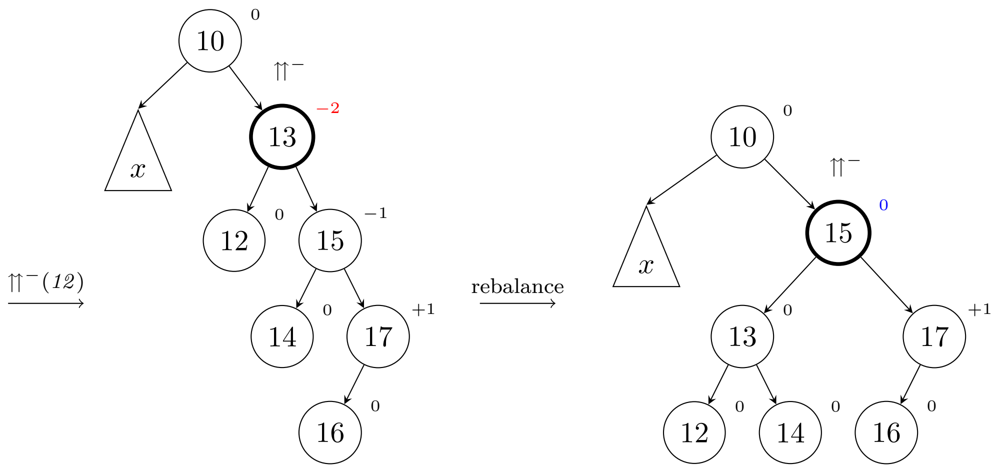

Figure 3.3: Delete with propagation $\upuparrows^-$ and stop condition - Part 2

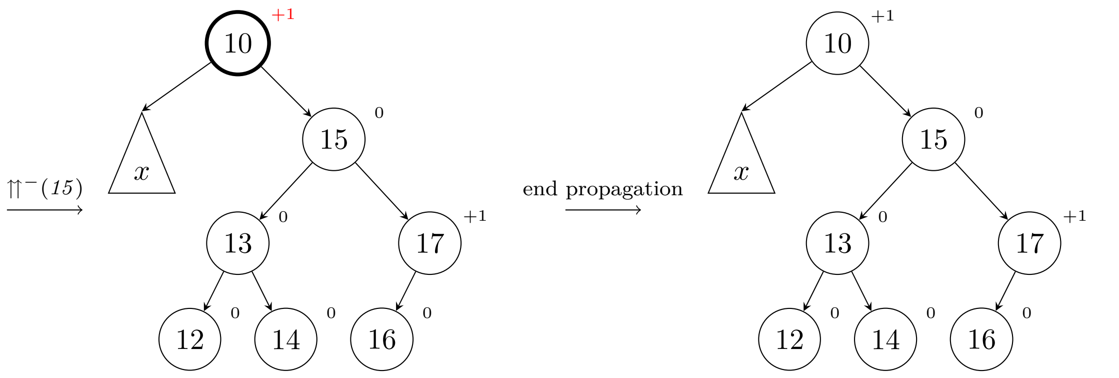

Figure 3.4: Delete with propagation $\upuparrows^-$ and stop condition - Part 3

Finally, the propagation reaches the root and the imbalance of it is updated to $+1$. Once again, we meet two criteria for the cancellation of the propagation, namely reaching the root and an imbalance of $+1$.

3.4 Trinode-Rotation

Generally speaking, there are four cases as shown in Figures 2.5–2.8. These cases directly represent the imbalances of the involved nodes. This gives us the option to extract the new imbalances of the restructured terms for each case.

3.4.1 Case 1

As seen in Figure 2.3, nodes for the trinode-rotation are selected based on their height compared to their sibling node. This approach is achieved by matching the correct imbalances. For the case in Figure 2.5, the nodes y and x must be higher than their siblings. The imbalances -2 at z and -1 at y ensure that this is the case. We are now able to implement the first rule for this scenario.

$$\begin{gathered} n(-(s(s(0))), x_0, x_a, n(-(s(0)), x_1, x_b, n(d_3, x_2, x_c, x_3))) \rightarrow \\ n(0, n(0, x_0, x_a, x_1), x_b, n(d_3, x_2, x_c, x_3)) \end{gathered} \tag{3.1}$$

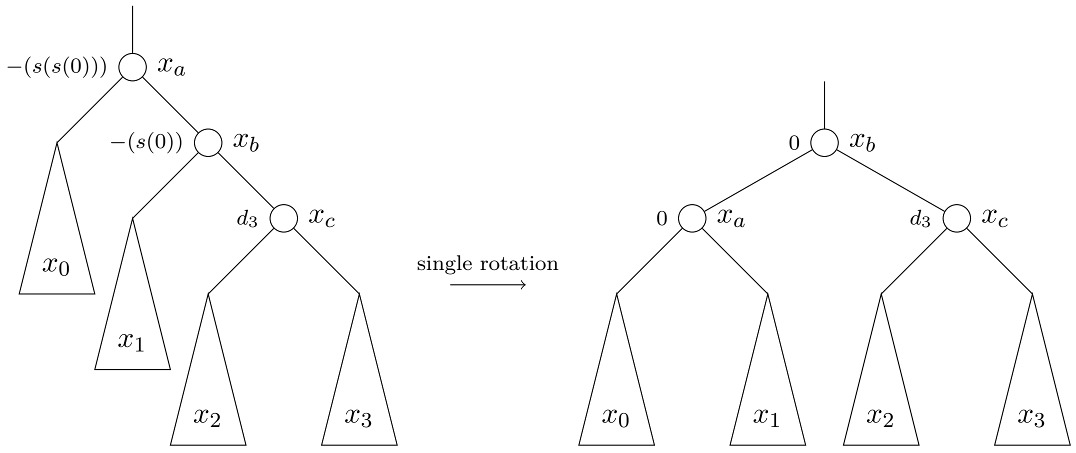

Figure 3.5: Trinode-rotation with TRS terms: Case 1

To show that the new imbalances after the rotation are correct, the initial structure is examined. Without knowing the actual heights, a relation can be used between the imbalances in order to find out the new ones.

Definition 3.8. We introduce a function that yields the imbalance of a node and is defined as following:

$$\begin{aligned} D &: Node \rightarrow \mathbb{Z} \\ n(d, x, a, y) &\mapsto d. \end{aligned}$$

Definition 3.9. Let H be a function that yields the height of a node so:

$$H\colon Node \to \mathbb{N}.$$

Lemma 3.10. *The rewrite rule* (3.1) *restores the AVL property of the three involved nodes so imbalances* $D(x_a) = -2$*,* $D(x_b) = -1$ *become* $D(x_a) = 0$*,* $D(x_b) = 0$ *while* $D(x_c)$ *does not change.*

Proof of Lemma 3.10. We want to show the correctness of the imbalance changes. Since the node x_c is not rotated, the subtrees stay the same, hence the imbalance of this node remains untouched. For the other two nodes, we consider their old structure and determine relative heights to apply them to the new structure.
Imbalances:

$$D(x_a) = H(x_0) - H(x_b) = -2 \tag{3.2}$$

$$D(x_b) = H(x_1) - H(x_c) = -1 \tag{3.3}$$

$$D(x_c) = H(x_2) - H(x_3) = d_3 \tag{3.4}$$

Heights:

$$(3.3) \Rightarrow H(x_c) = H(x_1) + 1 \tag{3.5}$$

$$(3.3) \wedge (3.5) \Rightarrow H(x_b) = H(x_c) + 1 = H(x_1) + 2 \tag{3.6}$$

$$(3.2) \Rightarrow H(x_b) = H(x_0) + 2 \tag{3.7}$$

$$(3.6) \wedge (3.7) \Rightarrow H(x_1) + 2 = H(x_0) + 2 \Rightarrow H(x_1) = H(x_0) \tag{3.8}$$

New Heights:

$$(3.8) \Rightarrow H(x_a) = H(x_1) + 1 \tag{3.9}$$

New Imbalances:

$$(3.8) \Rightarrow D(x_a) = H(x_0) - H(x_1) = 0 \tag{3.10}$$

$$\begin{aligned}(3.9) \wedge (3.5) \Rightarrow D(x_b) &= H(x_a) - H(x_c) = \\ &= (H(x_1) + 1) - (H(x_1) + 1) = 0\end{aligned} \tag{3.11}$$

□

3.4.2 Case 2

$$n(+(s(s(0))), n(+(s(0)), n(d_3, x_0, x_a, x_1), x_b, x_2), x_c, x_3) \rightarrow \\ n(0, n(d_3, x_0, x_a, x_1), x_b, n(0, x_2, x_c, x_3)) \tag{3.12}$$

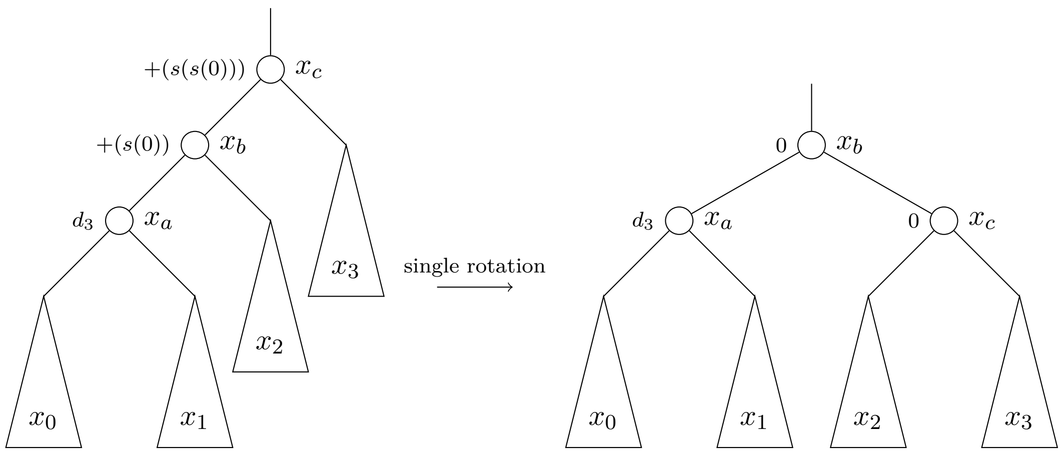

Figure 3.6: Trinode-rotation with TRS terms: Case 2

Lemma 3.11. *The rewrite rule* (3.12) *restores the AVL property of the three involved nodes so imbalances $D(x_c) = +2$, $D(x_b) = +1$ become $D(x_c) = 0$, $D(x_b) = 0$ while $D(x_a)$ does not change.*

Proof of Lemma 3.11. See Appendix A.1. □

3.4.3 Case 3

$$\begin{aligned} &n(-(s(s(0))), x_0, x_a, n(+(s(0)), n(-(s(0)), x_1, x_b, x_2), x_c, x_3)) \to \\ &\qquad n(0, n(+(s(0)), x_0, x_a, x_1), x_b, n(0, x_2, x_c, x_3)) \end{aligned} \tag{3.13}$$

$$\begin{aligned} &n(-(s(s(0))), x_0, x_a, n(+(s(0)), n(0, x_1, x_b, x_2), x_c, x_3)) \to \\ &\qquad n(0, n(0, x_0, x_a, x_1), x_b, n(0, x_2, x_c, x_3)) \end{aligned} \tag{3.14}$$

$$\begin{aligned} &n(-(s(s(0))), x_0, x_a, n(+(s(0)), n(+(s(0)), x_1, x_b, x_2), x_c, x_3)) \to \\ &\qquad n(0, n(0, x_0, x_a, x_1), x_b, n(-(s(0)), x_2, x_c, x_3)) \end{aligned} \tag{3.15}$$

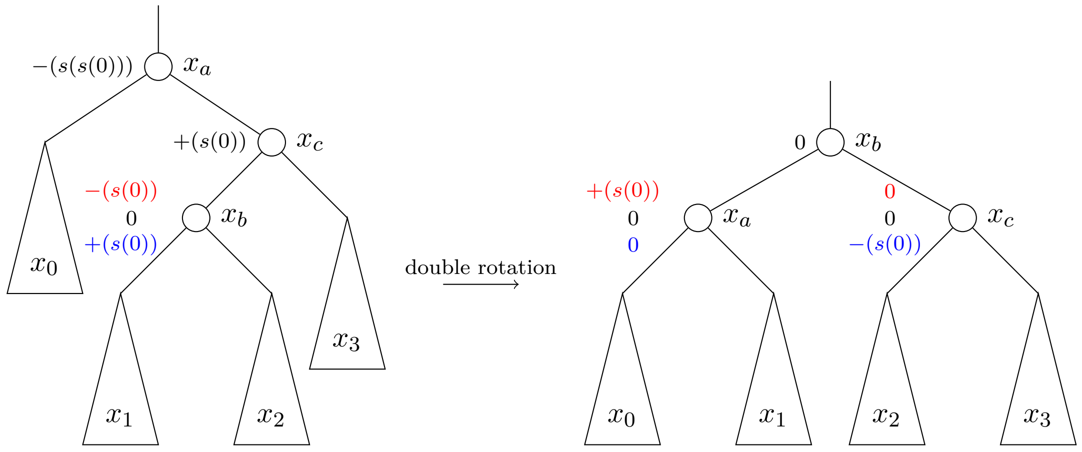

Figure 3.7: Trinode-rotation with TRS terms: Case 3

Lemma 3.12. *The rewrite rules* (3.13), (3.14) *and* (3.15) *restore the AVL property of the three involved nodes so imbalances* $D(x_a) = -2$, $D(x_b) = (-1, 0, +1)$ *and* $D(x_c) = +1$ *become* $D(x_a) = (+1, 0, 0)$, $D(x_b) = 0$ *and* $D(x_c) = (0, 0, -1)$*.*

Proof of Lemma 3.12. See Appendix A.2. □

3.4.4 Case 4

$$\begin{aligned} n(+(s(s(0))), n(-(s(0)), x_0, x_a, n(-(s(0)), x_1, x_b, x_2)), x_c, x_3) \to \\ n(0, n(+(s(0)), x_0, x_a, x_1), x_b, n(0, x_2, x_c, x_3)) \end{aligned} \tag{3.16}$$

$$\begin{aligned} n(+(s(s(0))), n(-(s(0)), x_0, x_a, n(0, x_1, x_b, x_2)), x_c, x_3) \to \\ n(0, n(0, x_0, x_a, x_1), x_b, n(0, x_2, x_c, x_3)) \end{aligned} \tag{3.17}$$

$$\begin{aligned} n(+(s(s(0))), n(-(s(0)), x_0, x_a, n(+(s(0)), x_1, x_b, x_2)), x_c, x_3) \to \\ n(0, n(0, x_0, x_a, x_1), x_b, n(-(s(0)), x_2, x_c, x_3)) \end{aligned} \tag{3.18}$$

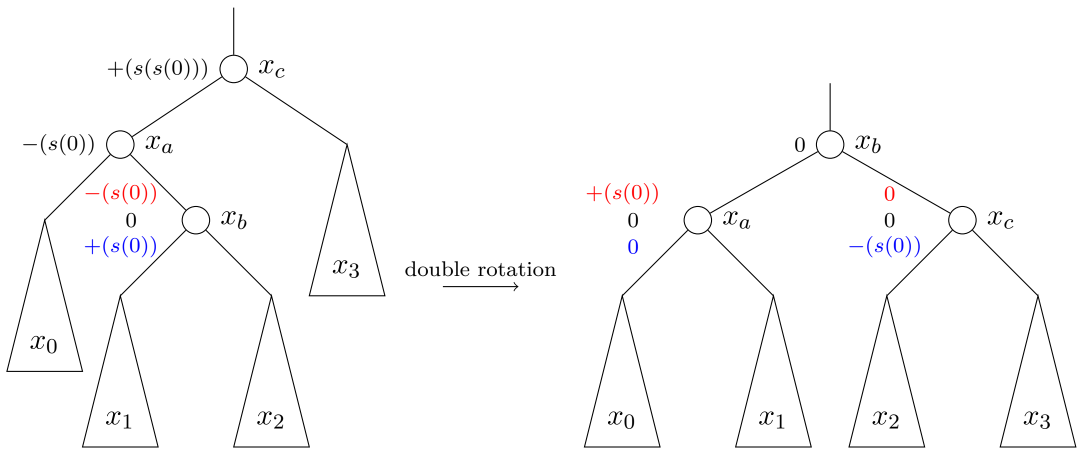

Figure 3.8: Trinode-rotation with TRS terms: Case 4

Lemma 3.13. *The rewrite rules* (3.16)*,* (3.17) *and* (3.18) *restore the AVL property of the three involved nodes so imbalances* $D(x_a) = -1$*,* $D(x_b) = (-1, 0, +1)$ *and* $D(x_c) = +2$ *become* $D(x_a) = (+1, 0, 0)$*,* $D(x_b) = 0$ *and* $D(x_c) = (0, 0, -1)$*.*

Proof of Lemma 3.13. See Appendix A.3. □

In total, there are eight rules for a trinode-rotation. With this set of rules, insertion can be done and a correct trinode-rotation is also performed in order to maintain the structure. But these eight rules for restructuring are not enough when considering the deletion of a node.

Example 3.14. After the deletion of the node 7, the tree is left in an unrecognized pattern.

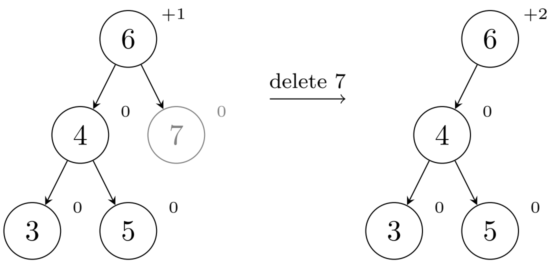

Figure 3.9: Delete leads to no pattern being matched

The trinode rules are extended so these scenarios are covered as well.

$$n(-(s(s(0))), x_0, x_a, n(0, x_1, x_b, n(d_3, x_2, x_c, x_3))) \rightarrow \\ n(+(s(0)), n(-(s(0)), x_0, x_a, x_1), x_b, n(d_3, x_2, x_c, x_3)) \tag{3.19}$$

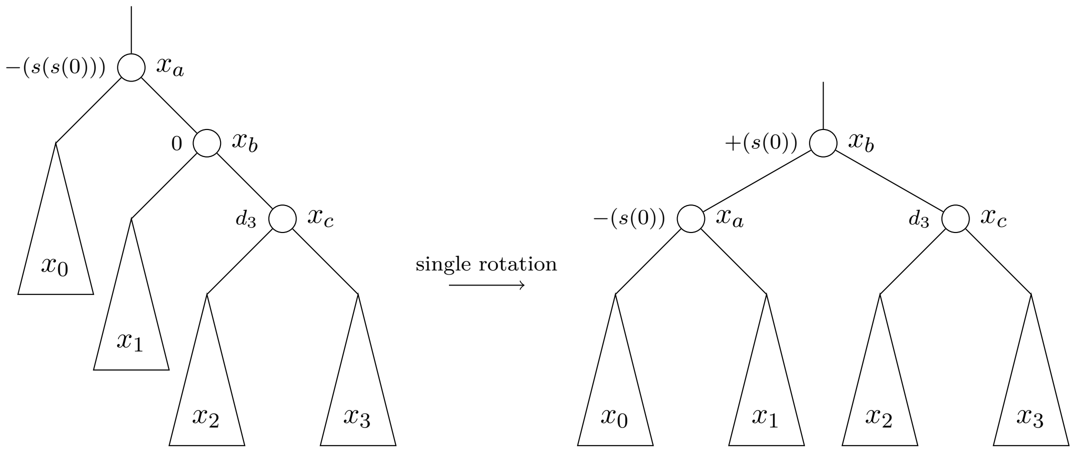

Figure 3.10: Trinode-rotation with TRS terms: Special Case 1

$$n(+(s(s(0))), n(0, n(d_3, x_0, x_a, x_1), x_b, x_2), x_c, x_3) \rightarrow n(-(s(0)), n(d_3, x_0, x_a, x_1), x_b, n(+(s(0)), x_2, x_c, x_3)) \tag{3.20}$$

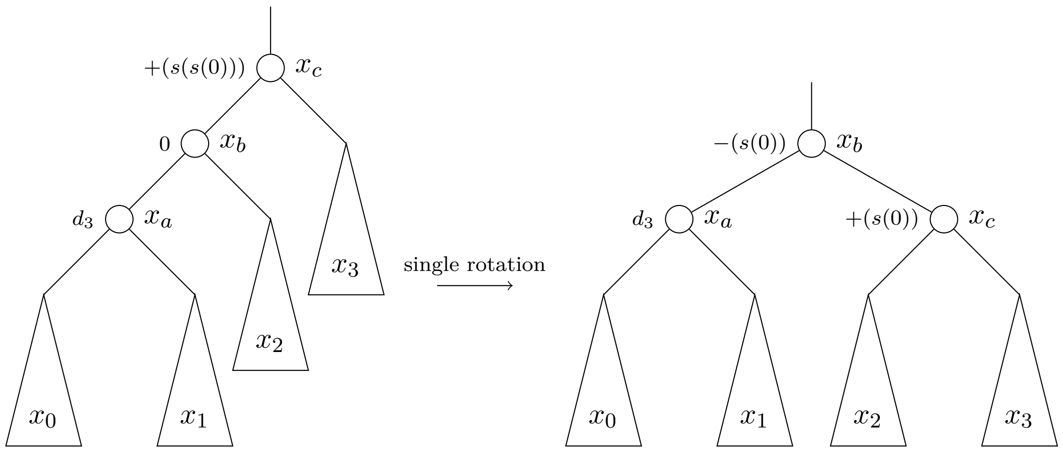

Figure 3.11: Trinode-rotation with TRS terms: Special Case 2

With the rule (3.20) the tree in Figure 3.9 can be fixed. Now the TRS with these rules is able to perform arbitrary inserts and deletes.

3.5 Sequential Implementation

This TRS is almost identical to the TRS in Section 3.6. The rules that differ from the parallel TRS can be seen in Appendix B.2.

The main intention behind the sequential implementation is to mimic the operations for an AVL tree and their behaviors as shown in textbooks [8, 10.2 AVL Trees]. For this, we want to be able to perform an arbitrary number of inserts and deletes, in any order on the TRS. Compared to the parallel version, we only allow one operation at a time to be executed on the term. This is done by treating the operations in a *FIFO* (first in, first out) manner, where the first operation acquires a lock. This lock ensures that only one operation at a time can be executed on the search term, while other operations are queued up and waiting until it is their turn. To understand what we mean by *queue* and *lock* see Example 3.15 below.

To release the lock, a propagation call from either an insert or delete will eventually transform itself into an unlock call, which is propagated upwards to the root. This happens when the propagation condition is interrupted (e.g. insert and the new imbalance is 0). If the operation propagation reaches the root, the lock of the tree will be released and the propagation will be omitted at the same time. If an operation fails (e.g. delete node cannot be found), it will propagate a release call for the lock right away.

Example 3.15. For the operations *insert* 10, *insert* 5, *delete* 7 and an initial search term $\mathit{tree}(n(0, \bullet, 9, \bullet))$ we get following term: $\mathit{delete}(7, \mathit{insert}(5, \mathit{insert}(10, \mathit{tree}(n(0, \bullet, 9, \bullet)))))$.

After one rewrite step, the term is locked and no more operations are applicable until the operation past the lock is done: $\mathit{delete}(7, \mathit{insert}(5, \mathit{tree}(\mathit{lock}(\mathit{insx}(10, n(0, \bullet, 9, \bullet))))))$.

Once an operations is done, an unlock propagation will release the lock so other operations can follow: $\mathit{delete}(7, \mathit{insert}(5, \mathit{tree}(n(-1, \bullet, 9, n(0, \bullet, 10, \bullet)))))$.

The sequential TRS models a *call by value* behavior.

3.6 Parallel Implementation

Earlier mentioned concepts were able to be implemented as rewrite rules and the fully implemented TRS can be seen in Appendix B.1.

When we say parallel, we now want to be able to simultaneously perform all operations on the TRS without having to wait for the release of a lock.

Comparison operations are atomic, meaning when values are compared, it is not possible for the node to get another value during this process. If not taken care of, it might happen that a value comparison is initiated and executed at a later stage but at some point, the node where it occurred changes its structure and value leading to a bogus result. This measure was a direct result of a previous error in older versions of the TRS.

This is also true for the sequential implementation, but it can be completely dropped, as we are only dealing with one operation at the time there, which ensures that the scenario described above cannot appear. We chose to keep the rules in the sequential TRS to minimize the difference to the parallel TRS.

Example 3.16. We want to show the urge of atomicity. Here, a comparison operation is deployed but the node-value is changed, possibly by a restructuring coming from the subtree x_0 or x_1, from 4 to 15 before the comparison is done.

At the time of the comparison, the insert operation would have to continue in the right subtree. But if we would compare with 15, the insert should continue in the left subtree because of the order $\geq$. However, this is not the case, so we end up in the wrong subtree For an initial term, we get following rewriting sequence:

$$\mathit{insx}(8, n(d, x_0, 4, x_1)) \to \mathit{insert1}(8, n(d, x_0, 4, x_1), \mathit{geq}(8, 4))$$

$$\xrightarrow{*} \mathit{insert1}(8, n(d, x_0, 15, x_1), \mathit{geq}(8, 4)) \to \mathit{insert1}(8, n(d, x_0, 15, x_1), \mathit{true})$$

$$\to n(d, x_0, 15, \mathit{insx}(8, x_1))$$

To overcome this issue, atomicity is enabled by temporarily rewriting the node term into something else, so no other rewrite rules can match and interfere until the comparison is done. So instead of $\to \mathit{insert1}(8, n(d, x_0, 4, x_1), \mathit{geq}(8, 4))$ we would get $\to \mathit{insert1}(8, n_8(d, x_0, 4, x_1), \mathit{geq}(8, 4))$.

3.6.1 Search

The rewrite rules for a search as described in Section 3.1 can be found in Appendix B.1.1.

3.6.2 Insert

This part utilizes mainly Section 3.2 and the implementation can be seen in Appendix B.1.2. One remark is that for an initial imbalance of ± 1 and an occurrence of a propagation symbol, every trinode case is directly matched to and locked so the rotation can be performed. This ensures that no special treatment for other occurring symbols within the trinode call is needed.

If for instance, a trinode-rotation on three selected nodes needs to be done and one of the nodes has a propagation symbol from another operation, rotating the nodes could cause bogus imbalances. Refer to Example 3.20 for an illustration of this issue.

Another reason is to keep the number of rules minimal while also making it easier to model.

3.6.3 Delete

Section 3.3 is fully integrated as a set of rules and can be seen in Appendix B.1.3. As with the previously mentioned insert, every comparison here is also locked down for the same reasons. Propagation handling is done in the exact same way with according rules.

Locking also occurs if a node needs to be deleted. Its current value will be erased and replaced by a non-numeric placeholder which does not allow any comparison operations. So any operation seeking passage will be held at this point until the delete operation is complete and the placeholder is replaced by a numeric value.

This prevents the operation from being continued in order to avoid the comparison with the node-value, which would yield a bogus result.

Example 3.17. Here the insert of 10 can only continue if the placeholder value is replaced by a numeric value: $\mathit{insert}(10, n(0, x_0, \mathit{placeholder}, \mathit{subTree}(x_1)))$.

This does not, however, mean that any operation that already has passed this node will be hindered in its ability to perform further action.

One scenario which could possibly harm the search terms validity is the following: A delete on a node is performed and its inorder successor needs to be found. Now, what happens if this inorder successor is somehow changed and targeted by other operations?

Example 3.18. We want to illustrate what happens if another delete call reaches the inorder successor of a deleted node before the replacement call. As long as other operations reach a node before a replacement call, the correct execution of operations is granted.

Let $\square$ be the placeholder for the deleted node-value and let $\star$ be the latest inorder successor of it. Here, *del* is the delete operation, *get* finds and returns the inorder successor and *ret* is the returning call of *get*, meaning it delivers the value to the target node.

The operation $del(9)$ is being halted at the node with $\square$ until a replacement call returns another value to replace $\square$. Before the lockdown, the operation $del(8)$ passed the deleted node and is being executed in the subtree of the node with $\square$.

The operation $del(8)$ is being performed successfully and $get(\star)$ targets a new inorder successor. One thing to keep in mind is, that $get(\star)$ is not an explicit call with an explicit target. It tries to reach the outermost node, which is not a leaf, to get the inorder value. This is why it seems that the update of the position $\star$ happens automatically, where in reality this is just a consequence of the rewrite rules that model the replacement call.

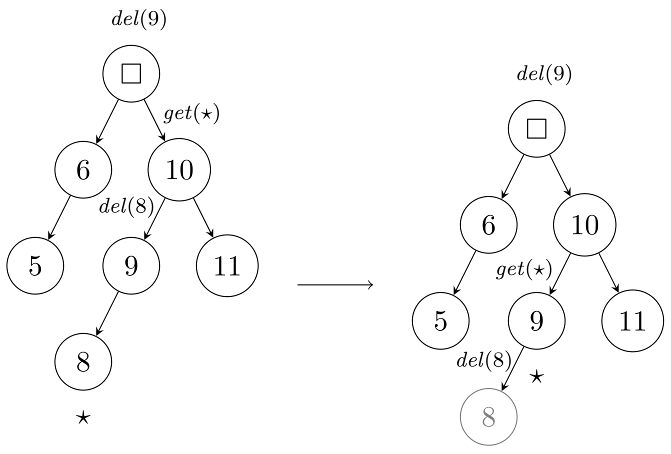

Figure 3.12: Correct execution-order of delete and replacement - Part 1

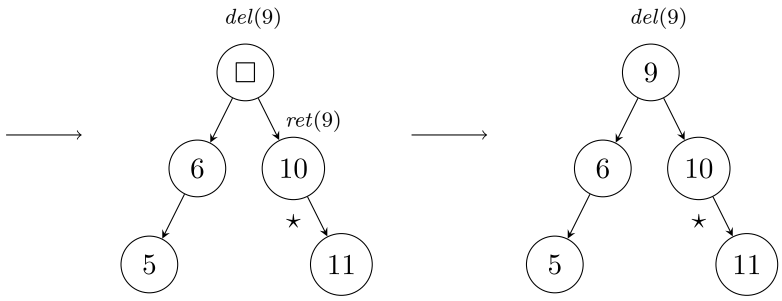

Figure 3.13: Correct execution-order of delete and replacement - Part 2

If the replacement call surpasses the deletion call, the inorder successor has been saved from being deleted. This is unacceptable because a performed delete operation would fail. Hence, we have to make sure that a replacement call never surpasses a delete call.

Example 3.19. Here, we show how a replacement call saves its inorder successor from being deleted. Again, *del*(9) needs to wait until □ is replaced by some value. The call *get*(⋆) surpasses the delete call *del*(8) and saves the value 8.

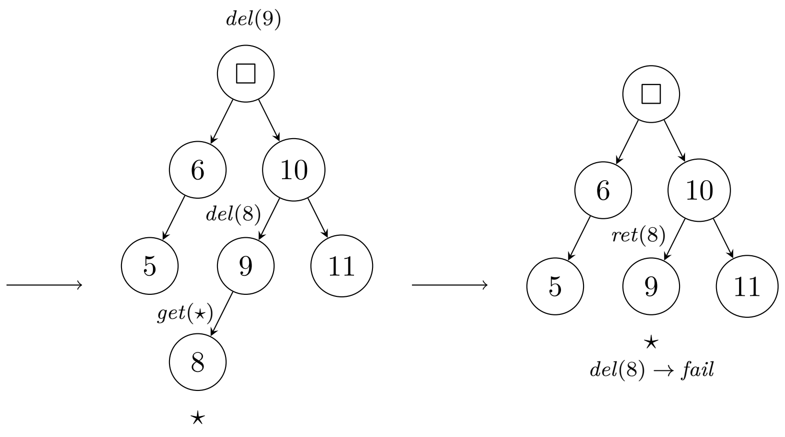

Figure 3.14: Problematic execution-order of delete and replacement - Part 1

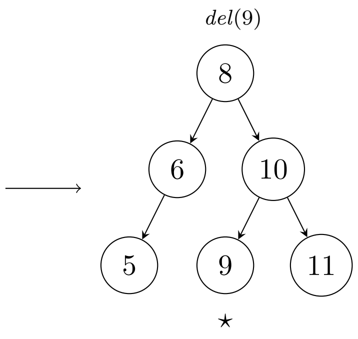

Figure 3.15: Problematic execution-order of delete and replacement - Part 2

It is also important to mention that the case distinction process seen in Definition 2.25 is an atomic process. This means when a scenario is matched, a proper replacement will happen.

When trying to find a replacement node in the right subtree, we need to make sure that the subtree does not vanish due to other delete calls. In case a scenario is matched, the replacement call will not allow any operation calls to surpass this node, hence the vanishing of the subtree cannot happen.

3.6.4 Trinode

The rules for trinode-rotation as seen in Section 3.4 are actually explicit calls, in order to avoid termination failure. Refer to Appendix B.1.4 for the implementation. The trinode calls are also wrapped in a function in order to check if further propagation is needed for the given operation.

The consumption of a propagation symbol and the lockdown for a trinode-rotation happens in one rewrite step. This ensures that the involved nodes for the rotation are free of any propagation symbols, which as mentioned earlier could cause false imbalances.

Example 3.20. We want to demonstrate why multiple occurrences of propagation symbols in a trinode-rotation application can lead to false results. The node 1 is deleted and the node 4 is inserted. The propagations of these operations reach the node 3.

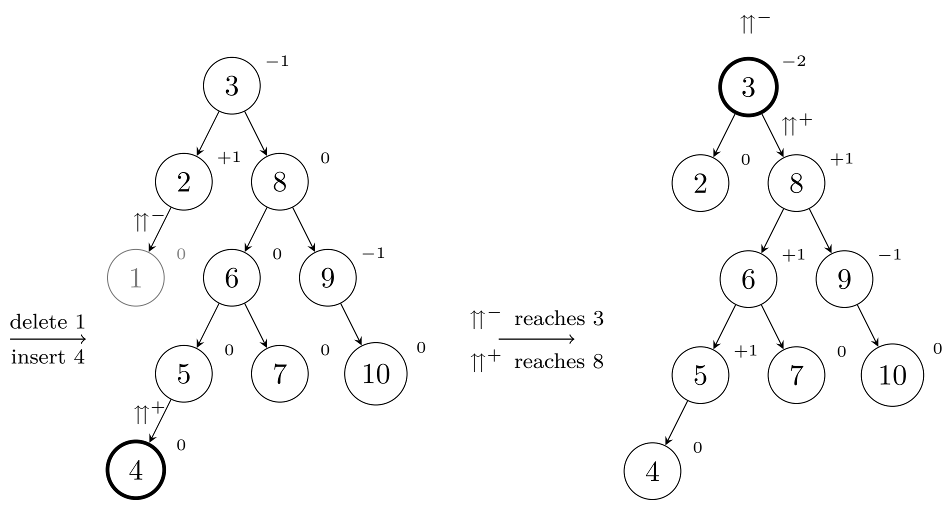

Figure 3.16: Trinode-rotation false imbalance - Part 1

Now, the node with the value 3 could consume the propagation symbol to its right and update its imbalance to -3. This is not allowed due to our previously mentioned restraint, which prohibits any imbalances, even if just temporary, which are greater than $+2$ or less than -2.

Performing the rotation and continuing the propagation is a viable option, but the propagation $\upuparrows^+$ becomes invalid. We cannot push the propagation downwards since the node 8 already consumed it. We will just leave the symbol $\upuparrows^+$ at the same spot, term wise, and rotate as seen in Figure 3.7 with the rule (3.13).

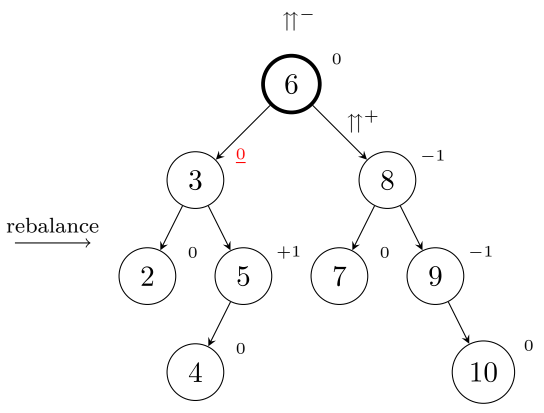

Figure 3.17: Trinode-rotation false imbalance - Part 2

The imbalance of the node 3 is a direct result of the rewrite rule and is invalid. This leads to the conclusion, that not only the propagation symbol becomes invalid, but that the rotation also yields a false result.

So to prevent this, in case of a possible trinode-rotation, the consumption of the propagation symbol happens if and only if a trinode-rotation on the three involved nodes is applicable. If that is not the case, the symbol will be consumed at a later point.

In Figures 3.18 and 3.19, one can see the correct handling of multiple occurrences of propagation symbols.

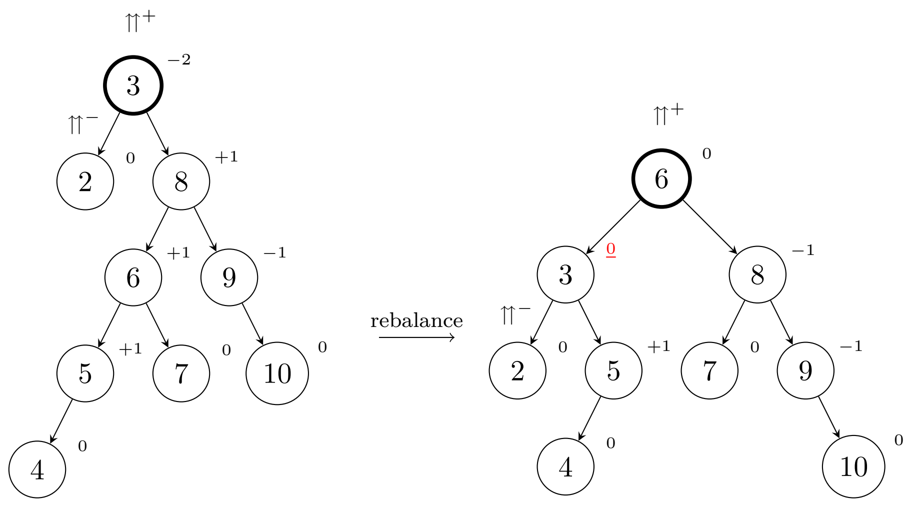

Figure 3.18: Trinode-rotation correct handling of propagation symbols - Part 1

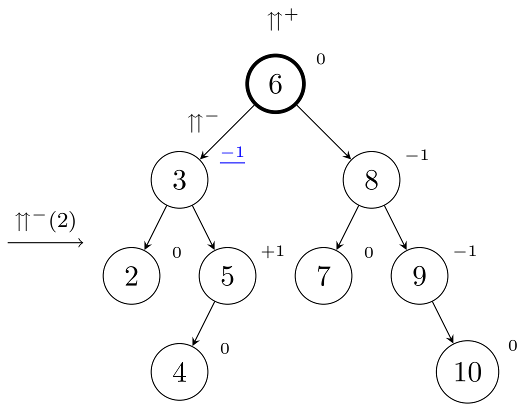

Figure 3.19: Trinode-rotation correct handling of propagation symbols - Part 2

The imbalance of the node 3 is temporarily incorrect, but it is later fixed by consuming the propagation symbol.

3.6.5 Arithmetic

Since we are dealing with natural numbers represented as function calls of s on 0, we want the ability to perform order and equivalence tests. This is done by a small set of rewrite rules that look similar to $eq(s(x), s(y)) \to eq(x, y)$. This rule removes one s on both sides and continues to do so, until one side reaches 0. The rewrite rules can be found in Appendix B.1.6.

3.6.6 Collision Handling

To enable multiple operations at the same time without having to lock the entire tree, a set of rules has been introduced to deal with the eventual collision of function calls.

Mainly the propagation call of an operation and the replacement call of a delete operation need additional rules so they can pass through eventual blockades. To see all the rules refer to Appendix B.1.5.

3.7 Performance Issues of Arithmetic

A problem that arises from Subsection 3.6.5 is the fact that for greater numeric values and a greater number of nodes, the rewrite steps increase drastically. To avoid this issue, numbers as constants could be introduced. That would cause the number of rewrite rules to rise. In fact, we need to be able to generate rewrite rules automatically for each constant that is added.

An issue that directly follows from this is the time loss on locking mechanisms. For example, since the term $insert(x, n(d_1, x_1, s(x_a), x_2), \geq (x, s(x_a)))$ is locked in order to ensure correct comparison by $\geq$, no other insert operation can deploy its comparison. Having constant numbers would reduce the duration of the lockdown to the duration of a single rewrite step.

For greater numeric values, this bottleneck becomes quite significant. Let us assume that we are able to perform 10^6 rewrite steps per second. Inserting 10 numbers above the value 10^6 would result, in the worst case, in a time loss of 10 seconds. And this is just the time loss at one single node term, without considering the rest of the search term.

For the complexity-analysis and the runtime-measurement we will abstract from this and assume that every numeric value of a node is 0.

3.8 Termination

The goal was to design a TRS that terminates and that the normal form of every term is a correct representation of an AVL tree. Every operation on the term is in a downwards manner and once an operation reaches a target node, rotations can only occur above this node. This means that the subtrees of a node are not affected and that a propagation call will eventually reach the root and vanish. A direct result of this is that terminating behavior is modeled by the rewrite rules, hence the TRS is in fact terminating.

Termination for both the parallel and sequential TRS was proven by TTT2 [9] and AProVE [1]. The proofs themselves utilize various methods such as *polynomial interpretation* [18, Section 6.2.2], *dependency pairs* and some others. They can be found as HTML outputs in a repository[2]. The size of the files is between 3 MB and 2 MB, meaning the proofs are quite large. Proving the termination of our term rewrite systems by hand would be unfeasible.

3.9 Complexity

As mentioned in Section 3.7, the greater the values of the nodes get, the more execution steps it requires to compare them with each other. For the analysis, we abstract from this.

We want to describe the complexity by the number of rewrite steps in relation to the number of nodes n. This is reasonable because we are dealing with linear rewrite steps. This means that variables in a term are not duplicated by the rewrite rules.

Example 3.21. We want to demonstrate how many rewrite steps an insert takes. Suppose we want to insert 8 and have the term $n(-1, n(0, \bullet, 2, \bullet), 4, n(0, n(0, \bullet, 6, \bullet), 7, n(0, \bullet, 9, \bullet)))$ that is represented by the illustration below.

[2]`https://git.uibk.ac.at/csat1763/SearchTerms/tree/master/Termination`

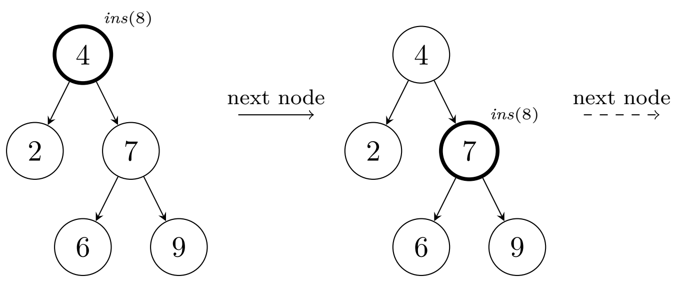

Figure 3.20: Rewrite steps needed for insert

When inserting a node, we go from node to node in order to find its right place. By *next node* above, we mean that the comparison at the previous node has finished and the correct subtree for further traversal has been chosen. We count the steps for one iteration of this process.

The rewrite steps at the first node are following:

$$\mathit{insx}(8, n(-1, n(-1, n(0, \bullet, 2, \bullet), 4, n(0, n(0, \bullet, 6, \bullet), 7, n(0, \bullet, 9, \bullet))))$$

$$\to \mathit{insert1}(8, n7(-1, n(0, \bullet, 2, \bullet), 4, n(0, n(0, \bullet, 6, \bullet), 7, n(0, \bullet, 9, \bullet))), \mathit{geq}(8, 4))$$

$$\to \mathit{insert1}(8, n7(-1, n(0, \bullet, 2, \bullet), 4, n(0, n(0, \bullet, 6, \bullet), 7, n(0, \bullet, 9, \bullet))), \mathit{true})$$

$$\to n(-1, n(0, \bullet, 2, \bullet), 4, \mathit{insx}(8, n(0, n(0, \bullet, 6, \bullet), 7, n(0, \bullet, 9, \bullet))))$$

So at each node we need at least 3 rewrite steps. For deleting, a similar number of steps is needed.

The number of rewrite steps for one iteration is expressed by a constant c. Considering the number of nodes n of an AVL tree, we know that the maximum height of the tree is $2 \cdot \log_2(n) + 2$ [8, Justification of Proposition 10.2]. This also directly corresponds to the height of the search term of this AVL tree. For the insert in a search term of an AVL tree with c and n, we get a maximum number $c \cdot (2 \cdot \log_2(n) + 2)$ of rewrite steps. If we express the runtime-complexity we get $\mathcal{O}(c \cdot (2 \cdot \log_2(n) + 2))$. Using $\log$ instead of $\log_2$ and reducing the function with respect to the big $\mathcal{O}$ notation, the complexity for an insert becomes $\mathcal{O}(\log n)$ effectively.

Since searching is included in the main routine of an insert, we can describe the number of rewrite steps in a similar way. Applying the same calculation, as previously, gives us the same runtime-complexity of an insert for a search. This also applies to deleting, meaning the main routine of a delete also has the same complexity.

Deleting requires an additional complexity of $\mathcal{O}(\log m)$ where m is the number of nodes in the subtrees of the the target node. When we add these two routines together we get $\mathcal{O}(\log m) + \mathcal{O}(\log n) = \mathcal{O}(\log n + \log m)$. Let us assume that for both routines we have the same number of nodes, namely n. Now we get $\mathcal{O}(\log n + \log n) = \mathcal{O}(2 \cdot \log n)$ which can be reduced to $O(\log n)$.

The restructuring does not depend on n, thus yielding a runtime-complexity of $\mathcal{O}(1)$.

Since every delete and insert comes with a propagation, the propagation itself, in worst case, can cause the same number of rewrite steps as the delete or the insert, which leads to $\mathcal{O}(2 \cdot \log n)$. A constant factor does not affect the complexity of $\mathcal{O}(\log n)$ which leaves us with this as the worst case for all operations.

3.10 Correctness

We want all permanent and temporary imbalances to be elements in the range of $[-2, 2]$.

Also, we need to ensure the correct termination of the TRS, so we are able to obtain an AVL tree from the search term. Lastly, the inorder list of a search term must not be changed by any rule. Only a delete or an insert either adds an element to the list or removes one from it, at the correct spot.

If we can show the correctness for the parallel implementation, it is safe to assume that the same will hold for the sequential implementation because the latter contains a subset of rules of the first one.

At this point we want to recall the balancedness $\mathcal{B}$ from Definition 2.22 and refer to the rewrite rules of the TRS in Appendix B.1.

If a node has an imbalance of ± 2, any propagations targeting this node will be halted until a trinode-rotation is complete.

Lemma 3.22. *When performing arbitrary operations, the temporary imbalances will always be elements of the set* $\{-2, -1, 0, 1, 2\}$.

Proof of Lemma 3.22. We want to show that we cannot reach any temporary value for any imbalance that is not in the set $F = \{-2, -1, 0, 1, 2\}$.

This follows from the rules of the TRS. Every rule, where an imbalance modification happens, changes the imbalance that is element of F to a new element of F. Since the rules directly map to constant imbalances, meaning neither an addition nor a subtraction of the imbalances occurs, an explicit rewrite rule would be needed to transform an imbalance to a value outside of the set F. No such rule exists in Subsection B.1.4, hence all imbalances are element of F. □

Theorem 3.23. *The global balancedness $\mathcal{B}$ of an AVL tree is restored by the rewrite rules that model a trinode-rotation.*

Proof of Theorem 3.23. To be able to rely on trinode-rotation to do the restoring of property $\mathcal{B}$, we need to ensure that every possible combination of node structures and imbalances is covered by the set of rules.

As seen in Subsections 3.4.1–3.4.4 we have all the rules needed for every combination. With Lemma 3.22 we also know that imbalances can only reach values of $\{-2, -1, 0, 1, 2\}$. Once a node reaches -2 or 2, due to the consumption of a propagation symbol, a trinode-rotation is applied immediately. Since these rules are very strict (see Subsection 3.6.4), the rotation yields a locally balanced subtree, following from Lemmas 3.10–3.13. We know that once a trinode-rotation restores the balancedness of a subtree, another violation can only occur above this subtree. Since AVL trees are rooted we know that violations cannot go beyond the root.

As long as an imbalance of a node is ± 2, a rewrite rule is applicable. The TRS is terminating, meaning we will always reach a normal form. But if a rewrite rule is applicable, we cannot be in normal form. Hence, $\mathcal{B}$ is always restored.

□

Let us recall the function I from Definition 2.30 that yields a list of elements of a search term. For the search term of an AVL tree we can say that the list is sorted by the order $\geq$.

Lemma 3.24. *Applying rewrite rules that model a trinode-rotation preserve the inorder list of a search term.*

Proof of Theorem 3.24. If $\ell \to r$ is a rewrite rule that models a trinode-rotation, then $I(\ell) = I(r)$. Let us consider the rule (3.1).

Let $\ell = n(-(s(s(0))), x_0, x_a, n(-(s(0)), x_1, x_b, n(d_3, x_2, x_c, x_3)))$
and $r = n(0, n(0, x_0, x_a, x_1), x_b, n(d_3, x_2, x_c, x_3))$.

We get $I(\ell) = (I(x_0), x_a, I(x_1), x_b, I(x_2), x_c, I(x_3))$
and $I(r) = (I(x_0), x_a, I(x_1), x_b, I(x_2), x_c, I(x_3))$.

So, indeed $I(\ell) = I(r)$ meaning this particular rewrite rule preservers the inorder list of the term.

The same analogy can be applied for the remaining trinode rewrite rules.

□

Lastly, we want to relate the operations on the search terms to operations on inorder lists. The operations search, insert, delete are expressed by following symbols respectively:

- For search terms: s_t, i_t, d_t.
- For inorder lists: s, i, d.

When we perform a search on a search term and construct the inorder list of that result by I, we want the same exact result for searching in the inorder list of the search term. This is expressed by: $I(s_t(v,t)) = s(v, I(t))$. We want the operations to behave as described in Section 2.4, where operations in different systems lead to equivalent representations.

Similar to the search, the delete is expressed by $I(d_t(v,t)) = d(v, I(t))$ and the insert by $I(i_t(v,t)) = i(v, I(t))$.

3.11 Rewriting Strategy and Optimization

The application of rewrite rules can happen in different ways. Some part of a term is selected for a rule application based on the used rewriting strategy.

A significant increase in performance could be achieved by using the maximal strategy [10, Chapter 7], which performs as many rewrite steps as possible, at once.

The rules

$$up(up-, up(up+, x_0)) \to x_0$$
$$up(up+, up(up-, x_0)) \to x_0$$

allow better performance, because they first stop the propagations and then cancel possible trinode-rotations. The left terms of the rules also match with other rules, which means it is not certain which of the rules will be applied. Introducing a rewriting strategy that prioritizes the rules above over other rules, would be necessary to ensure improvement.

We only used strategies provided by the used engines and did not further investigate in introducing new ones.

3.12 Parallel Execution

The TRS described in Section 3.6 is able to be executed in parallel, assuming the running engine is able to do so. If the engine is not able to run this in parallel, we can duplicate the engine and introduce an external process to monitor the term and manage temporary splits of execution.

When a bulk of operations is deployed, the first step is to have two separate processes for each subtree of the root, splitting the tree. Now, operations can truly be performed in parallel in the respective process of the subtree. When an operation traverses the node-terms, the splitting is continued up to a certain degree, which can be defined by the depth of the term. This should prevent the creation of too many processes, which would cause too much overhead at synchronization points.

Even though the operations are now being executed in parallel to a certain degree, the search-term needs to be updated and synchronized at some point. This must happen if a propagation reaches the outer boundaries of a process, meaning it is passed from one process to another. If the propagation does not cause any rotation, it is just passed from process to process. Once a propagation causes a rotation, the processes of the involved nodes need to synchronize and update their subtrees to the newly rotated term.

3.13 TRS Statistics

The parallel TRS in Appendix B.1 contains 104 rewrite rules and 39 function symbols, whereas the full sequential version of the TRS hinted in Appendix B.2 has 96 rules and 41 function symbols.

4 Tools for Development

Various tools have been implemented and used to make the implementation of the term rewrite systems easier. All of them can be found in the provided repository of the university [4].

4.1 TRS Engine

Running the TRS is made possible by implementing it in a logical environment named *Maude System* [16], which is a tool based on a framework for rewriting logic [5]. Fortunately, the syntax of the tool is almost identical to the one, that is used to implement the TRS. Once all the rules, operations, and types are defined, running the TRS in Maude is done by initiating a command like

```
  $ rew insert(s(0),tree(n(0,leaf,0,leaf))) .
```

which yields

```
rewrites: 7 in 0ms cpu (0ms real) (~ rewrites/second)
result node: tree(n(-(s(0)),leaf,0,n(0,leaf,s(0),leaf)))
```

as a result. The command rew runs a leftmost-outermost rewriting strategy on the given term.

Since writing a command like this, especially when inserting bigger numbers, can get quite tedious, Shell scripts and Java programs have been written, that perform an operation on the search term.

The script is called with arguments that indicate what the desired action is (e.g. insert 10, delete 5 and insert 20). Then, a Java program is executed that transforms the numbers in sequences of *s* calls and creates a Maude command with the given input. It is then executed in Maude and the output of the execution is trimmed using another Java program, so only the search term is displayed.

This all happens for a given input file which contains the search term. After the execution, the current file is rewritten to the latest search term and another file is generated that contains the search term before the execution.

All scripts can be found in the folder *Maude*[1].

[1] `https://git.uibk.ac.at/csat1763/SearchTerms/tree/master/TRS-Engine/Maude`

4.2 Parsing Terms

In order to allow validation tests and the transformation into an actual graphical search tree, parsing the search term is essential. By utilizing a Java parsing framework called *ANTLR* [17], further inspection of any search terms is made possible. Once a self-defined grammar file is created, the framework will automatically generate all Java classes needed for the parsing.

The base grammar and generation scripts can be found in the folder *Termparser*[2]. All these Java classes are used in Section 4.3.

4.3 Validation of Terms

Different validation tests are written in Java to ensure

1. the syntactical correctness of the term,
2. the correctness of the AVL property,
3. the correctness of the order and
4. the correctness of the execution of operations.

The first point is pretty much done by the parser itself, as it throws an Exception if a token cannot be parsed.

A routine that does an inorder visit of the nodes was enough to determine if every value is greater or equal than the previous one.

Using a similar approach, the imbalances of all nodes are calculated independently and compared with the current ones in the term.

To verify that a set of operations is done correctly, the values of the old term, new term and each operation with its value are given to a program that, based on the operations and values, applies arithmetic operations on these sets. The result will show two sets of values. One set describes the values that were modified by mistake and should have not been affected by the bulk of operations and the other set shows the values of the operations that could not be performed.

These tests also verify the modeling consistency shown in Figure 2.1.

All these validation tools can be found in the folder *TermAnalyzer*[3].

4.4 Drawing Trees

To get a graphical representation of the search term, a dot [2] file is generated. This file can then be viewed in a program such as GraphViz [7]. The trees in this thesis were not drawn by GraphViz. The program named *main.jar*, which generates the graph file, can be found in the folder *TermAnalyzer*[3].

[2]`https://git.uibk.ac.at/csat1763/SearchTerms/tree/master/Tools/TermParser`
[3]`https://git.uibk.ac.at/csat1763/SearchTerms/tree/master/Tools/TermAnalyzer`

4.5 Termination

Automatic termination tools are very convenient because proofs can get quite large. The tools TTT2 [9] and *CiME* [6] were used locally on a command line. The third one being *AProVE* [1] was used in the provided web-interface.

The setup and results can be found in the folder *Termination*[4].

4.6 Complexity

Tools that automatically show the runtime-complexity of a TRS also exist. One example is the Tyrolean Complexity Tool TcT [12]. For the operations *search* and *insert* of the parallel TRS, the tool yielded a lower-bound of $\mathcal{O}(1)$ and an upper-bound of $\mathcal{O}(n)$. For *delete*, we got a lower-bound of $\mathcal{O}(1)$ and an upper-bound of $\mathcal{O}(n^2)$. Unfortunately, we had to trim the TRS down to the basic operations to get results because otherwise we would have reached a timeout.

The files used for the testing and the description for the test procedure can be found in the folder *TcT-Files*[5].

[4]`https://git.uibk.ac.at/csat1763/SearchTerms/tree/master/Termination`
[5]`https://git.uibk.ac.at/csat1763/SearchTerms/tree/master/TcT-Files`

5 Running the TRS

As our final desire, we want to compare the parallel implementation with the sequential implementation and also describe the testing process that was deployed.

The TRS will be executed on a machine with the following properties:

- Operating System: CentOS Linux 7
- Hardware Architecture: x86_64
- CPU: Intel Xeon E312xx

and with two TRS engines, one being Maude 2.7.1.

5.1 Testing

Since the sequential implementation locks the whole term, it is safe to assume that any operation will be done successfully.

However, it is essential to run some tests on the parallel TRS, to see if any corner cases can occur and if the bulk operations is executed correctly.

Given a base term that represents an AVL tree with 100 existing nodes, we randomly generate numbers in the range of 0 to 100, each number being either inserted or deleted. Since one of the validation tools from Section 4.3 also checks the correct execution of the input, the set of random numbers will have no duplicates.

The general idea is to perform the bulk of operations on the constant term and give the old term, new term, and the operations to the validation routine to obtain results. This will be executed in a loop and if a validation test yields a negative result, the whole test will be interrupted with a proper termination message.

Finally, the correct deployment of bulk operations is left to the user. Meaning if a user decides to perform an insert and a delete on the same numeric value, then this will be executed. The input operations are always executed with disregard to their semantics.

5.1.1 Result

Using Maude as an engine, around 1 000 tests have been conducted and every single test was successful. The script that was used can be found in the main folder *Tests*[1] and the results can be found in the sub-folder named *oldResults*.

[1] `https://git.uibk.ac.at/csat1763/SearchTerms/tree/master/Tests`

5.2 Runtime Measurements

The practical results, when running the implemented systems, will be shown and compared with each other. For the runs, the initial search term is $tree(n(0, leaf, 0, leaf)$ and the operation is the insertion of 0.

The results (*measurement.dat*) and scripts (*run.sh*, *5xRun.sh*) can each be found in the folder *Measurements*[2]. The folder *MaudeMeasurement* is for the results in Figure 5.1 and the folder *CLToolMeassurement* is for the results in Figure 5.2. We took the average time of five runs for each number of inserts.

Maude

When Maude executes rewrite steps, it tries to mimic parallel execution by interleaving the execution between different parts of the term. Only one rewrite step at the time is being performed.

Number of Inserts	Sequential Time in ms	Parallel Time in ms
10	97	90
20	93	92
50	96	95
100	94	171
200	109	728
500	444	8,571
1,000	2,103	61,623

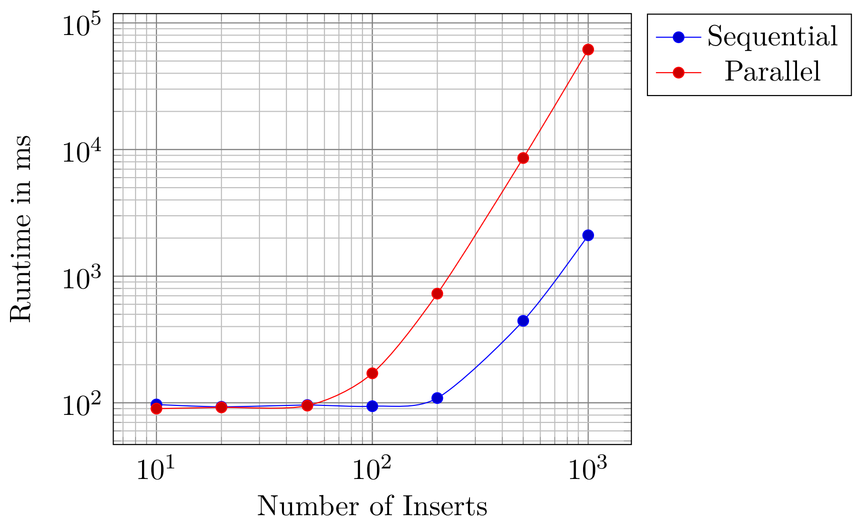

Figure 5.1: Parallel and Sequential TRS - Runtimes

[2]`https://git.uibk.ac.at/csat1763/SearchTerms/tree/master/Measurements`

According to the results, the parallel implementation performs worse compared to the sequential one. This is due to the fact that Maude itself does neither utilize a maximal strategy nor a parallel execution. This can be enabled in Maude, but we chose not to pursue it since it would have taken a substantial amount of time.

This stresses the need for a managing process described in Section 3.12 or an engine that can at least execute rewrite steps simultaneously.

CLTool

This engine was kindly provided by Fabian Mitterwallner who implemented it as a bachelor project [11]. It is an educational tool for term rewriting rather than performance testing. We were able to easily use the maximal strategy when running the TRS.

Number of Inserts	Sequential Time in ms	Parallel Time in ms
10	206	39
20	192	76
50	1,247	330
100	5,583	1,267
200	25,010	4,970
500	$1.8 \cdot 10^5$	31,757
1,000	$8.08 \cdot 10^5$	$1.26 \cdot 10^5$

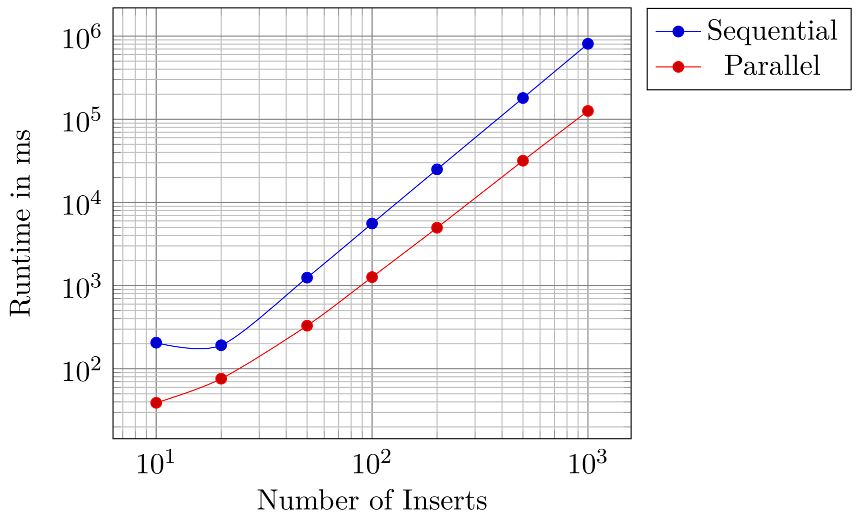

Figure 5.2: Parallel and Sequential TRS - Runtimes

Even though, with regards to the runtime, this engine performs worse than Maude, we are able to see that the parallel TRS performs better with just the maximal strategy.

6 Conclusion

The goal was to implement search trees and operations on them as term rewrite systems. To be precise, we implemented a sequential and a parallel TRS for AVL trees. For both, termination was proven by termination tools.

For the parallel TRS, introducing atomicity was a key part in enabling the simultaneous execution of bulk operations.

Performance could potentially be increased by running the TRS in an engine that can handle parallel execution steps.

Various tools that were useful during the development process have also been implemented, mainly scripts that helped to easily perform update-operations on search terms. Also, a test program verified the correctness of operations and the term. Lastly, the generated dot files enable the inspection of the graphical tree.

Future work could probably be the improvement of the parallel TRS in order to maximize performance. It would also be necessary to implement a program that manages the execution and parallelization of operations on search terms, as described in Section 3.12. A web-interface, that offers to perform update-operations and always displays the current search term with the equivalent graphical tree, would be convenient.

Related work is conducted by Prof. Tobias Nipkow. In his paper [15], he addresses the implementation of functional search trees, including AVL trees. Mainly, proofs for functional correctness are shown by referenced work [14] and the analysis of the complexities is done in [13]. In our work, formal correctness is neglected. Our main focus was to implement an AVL tree, which allowed the parallel execution of update-operations. For this implementation, we tried to express the runtime-complexity.

So in conclusion, term rewrite systems are indeed well suited for the implementation of search trees, though, when it comes to numerical values, a significant disadvantage is noticeable. The trade-off is a relation between the number of rewrite rules and the number of rewrite steps.

Either we introduce numeric values as constants, leading to an infinite number of rewrite rules, or we implement the TRS with a finite set of rules, leading to worse performance.

Bibliography

[1] *AProVE - Automated Program Verification Environment*. Research Group Computer Science 2 - RWTH Aachen University. http://aprove.informatik.rwth-aachen.de/.

[2] The dot language. http://www.graphviz.org/doc/info/lang.html.

[3] Termination problem data base format. http://tfmserver.dsic.upv.es:8080/TPDB.html. Accessed: 2019-05-01.

[4] A. Aspir. Repository for Search Terms. University of Innsbruck. https://git.uibk.ac.at/csat1763/SearchTerms.

[5] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. *All About Maude - a High-performance Logical Framework: How to Specify, Program and Verify Systems in Rewriting Logic*. Springer-Verlag, Berlin, Heidelberg, 2007.

[6] E. Contejean, C. Marché, B. Monate, and X. Urbain. *CiME*. Laboratoire de Recherche en Informatique - University of Paris-Sud, STIC - Centre national de la recherche scientifique. http://cime.lri.fr/.

[7] J. Ellson, E. Gansner, L. Koutsofios, S. North, G. Woodhull, S. Description, and L. Technologies. *Graphviz — open source graph drawing tools*. Springer-Verlag, 2001.

[8] M. T. Goodrich and R. Tamassia. *Data Structures and Algorithms in Java*. Wiley Publishing, 4th edition, 2005.

[9] M. Korp, C. Sternagel, H. Zankl, and A. Middeldorp. *Tyrolean Termination Tool 2*. Computational Logic Group - University of Innsbruck. http://cl-informatik.uibk.ac.at/software/ttt2/.

[10] A. Middeldorp. *Term Rewriting*. University of Innsbruck. LVA 703125+703126 (SS 2018).

[11] F. Mitterwallner. Automating Rewrite Strategies. University of Innsbruck. http://cl-informatik.uibk.ac.at/teaching/smb/theses/FM2.pdf.

[12] G. Moser, M. Avanzini, and M. Schaper. *Tyrolean Complexity Tool*. Computation with Bounded Resources - University of Innsbruck. http://cl-informatik.uibk.ac.at/software/tct/.

[13] T. Nipkow. Amortized complexity verified. In C. Urban and X. Zhang, editors, *Interactive Theorem Proving (ITP 2015)*, volume 9236 of *LNCS*, pages 310–324, 2015.

[14] T. Nipkow. Automatic functional correctness proofs for functional search trees. In J. Blanchette and S. Merz, editors, *Interactive Theorem Proving (ITP 2016)*, volume 9807 of *LNCS*, pages 307–322, 2016.

[15] T. Nipkow. Verified Analysis of Functional Data Structures. In D. Kesner and B. Pientka, editors, *1st International Conference on Formal Structures for Computation and Deduction (FSCD 2016)*, volume 52 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 4:1–4:2, Dagstuhl, Germany, 2016. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[16] P. C. Ölveczky and J. Meseguer. *The Real-Time Maude Tool.* `http://heim.ifi.uio.no/~peterol/RealTimeMaude/`.

[17] T. Parr. *The Definitive ANTLR 4 Reference.* Pragmatic Bookshelf, Raleigh, NC, 2 edition, 2013. `https://www.antlr.org/`.

[18] Terese. *Term Rewriting Systems*, volume 55 of *Cambridge Tracts in Theoretical Computer Science.* Cambridge University Press, 2003.

A Calculations

A.1 Imbalances of Case 2

Imbalances:

$$D(x_c) = H(x_b) - H(x_3) = +2 \tag{A.1}$$
$$D(x_b) = H(x_a) - H(x_2) = +1 \tag{A.2}$$
$$D(x_a) = H(x_0) - H(x_1) = d_3 \tag{A.3}$$

Heights:

$$(A.2) \Rightarrow H(x_a) = H(x_2) + 1 \tag{A.4}$$
$$(A.2) \wedge (A.4) \Rightarrow H(x_b) = H(x_a) + 1 = H(x_2) + 2 \tag{A.5}$$
$$(A.1) \Rightarrow H(x_b) = H(x_3) + 2 \tag{A.6}$$
$$(A.5) \wedge (A.6) \Rightarrow H(x_2) + 2 = H(x_3) + 2 \Rightarrow H(x_2) = H(x_3) \tag{A.7}$$

New Heights:

$$(A.7) \Rightarrow H(x_c) = H(x_2) + 1 \tag{A.8}$$

New Imbalances:

$$(A.7) \Rightarrow D(x_c) = H(x_2) - H(x_3) = 0 \tag{A.9}$$
$$\begin{aligned}(A.4) \wedge (A.8) \Rightarrow D(x_b) = H(x_a) - H(x_c) =\\ = (H(x_2) + 1) - (H(x_2) + 1) = 0\end{aligned} \tag{A.10}$$

A.2 Imbalances of Case 3

Imbalances:

$$D(x_a) = H(x_0) - H(x_c) = -2 \tag{A.11}$$
$$D(x_c) = H(x_b) - H(x_3) = +1 \tag{A.12}$$

A.2.1 First Value

$$D(x_b) = H(x_1) - H(x_2) = 0 \tag{A.13}$$

Heights:

$$(A.13) \Rightarrow H(x_1) = H(x_2) \tag{A.14}$$

$$(A.12) \Rightarrow H(x_b) = H(x_3) + 1 \tag{A.15}$$

$$(A.13) \wedge (A.14) \Rightarrow H(x_b) = H(x_2) + 1 = H(x_1) + 1 \tag{A.16}$$

$$(A.11) \Rightarrow H(x_c) = H(x_0) + 2 \tag{A.17}$$

$$(A.12) \wedge (A.16) \Rightarrow H(x_c) = H(x_b) + 1 = H(x_1) + 2 \tag{A.18}$$

$$(A.17) \wedge (A.18) \Rightarrow H(x_0) + 2 = H(x_1) + 2 \Rightarrow H(x_0) = H(x_1) \tag{A.19}$$

$$(A.15) \wedge (A.16) \Rightarrow H(x_3) + 1 = H(x_2) + 1 \Rightarrow H(x_3) = H(x_2) \tag{A.20}$$

New Heights:

$$(A.19) \Rightarrow H(x_a) = H(x_1) + 1 \tag{A.21}$$

$$(A.20) \Rightarrow H(x_c) = H(x_2) + 1 \tag{A.22}$$

New Imbalances:

$$(A.19) \Rightarrow D(x_a) = H(x_0) - H(x_1) = 0 \tag{A.23}$$

$$(A.20) \Rightarrow D(x_c) = H(x_2) - H(x_3) = 0 \tag{A.24}$$

$$\begin{aligned}(A.14) \wedge (A.21) \wedge (A.22) \Rightarrow D(x_b) = H(x_a) - H(x_c) = \\ = (H(x_1) + 1) - (H(x_2) + 1) = (H(x_1) + 1) - (H(x_1) + 1) = 0\end{aligned} \tag{A.25}$$

A.2.2 Second Value

$$D(x_b) = H(x_1) - H(x_2) = -1 \tag{A.26}$$

Heights:

$$(A.26) \Rightarrow H(x_1) + 1 = H(x_2) \tag{A.27}$$

$$(A.26) \wedge (A.27) \Rightarrow H(x_b) = H(x_2) + 1 = H(x_1) + 2 \tag{A.28}$$

$$(A.12) \wedge (A.28) \Rightarrow H(x_c) = H(x_b) + 1 = H(x_1) + 3 \tag{A.29}$$

$$(A.17) \wedge (A.29) \Rightarrow H(x_0) + 2 = H(x_1) + 3 \Rightarrow H(x_0) = H(x_1) + 1 \tag{A.30}$$

$$(A.15) \wedge (A.28) \Rightarrow H(x_3) + 1 = H(x_2) + 1 \Rightarrow H(x_3) = H(x_2) \tag{A.31}$$

New Heights:

$$(A.30) \Rightarrow H(x_a) = H(x_1) + 2 \tag{A.32}$$

$$(A.31) \Rightarrow H(x_c) = H(x_2) + 1 \tag{A.33}$$

New Imbalances:

$$(A.30) \Rightarrow D(x_a) = H(x_0) - H(x_1) = H(x_1) + 1 - H(x_1) = +1 \tag{A.34}$$

$$(A.31) \Rightarrow D(x_c) = H(x_2) - H(x_3) = 0 \tag{A.35}$$

$$\begin{aligned}(A.27) \wedge (A.32) \wedge (A.33) \Rightarrow D(x_b) = H(x_a) - H(x_c) = \\ = (H(x_1) + 2) - (H(x_2) + 1) = (H(x_1) + 2) - ((H(x_1) + 1) + 1) = 0\end{aligned} \tag{A.36}$$

A.2.3 Third Value

$$D(x_b) = H(x_1) - H(x_2) = +1 \tag{A.37}$$

Heights:

$$(A.37) \Rightarrow H(x_2) + 1 = H(x_1) \tag{A.38}$$

$$(A.37) \wedge (A.38) \Rightarrow H(x_b) = H(x_1) + 1 = H(x_2) + 2 \tag{A.39}$$

$$(A.12) \wedge (A.39) \Rightarrow H(x_c) = H(x_b) + 1 = H(x_1) + 2 \tag{A.40}$$

$$(A.17) \wedge (A.40) \Rightarrow H(x_0) + 2 = H(x_1) + 2 \Rightarrow H(x_0) = H(x_1) \tag{A.41}$$

$$(A.15) \wedge (A.39) \Rightarrow H(x_2) + 2 = H(x_3) + 1 \Rightarrow H(x_2) + 1 = H(x_3) \tag{A.42}$$

New Heights:

$$(A.41) \Rightarrow H(x_a) = H(x_1) + 1 \tag{A.43}$$

$$(A.42) \Rightarrow H(x_c) = H(x_2) + 2 \tag{A.44}$$

New Imbalances:

$$(A.41) \Rightarrow D(x_a) = H(x_0) - H(x_1) = H(x_0) - H(x_0) = 0 \tag{A.45}$$

$$(A.42) \Rightarrow D(x_c) = H(x_2) - H(x_3) = H(x_2) - (H(x_2) + 1) = -1 \tag{A.46}$$

$$\begin{aligned}(A.38) \wedge (A.43) \wedge (A.44) \Rightarrow D(x_b) = H(x_a) - H(x_c) = \\ = (H(x_1) + 1) - (H(x_2) + 2) = (H(x_1) + 1) - (H(x_1) + 1) = 0\end{aligned} \tag{A.47}$$

A.3 Imbalances of Case 4

Imbalances:

$$D(x_c) = H(x_a) - H(x_3) = +2 \tag{A.48}$$

$$D(x_a) = H(x_0) - H(x_b) = -1 \tag{A.49}$$

A.3.1 First Value

$$D(x_b) = H(x_1) - H(x_2) = 0 \tag{A.50}$$

Heights:

$$(A.50) \Rightarrow H(x_1) = H(x_2) \tag{A.51}$$

$$(A.50) \Rightarrow H(x_b) = H(x_0) + 1 \tag{A.52}$$

$$(A.50) \wedge (A.51) \Rightarrow H(x_b) = H(x_2) + 1 = H(x_1) + 1 \tag{A.53}$$

$$(A.48) \Rightarrow H(x_a) = H(x_3) + 2 \tag{A.54}$$

$$(A.49) \wedge (A.53) \Rightarrow H(x_a) = H(x_b) + 1 = H(x_2) + 2 \tag{A.55}$$

$$(A.52) \wedge (A.53) \Rightarrow H(x_0) + 1 = H(x_1) + 1 \Rightarrow H(x_0) = H(x_1) \tag{A.56}$$

$$(A.54) \wedge (A.55) \Rightarrow H(x_3) + 2 = H(x_2) + 2 \Rightarrow H(x_2) = H(x_3) \tag{A.57}$$

New Heights:

$$(A.56) \Rightarrow H(x_a) = H(x_1) + 1 \tag{A.58}$$

$$(A.57) \Rightarrow H(x_c) = H(x_2) + 1 \tag{A.59}$$

New Imbalances:

$$(A.56) \Rightarrow D(x_a) = H(x_0) - H(x_1) = 0 \tag{A.60}$$

$$(A.57) \Rightarrow D(x_c) = H(x_2) - H(x_3) = 0 \tag{A.61}$$

$$\begin{aligned}(A.51) \wedge (A.58) \wedge (A.59) \Rightarrow D(x_b) = H(x_a) - H(x_c) = \\ = (H(x_1) + 1) - (H(x_2) + 1) = (H(x_1) + 1) - (H(x_1) + 1) = 0\end{aligned} \tag{A.62}$$

A.3.2 Second Value

$$D(x_b) = H(x_1) - H(x_2) = -1 \tag{A.63}$$

Heights:

$$(A.63) \Rightarrow H(x_1) + 1 = H(x_2) \tag{A.64}$$

$$(A.63) \wedge (A.64) \Rightarrow H(x_b) = H(x_2) + 1 = H(x_1) + 2 \tag{A.65}$$

$$(A.49) \wedge (A.65) \Rightarrow H(x_a) = H(x_b) + 1 = H(x_2) + 2 \tag{A.66}$$

$$(A.52) \wedge (A.65) \Rightarrow H(x_0) + 1 = H(x_1) + 2 \Rightarrow H(x_0) = H(x_1) + 1 \tag{A.67}$$

$$(A.54) \wedge (A.66) \Rightarrow H(x_3) + 2 = H(x_2) + 2 \Rightarrow H(x_3) = H(x_2) \tag{A.68}$$

New Heights:

$$(A.67) \Rightarrow H(x_a) = H(x_1) + 2 \tag{A.69}$$

$$(A.68) \Rightarrow H(x_c) = H(x_2) + 1 \tag{A.70}$$

New Imbalances:

$$(A.67) \Rightarrow D(x_a) = H(x_0) - H(x_1) = (H(x_1) + 1) - H(x_1) = +1 \tag{A.71}$$

$$(A.68) \Rightarrow D(x_c) = H(x_2) - H(x_3) = 0 \tag{A.72}$$

$$\begin{aligned}(A.64) \wedge (A.69) \wedge (A.70) \Rightarrow D(x_b) = H(x_a) - H(x_c) = \\ = (H(x_1) + 2) - (H(x_2) + 1) = (H(x_1) + 2) - ((H(x_1) + 1) + 1) = 0\end{aligned} \tag{A.73}$$

A.3.3 Third Value

$$D(x_b) = H(x_1) - H(x_2) = +1 \tag{A.74}$$

Heights:

$$(A.74) \Rightarrow H(x_1) = H(x_2) + 1 \tag{A.75}$$

$$(A.74) \wedge (A.75) \Rightarrow H(x_b) = H(x_2) + 2 = H(x_1) + 1 \tag{A.76}$$

$$(A.49) \wedge (A.76) \Rightarrow H(x_a) = H(x_b) + 1 = H(x_2) + 3 \tag{A.77}$$

$$(A.52) \wedge (A.76) \Rightarrow H(x_0) + 1 = H(x_1) + 1 \Rightarrow H(x_0) = H(x_1) \tag{A.78}$$

$$(A.54) \wedge (A.77) \Rightarrow H(x_3) + 2 = H(x_2) + 3 \Rightarrow H(x_3) = H(x_2) + 1 \tag{A.79}$$

New Heights:

$$(A.78) \Rightarrow H(x_a) = H(x_1) + 1 \tag{A.80}$$

$$(A.79) \Rightarrow H(x_c) = H(x_2) + 2 \tag{A.81}$$

New Imbalances:

$$(A.78) \Rightarrow D(x_a) = H(x_0) - H(x_1) = 0 \tag{A.82}$$

$$(A.79) \Rightarrow D(x_c) = H(x_2) - H(x_3) = H(x_2) - (H(x_2) + 1) = -1 \tag{A.83}$$

$$\begin{split}(A.75) \wedge (A.80) \wedge (A.81) \Rightarrow D(x_b) = H(x_a) - H(x_c) = \\ = (H(x_1) + 1) - (H(x_2) + 2) = ((H(x_2) + 1) + 1) - (H(x_2) + 2) = 0\end{split} \tag{A.84}$$

B TRS Codes

B.1 Parallel AVL

```
( VAR x y x_a x_b x_c d_0 d_1 d_2 d_3 x_0 x_1 x_2 x_3 p_0 p_1 p_2 p_3 b
   b_0 )
( RULES
```

B.1.1 Search

```
search(x,tree(x_0)) -> search(x,x_0)
search(x,n(d_0,x_1,x_a,x_2)) -> search1(x,n(d_0,x_1,x_a,x_2),geq(x,x_a))
search1(x,n(d_0,x_1,x_a,x_2),true) -> search2(x,n(d_0,x_1,x_a,x_2),eq(x,
   x_a))
search2(x,n(d_0,x_1,x_a,x_2),true) -> entry(true,getValueFor(x))
search2(x,n(d_0,x_1,x_a,x_2),false) -> search(x,x_2)
search1(x,n(d_0,x_1,x_a,x_2),false) -> search(x,x_1)
search(x,leaf) -> false
```

B.1.2 Insert

```
insert(x,tree(x_0)) -> tree(insx(x,x_0))
insx(x,n(d_1,x_1,s(x_a),x_2)) -> insert1(x,n7(d_1,x_1,s(x_a),x_2),geq(x,
   s(x_a)))
insx(x,n(d_1,x_1,0,x_2)) -> insert1(x,n7(d_1,x_1,0,x_2),geq(x,0))
insert1(x,n7(d_1,x_1,x_a,x_2),true) -> n(d_1,x_1,x_a,insx(x,x_2))
insert1(x,n7(d_1,x_1,x_a,x_2),false) -> n(d_1,insx(x,x_1),x_a,x_2)
insx(x,leaf) -> up(up+,n(0,leaf,x,leaf))
root(x) -> tree(n(0,leaf,x,leaf))
n(0,x_0,x_a,up(up+,x_1)) -> up(up+,n(-(s(0)),x_0,x_a,x_1))
n(0,up(up+,x_0),x_a,x_1) -> up(up+,n(+(s(0)),x_0,x_a,x_1))
n(+(s(0)),x_0,x_a,up(up+,x_1)) -> n(0,x_0,x_a,x_1)
n(-(s(0)),up(up+,x_0),x_a,x_1) -> n(0,x_0,x_a,x_1)
n(+(s(0)),up(up+,n(+(s(0)),n(d_3,x_0,x_a,x_1),x_b,x_2)),x_c,x_3) ->
   continuePropagation(trinode(n1(+(s(s(0))),n1(+(s(0)),n1(d_3,x_0,x_a,
   x_1),x_b,x_2),x_c,x_3)),up+)
```

```
n(+(s(0)),up(up+,n(0,n(d_3,x_0,x_a,x_1),x_b,x_2)),x_c,x_3) ->
    continuePropagation(trinode(n1(+(s(s(0))),n1(0,n1(d_3,x_0,x_a,x_1),
    x_b,x_2),x_c,x_3)),up+)
n(+(s(0)),up(up+,n(-(s(0)),x_0,x_a,n(+(s(0)),x_1,x_b,x_2))),x_c,x_3) ->
    continuePropagation(trinode(n1(+(s(s(0))),n1(-(s(0)),x_0,x_a,n1(+(s
    (0)),x_1,x_b,x_2)),x_c,x_3)),up+)
n(+(s(0)),up(up+,n(-(s(0)),x_0,x_a,n(-(s(0)),x_1,x_b,x_2))),x_c,x_3) ->
    continuePropagation(trinode(n1(+(s(s(0))),n1(-(s(0)),x_0,x_a,n1(-(s
    (0)),x_1,x_b,x_2)),x_c,x_3)),up+)
n(+(s(0)),up(up+,n(-(s(0)),x_0,x_a,n(0,x_1,x_b,x_2))),x_c,x_3) ->
    continuePropagation(trinode(n1(+(s(s(0))),n1(-(s(0)),x_0,x_a,n1(0,x_1
    ,x_b,x_2)),x_c,x_3)),up+)
n(-(s(0)),x_0,x_a,up(up+,n(-(s(0)),x_1,x_b,n(d_3,x_2,x_c,x_3)))) ->
    continuePropagation(trinode(n1(-(s(s(0))),x_0,x_a,n1(-(s(0)),x_1,x_b,
    n1(d_3,x_2,x_c,x_3)))),up+)
n(-(s(0)),x_0,x_a,up(up+,n(0,x_1,x_b,n(d_3,x_2,x_c,x_3)))) ->
    continuePropagation(trinode(n1(-(s(s(0))),x_0,x_a,n1(0,x_1,x_b,n1(d_3
    ,x_2,x_c,x_3)))),up+)
n(-(s(0)),x_0,x_a,up(up+,n(+(s(0)),n(+(s(0)),x_1,x_b,x_2),x_c,x_3))) ->
    continuePropagation(trinode(n1(-(s(s(0))),x_0,x_a,n1(+(s(0)),n1(+(s
    (0)),x_1,x_b,x_2),x_c,x_3))),up+)
n(-(s(0)),x_0,x_a,up(up+,n(+(s(0)),n(-(s(0)),x_1,x_b,x_2),x_c,x_3))) ->
    continuePropagation(trinode(n1(-(s(s(0))),x_0,x_a,n1(+(s(0)),n1(-(s
    (0)),x_1,x_b,x_2),x_c,x_3))),up+)
n(-(s(0)),x_0,x_a,up(up+,n(+(s(0)),n(0,x_1,x_b,x_2),x_c,x_3))) ->
    continuePropagation(trinode(n1(-(s(s(0))),x_0,x_a,n1(+(s(0)),n1(0,x_1
    ,x_b,x_2),x_c,x_3))),up+)
continuePropagation(nonZeroBF(x_0),up+) -> up(up+,x_0)
continuePropagation(n(d_0,x_0,x_a,x_1),up+) -> n(d_0,x_0,x_a,x_1)
```

B.1.3 Delete

```
delete(x,tree(x_0)) -> tree(delx(x,x_0))
delx(x,n(d_1,x_1,x_a,x_2)) -> isTargetNode(x,n(d_1,x_1,x_a,x_2))
delx(x,leaf) -> leaf

isTargetNode(x,n(d_1,n(d_2,x_0,x_b,x_1),s(x_a),n(d_3,x_2,x_c,x_3))) ->
    isTargetNode1(x,n8(d_1,n8(d_2,x_0,x_b,x_1),s(x_a),n8(d_3,x_2,x_c,x_3)
    ),eq(x,s(x_a)))
isTargetNode(x,n(d_1,n(d_2,x_0,x_b,x_1),s(x_a),leaf)) -> isTargetNode1(x,
    n8(d_1,n8(d_2,x_0,x_b,x_1),s(x_a),leaf),eq(x,s(x_a)))
isTargetNode(x,n(d_1,leaf,s(x_a),n(d_3,x_2,x_c,x_3))) -> isTargetNode1(x,
    n8(d_1,leaf,s(x_a),n8(d_3,x_2,x_c,x_3)),eq(x,s(x_a)))
```

```
isTargetNode(x,n(d_1,leaf,s(x_a),leaf)) -> isTargetNode1(x,n8(d_1,leaf,s
    (x_a),leaf),eq(x,s(x_a)))
isTargetNode(x,n(d_1,n(d_2,x_0,x_b,x_1),0,n(d_3,x_2,x_c,x_3))) ->
    isTargetNode1(x,n8(d_1,n8(d_2,x_0,x_b,x_1),0,n8(d_3,x_2,x_c,x_3)),eq(
    x,0))
isTargetNode(x,n(d_1,n(d_2,x_0,x_b,x_1),0,leaf)) -> isTargetNode1(x,n8(
    d_1,n8(d_2,x_0,x_b,x_1),0,leaf),eq(x,0))
isTargetNode(x,n(d_1,leaf,0,n(d_3,x_2,x_c,x_3))) -> isTargetNode1(x,n8(
    d_1,leaf,0,n8(d_3,x_2,x_c,x_3)),eq(x,0))
isTargetNode(x,n(d_1,leaf,0,leaf)) -> isTargetNode1(x,n8(d_1,leaf,0,leaf
    ),eq(x,0))
isTargetNode1(x,n8(d_1,n8(d_2,x_0,x_b,x_1),x_a,n8(d_3,x_2,x_c,x_3)),
    false) -> goRight(x,n5(d_1,n(d_2,x_0,x_b,x_1),x_a,n(d_3,x_2,x_c,x_3))
    ,geq(x,x_a))

isTargetNode1(x,n8(d_1,n8(d_2,x_0,x_b,x_1),x_a,leaf),false) -> goRight(x,
    n5(d_1,n(d_2,x_0,x_b,x_1),x_a,leaf),geq(x,x_a))
isTargetNode1(x,n8(d_1,leaf,x_a,n8(d_3,x_2,x_c,x_3)),false) -> goRight(x,
    n5(d_1,leaf,x_a,n(d_3,x_2,x_c,x_3)),geq(x,x_a))
isTargetNode1(x,n8(d_1,leaf,x_a,leaf),false) -> n(d_1,leaf,x_a,leaf)
isTargetNode1(x,n8(d_1,leaf,x_a,leaf),true) -> up(up-,leaf)
isTargetNode1(x,n8(d_1,n8(d_2,x_0,x_b,x_1),x_a,leaf),true) -> up(up-,n(
    d_2,x_0,x_b,x_1))
isTargetNode1(x,n8(d_1,leaf,x_a,n8(d_2,x_0,x_b,x_1)),true) -> up(up-,n(
    d_2,x_0,x_b,x_1))
isTargetNode1(x,n8(d_1,n8(d_2,x_0,x_b,x_1),x_a,n8(d_3,x_2,x_c,x_3)),true
    ) -> n(d_1,n(d_2,x_0,x_b,x_1),placeholder,consume(newSubTree(n(d_3,
    x_2,x_c,x_3))))

n(d_0,replacement(x_c,x_0),x_a,x_1) -> replacement(x_c,n(d_0,x_0,x_a,x_1
    ))
n(d_0,x_0,x_a,replacement(x_c,x_1)) -> replacement(x_c,n(d_0,x_0,x_a,x_1
    ))
n(d_0,x_0,placeholder,consume(replacement(x_c,x_1))) -> n(d_0,x_0,x_c,
    x_1)
goRight(x,n5(d_1,x_1,x_a,x_2),true) -> n(d_1,x_1,x_a,delx(x,x_2))
goRight(x,n5(d_1,x_1,x_a,x_2),false) -> n(d_1,delx(x,x_1),x_a,x_2)
newSubTree(n(d_1,leaf,x_a,n(d_2,x_0,x_b,x_1))) -> replacement(x_a,up(up-,
    n(d_2,x_0,x_b,x_1)))
newSubTree(n(d_1,n(d_2,x_0,x_b,x_1),x_a,leaf)) -> n(d_1,newSubTree(n(d_2,
    x_0,x_b,x_1)),x_a,leaf)
newSubTree(n(d_1,n(d_2,x_0,x_b,x_1),x_a,n(d_3,x_2,x_c,x_3))) -> n(d_1,
    newSubTree(n(d_2,x_0,x_b,x_1)),x_a,n(d_3,x_2,x_c,x_3))
newSubTree(n(d_1,leaf,x_a,leaf)) -> replacement(x_a,up(up-,leaf))

n(0,x_1,x_a,up(up-,x_0)) -> n(+(s(0)),x_1,x_a,x_0)
n(0,up(up-,x_0),x_a,x_1) -> n(-(s(0)),x_0,x_a,x_1)
n(+(s(0)),up(up-,x_1),x_a,x_0) -> up(up-,n(0,x_1,x_a,x_0))
n(-(s(0)),x_1,x_a,up(up-,x_0)) -> up(up-,n(0,x_1,x_a,x_0))
n(+(s(0)),n(+(s(0)),n(d_3,x_0,x_a,x_1),x_b,x_2),x_c,up(up-,x_3)) ->
   continuePropagation(trinode(n1(+(s(s(0))),n1(+(s(0)),n1(d_3,x_0,x_a,
   x_1),x_b,x_2),x_c,x_3)),up-)
n(+(s(0)),n(0,n(d_3,x_0,x_a,x_1),x_b,x_2),x_c,up(up-,x_3)) ->
   continuePropagation(trinode(n1(+(s(s(0))),n1(0,n1(d_3,x_0,x_a,x_1),
   x_b,x_2),x_c,x_3)),up-)
n(+(s(0)),n(-(s(0)),x_0,x_a,n(+(s(0)),x_1,x_b,x_2)),x_c,up(up-,x_3)) ->
   continuePropagation(trinode(n1(+(s(s(0))),n1(-(s(0)),x_0,x_a,n1(+(s
   (0)),x_1,x_b,x_2)),x_c,x_3)),up-)
n(+(s(0)),n(-(s(0)),x_0,x_a,n(-(s(0)),x_1,x_b,x_2)),x_c,up(up-,x_3)) ->
   continuePropagation(trinode(n1(+(s(s(0))),n1(-(s(0)),x_0,x_a,n1(-(s
   (0)),x_1,x_b,x_2)),x_c,x_3)),up-)
n(+(s(0)),n(-(s(0)),x_0,x_a,n(0,x_1,x_b,x_2)),x_c,up(up-,x_3)) ->
   continuePropagation(trinode(n1(+(s(s(0))),n1(-(s(0)),x_0,x_a,n1(0,x_1
   ,x_b,x_2)),x_c,x_3)),up-)
n(-(s(0)),up(up-,x_0),x_a,n(-(s(0)),x_1,x_b,n(d_3,x_2,x_c,x_3))) ->
   continuePropagation(trinode(n1(-(s(s(0))),x_0,x_a,n1(-(s(0)),x_1,x_b,
   n1(d_3,x_2,x_c,x_3)))),up-)
n(-(s(0)),up(up-,x_0),x_a,n(0,x_1,x_b,n(d_3,x_2,x_c,x_3))) ->
   continuePropagation(trinode(n1(-(s(s(0))),x_0,x_a,n1(0,x_1,x_b,n1(d_3
   ,x_2,x_c,x_3)))),up-)
n(-(s(0)),up(up-,x_0),x_a,n(+(s(0)),n(+(s(0)),x_1,x_b,x_2),x_c,x_3)) ->
   continuePropagation(trinode(n1(-(s(s(0))),x_0,x_a,n1(+(s(0)),n1(+(s
   (0)),x_1,x_b,x_2),x_c,x_3))),up-)
n(-(s(0)),up(up-,x_0),x_a,n(+(s(0)),n(-(s(0)),x_1,x_b,x_2),x_c,x_3)) ->
   continuePropagation(trinode(n1(-(s(s(0))),x_0,x_a,n1(+(s(0)),n1(-(s
   (0)),x_1,x_b,x_2),x_c,x_3))),up-)
n(-(s(0)),up(up-,x_0),x_a,n(+(s(0)),n(0,x_1,x_b,x_2),x_c,x_3)) ->
   continuePropagation(trinode(n1(-(s(s(0))),x_0,x_a,n1(+(s(0)),n1(0,x_1
   ,x_b,x_2),x_c,x_3))),up-)
continuePropagation(nonZeroBF(n(d_0,x_0,x_a,x_1)),up-) -> n(d_0,x_0,x_a,
   x_1)
continuePropagation(n(d_0,x_0,x_a,x_1),up-) -> up(up-,n(d_0,x_0,x_a,x_1)
   )
```

B.1.4 Trinode

```
trinode(n1(-(s(s(0))),x_0,x_a,n1(-(s(0)),x_1,x_b,n1(d_3,x_2,x_c,x_3))))
    -> n(0,n(0,x_0,x_a,x_1),x_b,n(d_3,x_2,x_c,x_3))
trinode(n1(-(s(s(0))),x_0,x_a,n1(0,x_1,x_b,n1(d_3,x_2,x_c,x_3)))) ->
    nonZeroBF(n(+(s(0)),n(-(s(0)),x_0,x_a,x_1),x_b,n(d_3,x_2,x_c,x_3)))
trinode(n1(+(s(s(0))),n1(+(s(0)),n1(d_3,x_0,x_a,x_1),x_b,x_2),x_c,x_3))
    -> n(0,n(d_3,x_0,x_a,x_1),x_b,n(0,x_2,x_c,x_3))
trinode(n1(+(s(s(0))),n1(0,n1(d_3,x_0,x_a,x_1),x_b,x_2),x_c,x_3)) ->
    nonZeroBF(n(-(s(0)),n(d_3,x_0,x_a,x_1),x_b,n(+(s(0)),x_2,x_c,x_3)))
trinode(n1(-(s(s(0))),x_0,x_a,n1(+(s(0)),n1(+(s(0)),x_1,x_b,x_2),x_c,x_3
    ))) -> n(0,n(0,x_0,x_a,x_1),x_b,n(-(s(0)),x_2,x_c,x_3))
trinode(n1(-(s(s(0))),x_0,x_a,n1(+(s(0)),n1(-(s(0)),x_1,x_b,x_2),x_c,x_3
    ))) -> n(0,n(+(s(0)),x_0,x_a,x_1),x_b,n(0,x_2,x_c,x_3))
trinode(n1(-(s(s(0))),x_0,x_a,n1(+(s(0)),n1(0,x_1,x_b,x_2),x_c,x_3))) ->
     n(0,n(0,x_0,x_a,x_1),x_b,n(0,x_2,x_c,x_3))
trinode(n1(+(s(s(0))),n1(-(s(0)),x_0,x_a,n1(+(s(0)),x_1,x_b,x_2)),x_c,
    x_3)) -> n(0,n(0,x_0,x_a,x_1),x_b,n(-(s(0)),x_2,x_c,x_3))
trinode(n1(+(s(s(0))),n1(-(s(0)),x_0,x_a,n1(-(s(0)),x_1,x_b,x_2)),x_c,
    x_3)) -> n(0,n(+(s(0)),x_0,x_a,x_1),x_b,n(0,x_2,x_c,x_3))
trinode(n1(+(s(s(0))),n1(-(s(0)),x_0,x_a,n1(0,x_1,x_b,x_2)),x_c,x_3)) ->
     n(0,n(0,x_0,x_a,x_1),x_b,n(0,x_2,x_c,x_3))
```

B.1.5 Collision Handling

```
delx(x,up(p_0,x_1)) -> up(p_0,delx(x,x_1))
isTargetNode1(x,up(p_0,x_0),b) -> up(p_0,isTargetNode1(x,x_0,b))
isTargetNode(x,up(p_0,x_0)) -> up(p_0,isTargetNode(x,x_0))
goRight(x,up(p_0,x_0),b) -> up(p_0,goRight(x,x_0,b))
insx(x,up(p_0,x_0)) -> up(p_0,insx(x,x_0))
tree(up(p_0,x_0)) -> tree(x_0)
up(up-,up(up+,x_0)) -> x_0
up(up+,up(up-,x_0)) -> x_0
newSubTree(up(p_0,x_0)) -> up(p_0,newSubTree(x_0))
up(p_0,replacement(x_a,x_0)) -> replacement(x_a,up(p_0,x_0))
continuePropagation(trinode(replacement(x_a,x_0)),p_0) -> replacement(
    x_a,continuePropagation(trinode(x_0),p_0))
delx(x_b,replacement(x_a,x_0)) -> replacement(x_a,delx(x_b,x_0))
n(d_0,consume(replacement(x_c,x_0)),x_a,x_1) -> consume(replacement(x_c,
    n(d_0,x_0,x_a,x_1)))
```

B.1.6 Arithmetic

```
eq(s(x),s(y)) -> eq(x,y)
eq(0,s(x)) -> false
eq(s(x),0) -> false
eq(0,0) -> true
geq(x,0) -> true
geq(0,s(x)) -> false
geq(s(x),s(y)) -> geq(x,y)

)
```

B.2 Sequential AVL

In order to get the full version of this TRS, existing rules needed to be replaced and missing ones needed to be added from below to the TRS in B.1.

```
insert(x,tree(n(d_1,x_1,x_a,x_2))) -> tree(lock(insx(x,n(d_1,x_1,x_a,x_2
    ))))
n(+(s(0)),x_0,x_a,up(up+,x_1)) -> unlock(n(0,x_0,x_a,x_1))
n(-(s(0)),up(up+,x_0),x_a,x_1) -> unlock(n(0,x_0,x_a,x_1))
continuePropagation(n(d_0,x_0,x_a,x_1),up+) -> unlock(n(d_0,x_0,x_a,x_1)
    )
delete(x,tree(n(d_1,x_1,x_a,x_2))) -> tree(lock(delx(x,n(d_1,x_1,x_a,x_2
    ))))
delx(x,leaf) -> unlock(leaf)
isTargetNode1(x,n8(d_1,leaf,x_a,leaf),false) -> unlock(n(d_1,leaf,x_a,
    leaf))
n(0,x_1,x_a,up(up-,x_0)) -> unlock(n(+(s(0)),x_1,x_a,x_0))
n(0,up(up-,x_0),x_a,x_1) -> unlock(n(-(s(0)),x_0,x_a,x_1))
continuePropagation(nonZeroBF(n(d_0,x_0,x_a,x_1)),up-) -> unlock(n(d_0,
    x_0,x_a,x_1))
lock(up(p_0,x_0)) -> x_0
lock(unlock(x_0)) -> x_0
n(d_0,x_0,x_a,unlock(x_1)) -> unlock(n(d_0,x_0,x_a,x_1))
n(d_0,unlock(x_0),x_a,x_1) -> unlock(n(d_0,x_0,x_a,x_1))
```