Finite sets in mCRL2

by

C.T.M. Louwers

Supervisors:

Prof. Dr. J.C. van de Pol
Prof. Dr. Ir. J.F. Groote

Eindhoven, October 2007
Preface

As a conclusion to the Computer Science and Engineering (CSE) Master Program at the Eindhoven University of Technology (TU/e), a master project needs to be performed. This master project has been done by one student, is accounted for 40 ects and has been performed at the OAS group of the Eindhoven University of Technology.

This master thesis is the conclusion of my master project which was called “Finite sets in mCRL2”. The project started in February 2007 and lasted until October 2007. It describes the research and development for creating various implementations of finite sets in mCRL2. The targeted audience for this report consists of all people that are involved in the mCRL2 project.

Finally I would like to thank some people that helped me during the course of my master project. First of all my graduation tutor and graduation supervisor Prof. Dr. Jaco van de Pol for examining my work and commenting upon it and for reserving time to discuss my work and progress. I would also like to thank Prof. Dr. Ir. Jan Friso Groote for reading my work and commenting upon it. With their help I was able to improve my work and deliver it in the way as it is now.

I would also like to thank my friend Nick Kuijpers for reading various versions of my master thesis with which he helped to improve this document to what it is now.

Eindhoven, October 2007

C.T.M. Louwers
# Table of contents

**LIST OF FIGURES**

<table>
<thead>
<tr>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
</tr>
</tbody>
</table>

**LIST OF TABLES**

<table>
<thead>
<tr>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
</tr>
</tbody>
</table>

1. **INTRODUCTION**

1.1. PROJECT DESCRIPTION

1.2. mCRL2

1.3. OUTLINE

2. **PROJECT APPROACH**

2.1. PROJECT OVERVIEW

2.2. CHANGES WITH RESPECT TO THE ORIGINAL PROJECT DESCRIPTION

3. **ORDERING**

3.1. ORDERING OF ARBITRARY SortS

3.2. GENERAL ORDERING SCHEME

3.3. EXAMPLES

3.3.1. Example 1: Recursion

3.3.2. Example 2: Mutual recursion

4. **FINITE SET AS ORDERED LIST**

4.1. ORDERED LIST

4.2. mCRL2 SPECIFICATION

4.2.1. Sort definition

4.2.2. General functionality

4.2.3. Finite set specific functionality

5. **FINITE SET AS AVL TREE**

5.1. AVL TREE OVERVIEW

5.2. REBALANCING

5.2.1. Single left rotation

5.2.2. Double left rotation

5.2.3. Single right rotation

5.2.4. Double right rotation

5.3. mCRL2 SPECIFICATION

5.3.1. Sort definition

5.3.2. AVL tree specific functionality

5.3.3. Finite set specific functionality

6. **FINITE SET AS LEFT-BALANCED TREE**

6.1. UNIQUE TREE WITH LEFT-BALANCE
List of figures

Figure 1.1 Toolset overview ........................................................................................................ 7
Figure 5.1 AVL tree before single left rotation........................................................................... 28
Figure 5.2 AVL tree after single left rotation ........................................................................... 29
Figure 5.3 AVL tree before double left rotation ....................................................................... 30
Figure 5.4 AVL tree after double left rotation .......................................................................... 31
Figure 5.5 AVL tree before single right rotation ...................................................................... 31
Figure 5.6 AVL tree after single right rotation ...................................................................... 32
Figure 5.7 AVL tree before double right rotation .................................................................... 33
Figure 5.8 AVL tree after double right rotation ...................................................................... 34
Figure 6.1 Example of a left-balanced tree ............................................................................. 54
Figure 6.2 Insert case 1 ............................................................................................................. 58
Figure 6.3 Insert case 2 ............................................................................................................. 58
Figure 6.4 Insert case 3 ............................................................................................................. 59
Figure 6.5 Insert case 4 ............................................................................................................. 60
Figure 6.6 Delete case 1 ............................................................................................................. 64
Figure 6.7 Delete case 2 ............................................................................................................. 64
Figure 6.8 Delete case 3 ............................................................................................................. 65
Figure 6.9 Delete case 4 ............................................................................................................. 65
Figure 6.10 Delete case 5 ......................................................................................................... 66
Figure 9.1 Time results for AVL tree ....................................................................................... 133
Figure 9.2 Results for log(n)..................................................................................................... 134
Figure 9.3 Results for n * log(n)............................................................................................... 135
Figure 9.4 Results for n^2 ......................................................................................................... 136
Figure 9.5 Results for n^3 ......................................................................................................... 137
Figure 9.6 Time results for JITty c rewriting ........................................................................... 140
Figure 10.1 Time results for SSWP ......................................................................................... 143

List of tables

Table 9.1 Time results for AVL tree ......................................................................................... 132
Table 9.2 Time results JITty c rewriting .................................................................................. 139
Table 10.1 Time results for SSWP ........................................................................................... 143
1. Introduction

The first chapter is an introductory chapter and it starts with a description of the assignment as it was formulated at the start of the Master Project. After that, it continues with a short description of the formal specification language mCRL2 and it ends with an outline for the remainder of this document.

1.1. Project description

My master project “Finite sets in mCRL2” originally consisted of the following five parts:

1. Implement finite sets in mCRL2 such that this implementation is accepted by the mCRL2 parser and type checker;
2. Implement a finite set as a List such that this implementation is accepted by the mCRL2 parser and type checker;
3. Implement an ordered rewriter such that a finite set as a List is rewritten to an ordered List;
4. Test the acquired mCRL2 specifications with the help of existing mCRL2 or µCRL examples of (security) protocols;
5. Write a Master Thesis that describes the development of the finite sets in mCRL2.

1.2. mCRL2

The name of the formal specification language mCRL2 stands for milli Common Representation Language 2. It is the successor of the formal specification language µCRL, which stands for micro Common Representation Language. µCRL is a language which extends a basic process algebra based on the Algebra of Communicating Processes (ACP) with abstract data types.

mCRL2 has been developed at the department of Mathematics and Computer Science of the Eindhoven University of Technology (TU/e), LaQuSo (Laboratory for Quality Software) and the CWI (Center for Mathematics and Computer Science). An overview of mCRL2 and Process Algebra can be found in [3] and a motivation for switching from µCRL to mCRL2 can be found in [5].

As already said, mCRL2 is the successor of µCRL and therefore mCRL2 also makes use of abstract data types. These abstract data types for mCRL2 consist of sorts, operations on that sorts and equations on terms. The equations on terms are constructed from operations and variables and the terms are of the same sort.

mCRL2 consists of a number of predefined data types which are the Booleans, represented by the sort Bool, the numbers consist of the positive numbers, the natural numbers, the integers and the real numbers which are represented by Pos, Nat, Int and
Real respectively. mCRL2 also contains type constructors which are predefined operations on sorts with which sorts can be constructed. These type constructors contain the List, the Set, the Bag, the function types and the structured types. For a complete overview of all predefined types with their predefined operations see [7].

mCRL2 is a formal specification language and an overview of mCRL2 can be found in [4]. mCRL2 can be used for modeling, validating and verifying concurrent systems and protocols. It has a toolset with which system analysis and verification can be done automatically. Figure 1.1 displays a picture of the mCRL2 toolset.

![Figure 1.1 Toolset overview](image)

The tools that will be used most during the course of this project are the Lineariser, which creates a linear process specification from a mCRL2 specification, and the LTS generator which generates a Labeled Transition System (or state space) from a linear process specification. Because it is necessary to keep state spaces as small as possible and prevent a state space explosion, it is desirable that a finite set that is to be developed is unique.
1.3. **Outline**

In the remainder of this document, the process of creating finite sets in mCRL2 will be described in detail. The following topics will be treated:

- Chapter 2 describes how I approached the project; it also describes the changes that were made during the process with respect to the original assignment description.

- Chapter 3 deals with the ordering for arbitrary sorts. Such an ordering is necessary to use such a sort in one of the finite set implementations. Chapter 3 starts with giving requirements for a total ordering. After that, a general ordering scheme is described that can be used to generate equations that specify the ordering for an arbitrary sort. The last section of this chapter treats two examples in which can be seen how the general ordering scheme can be used.

- The fourth chapter describes the implementation of a finite set as an ordered List. In this chapter, the complete mCRL2 specification of the finite set as ordered List is covered and explained in detail. To provide an example, I chose to create a finite set as ordered List for the predefined data type Pos that represents the positive numbers.

- In chapter 5, an overview is given of the implementation of a finite set as an AVL tree. The first two sections of this chapter are about AVL trees in general, it describes what an AVL tree is and how they operate. In the last section of this chapter, the mCRL2 specification of a finite set as an AVL tree will be fully covered.

- The implementation of a finite set as a unique tree that is left-balanced is given in the sixth chapter. This chapter includes a part of the mCRL2 specification of this implementation and for that part, a detailed description is given.

- Chapter 7 describes another implementation of a finite set as a unique tree, but this implementation of a unique tree makes use of a conversion to an ordered List. This chapter also includes parts of the mCRL2 specification for this solution.

- Chapter 8 covers some formal proofs for properties of the finite set as ordered List and for the finite set as AVL tree. It contains a number of lemmas and a number of theorems for which detailed proofs are given.

- The ninth chapter describes performance tests that were done for the four finite set implementations. The first section covers the implementation of the finite set as AVL tree and how this implementation behaves under the different mCRL2 rewriting strategies. The second section describes how the four different implementations of the finite set behave under the mCRL2 JITty c rewriter.
Chapter 10 gives two examples of protocols in which finite sets could be used. The first protocol that is chosen for this purpose is the Sliding Window Protocol as it is available as one of the examples in the mCRL2 toolset. For this purpose this Sliding Window Protocol is adapted to a simplified version. This will be covered in the first section. The second section gives an example of an adhoc network protocol.

The final chapter, chapter 11, gives a project evaluation and does some recommendations on the use of the finite set specifications.
2. Project approach

This chapter describes how I approached the project and which choices were made during the process. It also contains a section that describes the changes that were made to the original project description and why those changes were made. I will first start with an overview of the project as it was stated in the original project description.

2.1. Project overview

As already said in the previous chapter, the project originally consisted of the following five parts:

1. Implement finite sets in mCRL2 such that this implementation is accepted by the mCRL2 parser and type checker;

The first part of the project consisted of developing a mCRL2 specification of a finite set in such a way that this specification could be transformed into a linear process specification by using the mcrl22lps tool (Lineariser). The parser and the type checker mentioned in the first aspect are part of this tool, so when a linear process specification can be made from a mCRL2 specification that represents a finite set, this part of the project was fully covered.

2. Implement a finite set as a List such that this implementation is accepted by the mCRL2 parser and type checker;

The second part of the project consisted of developing a finite set specification in mCRL2 in which a finite set was represented as a List (the List type is already available in mCRL2). This meant that the specification of a finite set had to be built using the existing List structure. Just as the previous part, this part was covered when a correct finite set specification could be transformed into a linear process specification by the mcrl22lps tool.

3. Implement an ordered rewriter such that a finite set as a List is rewritten to an ordered List;

In this phase of the project, the intention was to develop and implement an ordered rewriter with which the finite set as a List from the previous part could be transformed into an ordered List. This point was later omitted and the reasons for that will be described in the next section.

4. Test the acquired mCRL2 specifications with the help of existing mCRL2 or µCRL examples of (security) protocols;
This part of the project was designed to test the obtained mCRL2 specification of a finite set in such a way that the finite set was used in existing examples of (security) protocols.

5. Write a Master Thesis that describes the development of the finite sets in mCRL2.

The last part of my Master Project was to write a Master Thesis describing all the previous parts of the project.

2.2. Changes with respect to the original project description

After completing points 1 and 2 of the previous section, the idea came across to develop another mCRL2 specification of a finite set. The idea was to develop a finite set as a self-balancing binary search tree. To achieve this, I examined and researched a number of different self-balancing binary search tree structures such as the Red-Black tree, see [2] for more information, the AA tree, the AVL tree [2], the splay tree and the scapegoat tree.

The reason for choosing a self-balancing binary search tree instead of an ordinary binary search tree was due to the difference in efficiency. A self-balancing binary search tree can operate much more efficiently than an ordinary binary search tree. After comparing the different self-balancing binary search tree structures, I chose to create a finite set as an AVL tree. So instead of continuing with point 3 of the original project description, I continued with developing a mCRL2 specification in which a finite set was represented as an AVL tree.

At this point, yet another choice was made to develop an ordering structure for arbitrary sorts. This choice was made because an ordering has to be available for each sort that is used in an AVL tree. So we decided to provide a general ordering scheme with which equations can be created that describe the ordering for an arbitrary sort. This topic is fully covered in chapter 3.

The next choice I made, in accordance with my supervisors, was to create another two finite set specifications and instead of implementing one of these specifications as a data type into the mCRL2 toolset, provide a document in which all four different specifications are described in detail.

The reason for not implementing the data type myself but instead provide a theoretical foundation was that it would take a disproportionate amount of time to get acquainted with the source code of the toolset. So this meant that the third point of the project description was omitted and the project description was extended with two more points: create two finite set specifications, one of a finite set as a unique tree with left-balance and one of a finite set as a unique tree that makes use of list conversion, and provide theoretical proofs for some properties of the finite set as ordered List and for the finite set as AVL tree.
3. Ordering

3.1. Ordering of arbitrary sorts

To use finite sets of arbitrary sorts, a sort has to have an ordering. For some sorts, such as Pos and Nat, an ordering is already available in mCRL2. For most other sorts this ordering is not yet available. This means that when someone specifies a finite set consisting of a fixed number of elements of a certain sort, an ordering for that sort has to be generated when it is not available. The reason for such an ordering is that with an ordering, a unique List can be created, i.e. a List that is predefined in mCRL2 is not unique, but an ordered List would be unique. Also for the binary search tree implementations of a finite set, ordering is necessary, because this ordering is used for inserting elements into a binary search tree, deleting elements from a binary search tree and searching for a specified element in a binary search tree.

This means that for each sort, the relational operators < and > have to be implemented. The ordering that is used has to be strictly total, which means that it has the following requirements:

For all \( a, b \) and \( c \) in a set \( S \), a relation < is a strict order if:

- \( a < a \) doesn’t hold for any \( a \in S \) (irreflexivity).
- \( a < b \land b < c \) then \( a < c \) (transitivity).

A strict order is total if:

- \( a < b \lor b < a \lor a = b \) (trichotomy).

From the trichotomy requirement given above it follows that:

- if \( a < b \) then \( b < a \) doesn’t hold (asymmetry).

The ordering should be generated automatically depending on the sorts that are declared in the mCRL2 specification and the ordering should meet the above requirements given for <.

A way to create such an ordering for an arbitrary sort is to create the ordering depending on the order of appearance of the constructors of a sort; this will be described in more detail in section 3.2. For each sort, the equations for < are generated, and the equations for > are written in terms of <. In the next section, a general scheme for generating an ordering for an arbitrary sort is given. The last section gives two examples of sort declarations and equations for < and > belonging to those sorts.
3.2. **General ordering scheme**

As already said in the previous section it is necessary to have an ordering for a sort that is to be used in a finite set in mCRL2. To prevent that all these ordering rules have to be created manually, it is helpful to have a general ordering scheme. This means that each ordering for each sort is generated according to the same scheme. Such a general scheme could look like the following:

Take the mCRL2 specification for the sort $S$ that has constructors $c_{-i},...,c_{-n}$ and possible arguments $a_{-i_j},...,a_{-i_m}$ and $b_{-i_k},...,b_{-i_p}$. With the following rules, all equations for $<$ can be generated:

1. $\forall i: 1 \leq i \leq n-1: \forall x: 1 \leq x \leq n-i: \forall j: 0 \leq j \leq m: \forall k: 0 \leq k \leq p:\n c_{-i}(a_{-i_j},...,a_{-i_m}) < c_{-i}(i+x)(b_{-i+k},...,b_{-i+p}) = true;$

2. $\forall i: 2 \leq i \leq n: \forall x: 1 \leq x \leq n-1: \forall j: 0 \leq j \leq m: \forall k: 0 \leq k \leq p:\n c_{-i}(a_{-i_j},...,a_{-i_m}) < c_{-i}(i-x)(b_{-i-k},...,b_{-i-p}) = false;$

3. $\forall i: 1 \leq i \leq n: \forall j: 0 \leq j \leq m: \forall k: 0 \leq k \leq p:\n c_{-i}(a_{-i_j},...,a_{-i_m}) < c_{-i}(b_{-i_k},...,b_{-i_p}) = a_{-i_j} < b_{-i_k} \land ((a_{-i_j} == b_{-i_k}) \& \& (a_{-i_k+1} < b_{-i_k+1})) \land ((a_{-i_j} == b_{-i_k}) \& \& (a_{-i_k+1} == b_{-i_k+1})) \& \& ...
\& \& ((a_{-i_j} == b_{-i_k}) \& \& (a_{-i_k+1} == b_{-i_k+1})) \& \& ...
\& \& (a_{-i_m} < b_{-i_p})$.

Rule number 1 says that for all $c_{-i},...,c_{-n-1}$, each constructor is compared to all its successors, i.e. for $n = 3$ this means that constructor $c_{-1}$ is compared to constructor $c_{-2}$ and $c_{-3}$, so with this rule, two equations are generated for $c_{-1}$: $c_{-1} < c_{-2} = true$; and $c_{-1} < c_{-3} = true$. For this rule, it doesn’t matter whether the constructor contains any arguments or not.

The second rule does the opposite of the first rule; it says that each constructor is compared to all its predecessors. E.g. for $n = 3$ it means that constructor $c_{-3}$ is compared to constructor $c_{-1}$ and $c_{-2}$, so for $c_{-3}$ two equations are generated with this rule: $c_{-3} < c_{-1} = false$; and $c_{-3} < c_{-2} = false$. Just as in rule 1, it doesn’t matter whether the constructor contains any arguments.

The third and last rule generates equations for the cases where two constructors that are the same are compared. It also takes into account that these constructors may contain arguments ($a_{-i_j},...,a_{-i_m}$ and $b_{-i_k},...,b_{-i_p}$). It states that when two constructors $c_{-i}$ are compared, it checks on its arguments, i.e. the first argument of the first $c_{-i}, a_{-i_j}$, is compared to the first argument of the second $c_{-i}$, which is $b_{-i_k}$. When $a_{-i_j}$ is smaller than $b_{-i_k}$ the result yields true. When this is not the case, a check is done whether $a_{-i_j}$
and $b_{i_k}$ are equal and $a_{i_{j+1}}$ is smaller than $b_{i_{k+1}}$. In this way, all elements of $c_i$ will be checked. When $a_{i_{j+1}}$ and $b_{i_{k+1}}$ are also equal, then the next arguments, $a_{i_{j+2}}$ and $b_{i_{k+2}}$ are compared. With this rule all arguments will be compared when necessary, because the last part of the rule says that when all arguments are equal, then the last argument $a_{i_m}$ has to be smaller than $b_{i_p}$.

With these three rules it is possible to generate equations that form an ordering for an arbitrary sort that is declared in mCRL2. Because of the ordering, it is then possible to use such an arbitrary sort in one of the finite set implementations that can be found later in this document.

For the opposite of $<$, namely the relational operator $>$, just one rule is necessary to generate exactly one equation for $>$:

$$\forall x, y : x > y = y < x;$$

This means that for each constructor, the equation for $>$ is described in terms of $<$.

The next section will describe two examples that make use of the general ordering scheme that is proposed above. The first example treats a simple form of recursion for two sorts. The second example will treat mutual recursion between three defined sorts.

3.3. Examples

In this section, two examples are given of how an ordering for arbitrary sorts could look. The first example deals with an ordering in which a simple form of recursion occurs. The second example has a more elaborate form of recursion, namely mutual recursion, within the sort declaration.

3.3.1. Example 1: Recursion

This example treats a simple form of recursion and consists of two sorts, $F$ and $G$. Both are declared as a `struct` where $F$ has one constructor $f1$ with a positive number ($\text{Pos}$) argument. $G$ has two constructors, the first constructor, $g1$, has a positive number ($\text{Pos}$) argument, whereas the second constructor, $g2$, has an $F$ argument and a $\text{Pos}$ argument. The sorts $F$ and $G$ are declared as follows:

```ml
sort
    F = struct f1(Pos);
    G = struct g1(Pos) | g2(F, Pos);
```
For both sorts, equations have to be generated for the relational operators < and >. When we follow the rules that are available for generating an ordering, we come to the following equations for $F$ and $G$:

The relation $<$ (which is called $lt$ here) for $F$:

- **Less than**, $lt : F \# F \rightarrow \text{Bool}$;

  ```
  map
  lt : F \# F \rightarrow \text{Bool};
  var
  x, y : Pos;
  eqn
  lt(f1(x), f1(y)) = x < y;
  ```

This equation is generated by rule number 3 and because $F$ consists of only one constructor, only one equation is generated.

Greater than (called $gt$) is specified as follows:

- **Greater than**, $gt : F \# F \rightarrow \text{Bool}$;

  ```
  map
  gt : F \# F \rightarrow \text{Bool};
  var
  p, q : F;
  eqn
  gt(p, q) = lt(q, p);
  ```

This greater than equation says that $p$ is greater than $q$ only if $q$ is less than $p$.

For $G$ the relation less than is:

- **Less than**, $lt : G \# G \rightarrow \text{Bool}$;

  ```
  map
  lt : G \# G \rightarrow \text{Bool};
  var
  x, y, z, z1 : Pos;
  p, q : F;
  eqn
  lt(g1(x), g1(y)) = x < y;
  lt(g1(x), g2(p, z)) = true;
  lt(g2(p, y), g1(z)) = false;
  lt(g2(p, y), g2(q, z1)) = lt(p, q) ||
                   ((p == q) && (y < z1));
  ```

Less than consists of four equations. This is because $G$ has two constructors and to make all the comparisons, this yields four equations. The first equation is generated by rule number 3, the second equation by rule number 1, the third by rule number 2 and the last equation by number 3 again. The greater than relation will not be described anymore
because it has exactly the same structure as for sort $F$ and is the same for any arbitrary sort.

### 3.3.2. Example 2: Mutual recursion

The second example describes an ordering for three sorts that have mutual recursion: $R$, $S$ and $T$. These sorts are declared as follows:

```plaintext
sort
  T = struct nill | node(T, T);
  R = struct leaf(Pos) | tree(S);
  S = struct zero | forest(R, S);
```

As can be seen, all three sorts contain two constructors; this means that for the less than relation of each of them, four equations have to be created. We start with generating the equations for $T$.

- **Less than**, $lt : T \times T \rightarrow \text{Bool}$

  ```plaintext
  map
  lt : T \times T \rightarrow \text{Bool};
  var
  s, t, u, v : T;
  eqn
  lt(nill, nill) = false;
  lt(nill, node(s, t)) = true;
  lt(node(s, t), nill) = false;
  lt(node(s, t), node(u, v)) = lt(s, u) ||
                         ((s == u) && lt(t, v));
  ```

  The first three equations are generated by rules number 3, 1 and 2, respectively. The fourth equation is also generated by rule number 3.

Less than for sort $R$:

- **Less than**, $lt : R \times R \rightarrow \text{Bool}$

  ```plaintext
  map
  lt : R \times R \rightarrow \text{Bool};
  var
  x, y : Pos;
  s, t : S;
  eqn
  lt(leaf(x), leaf(y)) = x < y;
  lt(leaf(x), tree(s)) = true;
  lt(tree(s), leaf(x)) = false;
  lt(tree(s), tree(t)) = lt(s, t);
  ```

  Equations 1 and 4 are generated by rule number 3, equation 2 by rule number 1 and equation 3 by rule number 2.
The last sort in this example is the sort $S$ and for $S$, less than is specified as follows:

- **Less than**, $lt : S \# S \rightarrow \text{Bool}$;

  ```
  map
  $lt : S \# S \rightarrow \text{Bool}$;
  var
  $r, t : R$;
  $s, u : S$;
  eqn
  $lt(zero, zero) = false$;
  $lt(zero, forest(r, s)) = true$;
  $lt(forest(r, s), zero) = false$;
  $lt(forest(r, s), forest(t, u)) = lt(r, t) | |
  ((r == t) && lt(s, u))$;
  ```

The first three equations are generated by rules number 3, 1 and 2, respectively. Equation 4 is also generated by rule number 3. For each of the sorts, also an equation greater than has to be generated in the way as is shown in section 3.3.1.
4. Finite set as ordered List

4.1. Ordered List

The first option chosen to represent finite sets in mCRL2 was to model a finite set as an ordered List. To model this finite set as ordered List, I chose to model a finite set of positive numbers, i.e. a finite set of the data type Pos that is available in mCRL2. The reason for choosing the data type Pos was the fact that this data type contains an ordering. This means that for the data type Pos, the relational operators < and > are available. This ordering is necessary to keep the List that represents the finite set ordered. As already said, for this chapter, and also for chapters 5, 6 and 7, the data type Pos is used but with the ordering as created in chapter 3, any sort could be used. In section 4.2 an overview is given of the mCRL2 specification of the finite set as ordered List. This overview contains functionality belonging to Lists and functionality specific for a finite set.

4.2. mCRL2 specification

The mCRL2 specification of the finite set as an ordered List starts with a sort definition, which will be described in section 4.2.1. The specification also contains functionality that is not specific for a finite set and functionality specific for a finite set. With functionality that is not specific for a finite set, the functions insert, delete, element_in and ordered are meant. With functionality specific for a finite set, functions such as cardinality, subset and union is meant.

When creating the functionality that is specific for a finite set, I didn’t use the function names that are used in the sort Set in mCRL2, i.e. the function for calculating the union of two finite sets is called union instead of ∪. The reason for choosing different names is the fact that it is not allowed to overrule these already existing names, i.e. it is not possible to generate an .lps-file from the mCRL2 specification when overruling these names. An option for the future would be to add the data type finite set to the mCRL2 language and extend its syntax, i.e. make it possible to use a finite set with function names such as ∪ for union, which can’t be used now.

4.2.1. Sort definition

The sort definition for the finite set as ordered List is specified as follows:

1. sort fsetList = List(Pos);

This line of mCRL2 code says that the sort fsetList, which is an abbreviation of finite set as List, is in fact a List of positive numbers, where the data type used, in this case
Finite sets in mCRL2  October 2007

Pos, is specified between curly brackets after the declaration of List. By specifying the sort in this way, the sort fsetList can be used throughout the rest of the specification.

4.2.2. General functionality

The specification contains not only functionality that is specific for finite sets but it also contains functionality that is more general. For each function specified, the mapping of the function and its equation(s) are described.

Insert

1. map
2. insert : Pos # fsetList -> fsetList;
3. var
4. p, q : Pos;
5. s : fsetList;
6. eqn
7. insert(p, [ ]) = p |> [ ];
8. insert(p, q |> s) = if(p < q,
9.   p |> (q |> s),
10.   %else
11.     if(p > q,
12.       q |> (insert(p, s)),
13.       %else
14.         q |> s));
15.   %endif
16. %endif

Insert is a function that can be used to add an element to a finite set at the correct position. This means that the finite set stays ordered throughout the process. The insert function takes a positive number and a finite set as input and returns a finite set. What can be seen in the equations is that the constructors for the data type List are used since a finite set is in fact a List of some type. The expected time for using this insert function is of $O(n)$, because insert is invoked recursively at most $n$ times for a List of length $n$.

The function insert consists of two equations. The first equation specifies what happens when a positive number is inserted into an empty finite set. The second equation is an insert of a positive number into a non-empty finite set. To keep the List ordered, this equation first checks whether the number to be inserted, $p$, is smaller than the first element in the set, $q$. When this happens to be the case, the number $p$ is attached to the front of the set and we are done. When $p$ is not smaller than $q$, another check is done that checks whether $p$ is bigger than $q$. When $p$ is bigger than $q$, a recursive call is done on the remainder of the set, thus the set without element $q$. When $p$ happens to be equal to $q$, then nothing happens and the original finite set with which the equation started is returned.

In section 8.1, a proof is given for the fact that after an insertion of an element into an ordered List, this element can be found in the ordered List, theorem 1. And a proof is
given for the fact that an ordered List is still ordered after inserting an element into that ordered List, this can be seen in theorem 3.

Delete

1. map
2. delete : Pos # fsetList -> fsetList;
3. var
4. p, q : Pos;
5. s : fsetList;
6. eqn
7. delete(p, []) = [];
8. delete(p, q |> s) = if(p < q,
9. q |> s,
10. %else
11. if(p > q,
12. q |> delete(p, s),
13. %else
14. s));
15. %endif
%endif

The delete function is meant to remove a certain element from a finite set. Delete takes a positive number and a finite set as its input and will return a finite set as result. Delete is built from two equations. The first says that deleting the number p from the empty finite set [] results in the empty finite set []. This makes sense because when a finite set is empty, there is just nothing to remove. The second equation specifies what happens when one tries to delete a positive number p from a non-empty finite set q |> s, where q is the first element of the finite set and s is the remainder. So every element in s is bigger than the element q. Just as with the insert function, delete is expected to run in O(n) time because a recursive call of delete is done at most n-1 times for a List of length n.

Delete first checks whether the element to be deleted, p, is smaller than the first element in the set, q. When this is the case, it means that p is not present in the set, since the set is in fact an ordered List. In this case, the original set is returned as result. The next if statement in the equation checks whether p is bigger than q. When this is the case, a recursive call of delete is done with as arguments the positive number p and the finite set s, i.e. the finite set without the element q. When the element p is found within the set, thus when p is equal to an element q, then the set s is returned as result, so the element q is removed from the original set q |> s.

Section 8.1 also contains proofs for the delete function. Theorem 2 from that section says that after deleting an element from an ordered List, that element can not be found any more in the resulting ordered List. Theorem 4 of section 8.1 says that deleting an element from an ordered List results in an ordered List again.

Element test

1. map
2.  element_in : Pos # fsetList -> Bool;
3. var
4.   p, q : Pos;
5.   s : fsetList;
6. eqn
7.   element_in(p, []) = false;
8.   element_in(p, q |> s) = if(p == q,
9.     true,
10.    %else
11.     if(p < q,
12.       false,
13.        %else
14.         element_in(p, s));
15.    %endif

The function that tests whether a given element is present in a set is called \texttt{element_in}. It returns \texttt{true} when the element asked for is present in the set and will return \texttt{false} when the element asked for is not present in the set. \texttt{Element_in} is also \textit{expected} to run in $O(n)$ time because a recursive call of \texttt{element_in} is done at most $n$ times for a List of length $n$.

The function that specifies the element test is called \texttt{element_in}. It is mapped with a positive number and a finite set as input and a Boolean as output. Just as \texttt{insert} and \texttt{delete}, \texttt{element_in} consists of two equations: one for searching through an empty set and one for searching through a non-empty set.

When searching for a number $p$ in an empty set $[]$, this function yields \texttt{false} because $p$ is not present, since no numbers are present in an empty set. When searching through a non-empty set $q |> s$ in line 8, where $q$ is a number and $q$ is followed by a finite set $s$ (which in itself can be empty or non-empty again), the function checks whether $p$ is equal to $q$. When that is the case, \texttt{true} is returned as result. When $p$ is smaller than $q$, see lines 11 and 12, then \texttt{false} is returned. When $p$ is not equal to $q$ and $p$ is not smaller than $q$, then a recursive call is done on $p$ and the finite set $s$.

\textbf{Ordered}

1. map
2. ordered : fsetList -> Bool;
3. var
4.   p, q : Pos;
5.   s : fsetList;
6. eqn
7.   ordered([]) = true;
8.   ordered(p |> []) = true;
9.   ordered(p |> (q |> s)) = (p < q) \&\& ordered(q |> s);

\texttt{Ordered} is a function that takes a finite set as input and returns \texttt{true} when the finite set is correctly ordered. When a finite set is not ordered, this function will return \texttt{false}. \texttt{Ordered} consists of three equations: the first one says that an empty set is always ordered, the second says that a set with only one element is also always ordered. And the
third and last equation specifies when a finite set with two or more numbers is ordered. This is the case when the first element is smaller than the following element, so \( p \) is smaller than \( q \), and when the remainder, \( q \mid> s \), is also ordered. The expected performance of the function ordered is of \( O(n) \) because each element is compared to its successor and the last element yields true in itself by the equation of line 8.

4.2.3. **Finite set specific functionality**

This subsection covers the part of the specification that deals with the functionality that is specific for finite sets.

**Cardinality**

```plaintext
1. map
2.  card : fsetList -> Nat;
3. var
4.   q : Pos;
5.   s : fsetList;
6. eqn
7.   card([]) = 0;
8.   card(q |> s) = card(s) + 1;
```

The cardinality of a finite set is the number of elements present in the set. The function for calculating the cardinality is called \( \text{card} \) and takes a finite set as input and returns the number of elements in the form of a natural number. The first of the two equations specifies that the cardinality of an empty set is equal to 0. The second equation says that the cardinality of a finite set of the form \( q |> s \), is one (because of \( q \)) added to the cardinality of \( s \). The performance of the cardinality function is of \( O(n) \), because for each element 1 is added to the cardinality of the remaining List, so in total \( n + 1 \) calls on \( \text{card} \) are done to calculate the cardinality of an ordered List of length \( n \).

**Subset**

```plaintext
1. map
2.  subset : fsetList \# fsetList -> Bool;
3. var
4.   q : Pos;
5.   s, t : fsetList;
6. eqn
7.   subset([], s) = true;
8.   subset(q |> t, s) = element_in(q, s) && subset(t, s);
```

Subset is a function which determines whether a given finite set is a subset of another given finite set. Therefore it takes two finite sets as input and returns a \( \text{Boolean} \). This function consists of two equations. The first equation says that the empty set is always a subset of a non-empty set. The second equation says that a finite set \( q |> t \) is a subset of the finite set \( s \) when \( q \) is present in \( s \) (this is checked by the \( \text{element_in} \) function that we saw earlier) and when \( t \) is a subset of \( s \). The expected performance of the subset function
is $O(n^2)$ because for each element in the List, a call to the element_in function is done and that function performs in $O(n)$.

**Proper subset**

1. map
2.    proper_subset : fsetList # fsetList -> Bool;
3. var
4.    s, t : fsetList;
5. eqn
6.    proper_subset(s, t) = subset(s, t) && s != t;

The proper subset function checks whether a given finite set is a proper subset of another given finite set. It is mapped with two finite sets as input and a Boolean as output. The proper_subset function is built from just one equation. This equation says that a finite set $s$ is a proper subset of a finite set $t$ when $s$ is a subset of $t$ and $s$ is not equal to $t$. The function’s performance is the performance of the subset function, $O(n^2)$ and added to that a call on the inequality function $!=,$ so this yields a performance of $O(n^2)$.

**Union**

1. map
2.    union : fsetList # fsetList -> fsetList;
3. var
4.    q : Pos;
5.    s, t : fsetList;
6. eqn
7.    union([], s) = s;
8.    union(q |> t, s) = insert(q, union(t, s));

With the union function, it is possible to create a new set from two other sets. It takes two finite sets as input and returns one finite set as output. In this output all elements of the two input sets are present. This function also has two equations: the first equation says that the union of an empty set and a non-empty set results in that same non-empty set. The second equation says that when taking the union of two sets of the form $q |> t$ and $s$, the number $q$ is inserted into the union of $t$ and $s$. In this way, the resulting set contains all elements of $q |> t$ and $s$ in the correct order and without duplicates. For a set $t$ of length $n$, the performance of the union function is $n$ times the performance of the insert function, $O(n)$, so this yields a performance of $O(n^2)$.

**Difference**

1. map
2.    difference : fsetList # fsetList -> fsetList;
3. var
4.    q : Pos;
5.    s, t : fsetList;
6. eqn
7.    difference(s, []) = s;
8.    difference([], s) = [];

9. difference(q |> t, s) = if(element_in(q, s),
10.     difference(t, s),
11.     %else
12.     q |> difference(t, s));
13.     %endif

The difference function computes the set-theoretic difference of two finite sets, i.e. the difference between the sets $S$ and $T$ yields the set that contains the elements that are present in $S$ but that are not present in $T$, so: $x \in S \land x \notin T$.

As already said, the function takes two arguments, both finite sets, and yields one finite set as result. Difference has three equations: the first says that the difference between a non-empty set $s$ and the empty set yields the non-empty set $s$, the second equation says that the difference of the empty set and the non-empty set $s$ results in the empty set. And the last equation computes the difference between the finite set $q \triangleright t$ and $s$ as follows: when the element $q$, the first element of $q \triangleright t$, is present in $s$ then it can be omitted and a recursive call of difference is done on $t$ and $s$. When $q$ is not present in $s$, then $q$ is part of the result by adding $q$ to the front of the ordered List that represents the difference of $t$ and $s$. The performance of difference for a List $q \triangleright t$ of length $n$ and List $s$ of length $m$ is $O(n^2)$ because for each of the $n$ elements from $q \triangleright t$ a call of element_in is done at most.

Intersection

1. map
2.   intersection : fsetList # fsetList -> fsetList;
3. var
4.   q : Pos;
5.   s, t : fsetList;
6. eqn
7.   intersection(s, []) = [];
8.   intersection(s, q |> t) = if(element_in(q, s),
9.       q |> intersection(s, t),
10.      %else
11.         intersection(s, t));
12.     %endif

Intersection computes the intersection of two finite sets. This means that the resulting set only contains the elements that are present in both sets that serve as input. So intersection takes two finite set arguments as input and yields one finite set as output. The intersection of a non-empty set $s$ and the empty set results in the empty set, which is specified in the equation in line number 7. The intersection of the finite set $s$ with the finite set $q \triangleright t$ is computed in the following way: with the help of the function it is checked whether the element $q$, that is the first element of $q \triangleright t$, is part of the set $s$. When this is the case, the element $q$ is part of the resulting set by adding it to the front of the ordered List that represents the intersection of $s$ and $t$. When $q$ is not part of $s$, then it won’t be part of the result and a recursive call of intersection is done on $s$ and $t$ and $q$ is omitted. The performance of intersection for a List $q \triangleright t$ of length $n$
and List $s$ of length $m$ is $O(n^2)$ because for each of the $n$ elements from $q \mid\mid t$ a call of \texttt{element\_in} is done at most.

**Finite set to Bag**

1. map
2. \texttt{fsetList2Bag} : fsetList -> Bag(Pos);
3. var
4. \texttt{q} : Pos;
5. \texttt{s} : fsetList;
6. eqn
7. \texttt{fsetList2Bag}([\]) = \{\};
8. \texttt{fsetList2Bag}(q \mid\mid s) = \{q:1\} + \texttt{fsetList2Bag}(s);

Finite set to bag is a function that converts a finite set to a Bag; in this case it converts a finite set of positive numbers into a Bag of positive numbers. The function is named \texttt{fset2bag} and takes a finite set as input and yields a Bag of positive numbers as result. \texttt{Fset2bag} consists of two equations. The first equation, see line number 7, specifies that converting an empty set into a bag yields an empty Bag. The second equation, line 8, specifies what happens when a non-empty set $q \mid\mid s$ is converted to a Bag. Because the Bag data type can contain duplicates, it keeps track of how many times a certain element occurs, so when a positive number $r$ is present 10 times in a Bag, it is written as $r:10$. So when converting a finite set to a Bag, each element of the set is added once because the finite set doesn’t contain duplicates. First $q$ is added once and then a recursive call is done on the remainder of the set, $s$. Both these results are merged to form one Bag. The performance of this function is of $O(n)$, because for an ordered List of length $n$, for each element a call of the Bag enumerator {\} is done.

**Finite set to Set**

1. map
2. \texttt{fsetList2Set} : fsetList -> Set(Pos);
3. var
4. \texttt{q} : Pos;
5. \texttt{s} : fsetList;
6. eqn
7. \texttt{fsetList2Set}([\]) = \{\};
8. \texttt{fsetList2Set}(q \mid\mid s) = \{q\} + \texttt{fsetList2Set}(s);

The last function in this specification is a conversion function that converts a finite set to an infinite Set, this function is called \texttt{fsetList2set}. It takes a finite set as input and results in an infinite Set of the same type, which in this case is a Set of positive numbers because the finite set contains positive numbers. The two equations specify what happens when converting an empty finite set to a Set and when converting a non-empty finite set to a Set.

Looking at the first equation we see that converting an empty finite set to a Set results in an empty infinite Set. When we look at the second equation we see that the finite set of the form $q \mid\mid s$ is converted as follows: first $q$ is added to an empty Set resulting in
\{q\}, to this result, the result of the conversion of \( s \) is added by using the + operator that is used to concatenate two infinite sets. The performance of this function is of \( O(n) \), because for an ordered List of length \( n \), for each element a call of the Set enumerator \( \{} \) is done.
5. Finite set as AVL tree

5.1. AVL tree overview

An AVL tree is a self-balancing binary search tree and its name comes from its two inventors, G.M. Adelson-Velsky and E.M. Landis. In an AVL tree, the heights of the two child sub trees of any node differ by at most one. Therefore, an AVL tree is called height-balanced.

Every node in the AVL tree used to model a finite set has a height parameter. This parameter is used to calculate the balance factor of a node. This balance factor is calculated by subtracting the height of the left sub tree of a node from the height of the right sub tree of that node. When a node has a balance factor of -1, 0 or 1, the node is considered balanced. In an AVL tree, an insert or a removal of an element could be followed by one or more tree rotations that preserve the balance in the tree.

Balancing is a method to maintain a good performance, i.e. $O(\log (n))$, because the heights of two sub trees of a node will not differ more than 1. In this way it will never occur that a node has a left sub tree with height 20 and a right sub tree of 0, and each node in the left sub tree only has a left sub tree and an empty right sub tree. When this would be possible, the performance of inserting, deleting or searching for an element would decrease drastically, i.e. go to $O(n)$. More information on AVL trees can be found in [2], [6] and [8].

To model a finite set as an AVL tree, first a sort definition is needed. The specification also needs to have AVL tree specific functions and finite set specific functions. Added to that, some helper functions are specified. In the sections 5.3.1, 5.3.2 and 5.3.3, all these functions will be described.

In the remainder of this chapter, the following subjects are covered: in section 5.2, I will describe the tree rotations that an AVL tree uses to preserve its balance and after that, in section 5.3, the mCRL2 specification is treated.

5.2. Rebalancing

As already said in section 5.1, AVL trees use tree rotations to preserve its balance. This is due to the fact that after the insertion of a new element, or after the removal of an element, one or more nodes can become unbalanced, i.e. their balance factor is bigger than 1 or smaller than -1. AVL trees use four different tree rotations: a single left rotation, a double left rotation, a single right rotation and a double right rotation. All these rotations will be described in the following sections.
Single left rotation

When the balance factor of a node becomes bigger than 1 after an insert or a delete operation and that node’s right child has a balance factor of 0 or 1, then a single left rotation has to be done on the node to restore the balance. In Figure 5.1 an example of a tree is given that needs to be rebalanced by a single left rotation.

![AVL tree before single left rotation](image)

As can be seen in Figure 5.1, the node with label q has a balance factor of 2. This is calculated by subtracting the height of the left sub tree of q, which is 1, from the height of the right sub tree of q, which is 3. To rebalance this tree with a single left rotation, the following steps are taken:

1. The node with label R becomes the new root of the tree;
2. The node with label R takes the node with label q as its left child;
3. The node with label q takes the left child of the node with label R, which is RL, as its right child.
When these steps are taken and thus a single left rotation is done, the tree is balanced again. This rebalanced tree can be seen in Figure 5.2.

5.2.2. **Double left rotation**

Another kind of rotation is the double left rotation. The double left rotation is necessary when a node has a balance factor that is bigger than 1 and when this node has a right sub child that has a balance factor that is smaller than, or equal to -1. An example of such a tree can be found in Figure 5.3.
Figure 5.3 AVL tree before double left rotation

The AVL tree in Figure 5.3 has a node, labeled with q, with a balance factor bigger than 1 and that node q has in its turn a right sub child node with a balance factor of -1. The double left rotation performs the following steps to rebalance the tree:

1. A single right rotation on the right sub tree;
2. A single left rotation on the complete tree.

When these two steps are taken, the tree is balanced again, this tree can be found in Figure 5.4.
5.2.3. Single right rotation

When the balance factor of a node becomes smaller than -1 after an insert or delete operation and its left child has a balance factor of -1 or 0, then a single right rotation is needed to restore the balance. In Figure 5.5 an example is given of a tree that needs to be rebalanced by a single right rotation.
In Figure 5.5, the node with label q has a balance factor of 2; this means that this node is unbalanced. When looking at the left sub tree, it can be seen that the node with label L has a balance factor of -1; this means that a single right rotation has to be done to balance the tree. This single right rotation consists of the following steps:

1. The node with label L becomes the new root of the tree;
2. The node with label L takes the node with label q as its right child;
3. The node with label q takes the right child of the node with label L, LR, as its left child.

The result of the single right rotation on the tree from Figure 5.5 can be seen in Figure 5.6.

![Figure 5.6 AVL tree after single right rotation](image)

**5.2.4. Double right rotation**

The last tree rotation function that is used in the AVL tree structure is the double right rotation. This kind of rotation is necessary when after an insert or delete operation a node gets a balance factor smaller than -1 and when that particular node also has a left child node with a balance factor that is bigger than, or equal to 1. An example of such an AVL tree is given in Figure 5.7.
The tree in Figure 5.7 is unbalanced because the node with label q has a balance factor of -2. Because the node with label q also has a left child node with a balance factor that is equal to 1, a double right rotation is needed to rebalance the tree. A double right rotation does the following two things:

1. First a single left rotation on the left sub tree is done;
2. Then a single right rotation on the complete tree is done.

The result of a right rotation on the tree from Figure 5.7 can be found in Figure 5.8.
5.3. **mCRL2 specification**

In this section, the mCRL2 specification of the finite set as AVL tree will be given. This implementation is an implementation that is accepted by the mCRL2 parser and type checker and can be used to represent finite sets in mCRL2. First the sort definition of the finite set is described, after that the functionality that is specific for AVL trees is covered and in the last section the functionality that is specific for finite sets is described.

### 5.3.1. **Sort definition**

The AVL tree implementation of the finite set is represented by the sort fset. The sort fset is modeled as a structured data type consisting of two constructors. The way it is specified in the mCRL2 specification is:

```
1. sort fsetAVL = struct empty | node(Pos, fsetAVL, fsetAVL, Nat);
```

This line says that the sort fset has two possibilities to represent: either it is an empty tree or it is a node with a key, represented by the positive number since the finite set was modeled as a finite set of positive numbers, a left and a right sub tree that are both of the type fsetAVL and a natural number parameter specifying the height of the tree. The height of a tree represents the length of the longest path from the node to a leaf. The height parameter is part of the node because it is used to calculate balance factors. This height parameter was added to the node structure to prevent that each time a balance factor has to be calculated, the height of a node had to be calculated again and that is an expensive operation.

![Figure 5.8 AVL tree after double right rotation](image-url)
5.3.2. **AVL tree specific functionality**

The functions that are treated in this section are the functions that are specific for an AVL tree. The functions that are covered here are: `singleleft`, `doubleleft`, `singleright`, `doubleright`, `rebalance`, `insert`, `delete`, `element_in`, `ordered`, `allsmaller` and `allbigger`. Also some general functions are described and those are: `key`, `left`, `right`, `minimum`, `height` and `get_height`.

**Single left rotation**

1. map
2. `singleleft : fsetAVL -> fsetAVL;`
3. var
4. `p, q, qr : Pos;`
5. `l, r, lr, rr : fsetAVL;`
6. `h, hr : Nat;`
7. eqn
8. `singleleft(node(q, l, node(qr, lr, rr, hr), h)) =`
   `node(qr, new_l, rr,`  
   `max(get_height(new_l), get_height(rr)) + 1)`
9. `whr new_l = node(q, l, lr,`  
   `max(get_height(l), get_height(lr)) + 1)`
10. `end;`

The single left rotation has already been mentioned in section 5.2.1 and here we see the corresponding mCRL2 code for that function. The function `singleleft` takes a finite set as its input and results again in a finite set. `singleleft` consists of only one equation, since it is only used on non-empty sets. This equation creates a new tree from the tree that is given as input. As already said before, it uses three steps to calculate the new tree:

1. The root of the right sub tree becomes the new root of the tree, in this case the node with label `qr`;
2. The node with label `qr` takes the original left sub tree as its left sub tree;
3. The node with label `q` takes the left child of the node with label `qr, lr`, as its right child.

In this equation, I made use of a where clause that specifies a value `new_l` that specifies the left sub tree after rotation. `new_l` can be calculated once and can then be used multiple times in the equation without recalculating it every time it appears.

The performance of the single left rotation is of $O(1)$ because building a new sub tree from a given node can be done in constant time since only the functions `get_height`, which in itself operates in $O(1)$ time, and `max` are used.

For the single left rotation function, properties are proved in section 8.2. Lemma I in that section proves that if an element is present in a finite set as AVL tree, then it is still
present after applying a single left rotation on that tree. In Lemma VI it is proven that if an element is not present in a finite set, then it is also not present after a single left rotation.

When a finite set is ordered, then it will still be ordered after a single left rotation. This is proven in Lemma XI. In Lemma XVI a proof is given for the fact that if all elements in a tree are smaller than a certain value, then all these elements are still smaller after applying a single left rotation on the tree. Lemma XX of section 8.2 proofs that if all elements in a tree are bigger then a certain value, that they are still bigger after a single left rotation.

**Double left rotation**

1. map
2. doubleleft : fsetAVL -> fsetAVL;
3. var
4. q : Pos;
5. l, r : fsetAVL;
6. h : Nat;
7. eqn
8. doubleleft(node(q, l, r, h)) =
9. singleleft(node(q, l, singleright(r), h));

The second tree rotation function that can be used is the double left rotation. The name of that function is `doubleleft`. It takes a finite set as input and yields a finite set as output. Like the `singleleft` function, `doubleleft` also has just one equation because it is only used on non-empty sets. This equation just says that first a single right rotation has to be done on the right sub tree and that a single left rotation has to be done on the resulting tree. The performance of the double left rotation is of $O(1)$ because it is built from a single left rotation and a single right rotation that both operate in $O(1)$ time.

With regard to the double left rotation function, proofs are given in section 8.2. The first proof that is given for the double left rotation is given in Lemma III which states that if an element is present in a tree, then that element is still present in that tree after applying a double left rotation. Lemma VIII proofs that if an element is not present in a tree, then it is still not present after a double left rotation of that tree.

The next proof for double left is given in Lemma XIII, which says that an ordered finite set is still ordered after a double left rotation. So the double left rotation preserves order. Lemma XVIII states that if all elements in a tree are smaller than a certain value, then all these elements are still smaller than that value after a double left rotation. The opposite is proven in Lemma XXII, namely if all elements in a tree are bigger than a certain value, then they are still bigger after applying a double left rotation.

**Single right rotation**

1. map
2. singleright : fsetAVL -> fsetAVL;
3. var
4. q, ql : Pos;
5. l, r, ll, rl : fsetAVL;
6. h, hl : Nat;
7. eqn
8. singleright(node(q, node(ql, ll, rl, hl), r, h)) =
9. node(ql, ll, new_r,
    max(get_height(ll), get_height(new_r)) + 1)
10. whr new_r = node(q, rl, r,
    max(get_height(rl), get_height(r)) + 1)
11. end;

Singleright takes care of the single right rotation that can be performed on an AVL tree. It takes a finite set and delivers a finite set. The equation of singleright uses three steps to calculate the new tree:

1. The root node of the left sub tree becomes the new root of the tree, in this case that is ql;
2. The new root ql takes the original root node as its right child;
3. The original root node q takes the right sub tree of the original left sub tree as its left sub tree.

The performance of the single right rotation is of $O(1)$ because building a new sub tree from a given node can be done in constant time since only the functions get_height, which in itself operates in $O(1)$ time, and max are used.

In section 8.2, proofs are also given for the single right rotation function. The first proof about the single right rotation function is given in Lemma II which says that if an element is present in a finite set, then it will still be present after applying a single right rotation on the tree that represents that set. Lemma VII says that if an element is not present in a tree, then it will still not be present after a single right rotation.

In Lemma XII of section 8.2, a proof is given for the fact that if a finite set as AVL tree is ordered, then this finite set is still ordered after a single right rotation of that tree. Lemma XVII says that if all elements in a tree are smaller than a certain value, then they are also smaller after a single right rotation and Lemma XXI proves that if all elements are bigger than a certain value, they are still bigger after a single right rotation.

Double right

1. map
2. doubleright : fsetAVL -> fsetAVL;
3. var
4. q : Pos;
5. l, r : fsetAVL;
6. h : Nat;
7. eqn
8. doubleright(node(q, l, r, h)) =
9. singleright(node(q, singleleft(l), r, h));
The last of the tree rotation functions that are used in the AVL tree structure is the double right rotation. This function is called \texttt{doubleright} and takes a finite set as input and has a finite set as output. The equation in lines 8 and 9 says that in case of a double right rotation first a single left rotation on the left sub tree has to be done and after that a single right rotation has to be done on the result. The cost for a double right rotation is built from the cost for a single right rotation on a tree single left rotation on a tree. This comes to a total performance of $O(1)$.

For the fourth and last of the rotation functions of the AVL tree, proofs are also given in section 8.2, starting with a proof in Lemma IV for the fact that if an element is present in a tree, then it is still present after a double right rotation on the tree. The next proof for this function is given in Lemma IX which says that if an element is not present in a tree, then it will also not be present after applying a double right rotation on the tree.

Lemma XIV states that an ordered finite set as AVL tree is still ordered after a double right rotation on that tree. In Lemma XIX a proof is given for the fact that if all elements in a tree are smaller than a certain value that they are still smaller after a double right rotation and the last lemma for this function, Lemma XXIII says that if all elements in a tree are bigger than a certain value, then they are also bigger after a double right rotation.

\textbf{Rebalance}

1. map
2. rebalance : fsetAVL -> fsetAVL;
3. var
4. q : Pos;
5. l, r : fsetAVL;
6. h : Nat;
7. eqn
8. rebalance(empty) = empty;
9. rebalance(node(q, l, r, h)) =
10. if((get_height(r) - get_height(l)) > 1,
11. if((get_height(right(r)) - get_height(left(r))) \leq -1,
12. doubleleft(node(q, l, r, h)),
13. %else
14. singleleft(node(q, l, r, h))),
15. %else
16. if((get_height(r) - get_height(l)) < -1,
17. if((get_height(right(l)) - get_height(left(l))) \geq 1,
18. doubleright(node(q, l, r, h)),
19. %else
20. singleright(node(q, l, r, h))),
21. %else
22. node(q, l, r, max(get_height(l), get_height(r)) + 1));

\texttt{Rebalance} is an important function in the AVL tree structure because it checks whether the tree needs to be rebalanced after an \texttt{insert} or a \texttt{delete} operation. It performs some checks on the balance factors of the nodes and based on these balance factors it calls a rotation function when necessary. \texttt{Rebalance} takes a finite set as input and yields a finite
set as result. The first of its equations says that rebalancing an empty set yields an empty set (line 8).

The second equation rebalances a non-empty set by performing checks on the balance factors of the nodes (calculated by using the height parameters of the nodes). This equation starts with an if-statement that checks whether the height of the right sub tree minus the height of the left sub tree is bigger than 1. When this is the case, the balance factor of the root node of the right sub tree is calculated. When that balance factor is smaller than or equal to -1, a double left rotation is done (line 12), otherwise a single left rotation is done (line 14).

So far the case where the balance factor of the root node is bigger than 1. When this balance factor is not bigger than 1, a new check is done whether the balance factor is smaller than -1. When that is the case, the balance factor of the root node of the left sub tree is calculated. When that balance factor is bigger than or equal to 1, a double right rotation is done (line 18); otherwise a single right rotation is done (line 20).

When the balance factor of the root node of the tree is -1, 0 or 1 then nothing happens, only the height of the node is recalculated because an element can be inserted or deleted without disturbing balance factors (line 22).

The performance of the rebalance function depends on five possibilities, for each call of the rebalance, one of the four rotations is done, or when nothing needs to be rotated only the height of the node is recalculated. So when a single left rotation is done, the performance is of $O(1)$, when a double left rotation is done then the performance will be of $O(1)$ and in the case of a single right rotation it will also be of $O(1)$. For the case where a double right rotation is done, the performance is of $O(1)$ and when only the height is recalculated then the performance is of $O(1)$. So a rebalance of one node can be done in constant time.

With regard to the rebalance function, proofs are given in section 8.2. In Lemma V, a proof is given for the fact that if an element is present in an AVL tree, i.e. it can be found with the element_in function, then this element can still be found after rebalancing the tree. In Lemma X is proven that if an element is not present in an AVL tree, then it will also not be present after rebalancing the tree.

Lemma XV gives a proof for the fact that a finite set that is ordered is still ordered after rebalancing the AVL tree representing that set. In Lemma XXIV a proof is given for the fact that when all elements in an AVL tree are smaller than a certain value, then these elements are still smaller after rebalancing the AVL tree. Lemma XXV proves that when all elements in an AVL tree are bigger than a certain value, then they are still bigger after rebalancing that tree.
Insert

1. map
2. insert : Pos # fsetAVL -> fsetAVL;
3. var
4. p, q : Pos;
5. l, r : fsetAVL;
6. h : Nat;
7. eqn
8. insert(p, empty) = node(p, empty, empty, 1);
9. insert(p, node(q, l, r, h)) =
10. if((p == q),
11. node(q, l, r, h),
12. %else
13. if((p < q),
14. rebalance(node(q, insert(p, l), r, h)),
15. %else
16. rebalance(node(q, l, insert(p, r), h)));

The insert function is, as its name already tells, used for inserting elements into the tree. Insert is built in such a way that the tree that is the result of the application of insert is a correct AVL tree again, i.e. the tree is ordered and the balance factor of each node is -1, 0 or 1.

Insert takes a positive number and a finite set as its parameters. The positive number represents the element that should be added and the finite set is the tree to which the number is added. As can be seen in the code example, insert is built from two equations.

The first one specifies that inserting a positive number \( p \) into an empty set yields a node with key \( p \), an empty left sub tree, an empty right sub tree, and a height of value 1. What happens when an insert is done on a non-empty set is specified in the second equation that starts in line 9. This equation starts with an if-statement that checks whether the positive number to be added, \( p \), is equal to the key of the root of the tree, \( q \). When this is the case, the function terminates and returns the original tree since no duplicates are allowed. When \( p \) is not equal to the key \( q \), then another if-statement is opened that checks whether \( p \) is smaller than \( q \). When that is the case, a recursive call of insert is done with parameters \( p \) and the left sub tree \( l \). As can be seen, this recursive call is enveloped by a call of the function rebalance, this is necessary to make sure that the tree is balanced after the insertion of element \( p \) again.

The last possibility is that when the element \( p \) is bigger than \( q \), then a recursive call of insert is done with as parameters the positive number \( p \) and the right sub tree \( r \). This call is also enveloped by a call of the rebalance function for the same reasons as for the left sub tree.

It is expected that the insert function runs in \( O(\log n) \) time because of the fact that a recursive call of insert occurs at most \((\log n) + 1\) times for an AVL tree containing \( n \) elements. Some extra overhead is generated because each call of insert is also
enveloped by a call of the rebalance function, so for each insert, rebalance is called one time and within rebalance a call is done on singleleft, singleright, doubleleft or doubleright when rotating is necessary.

For the insert function on an AVL tree, some properties are proven in theorems that are given in section 8.2. For the insert function it is proven in Theorem 1 that after inserting a certain element into a tree, this element will be found by using the element_in function, i.e. after an insert, an element is present in the tree.

In Theorem 3 a proof is given that if all elements in a tree are smaller than a certain value, and an insert is done of an element that is also smaller than that certain value, then all elements of the resulting tree are smaller than that certain value. Theorem 4 states that if all elements in a tree are bigger than a certain value and another value is inserted into the tree that is also bigger, then all elements in the resulting tree are bigger. The fact that an ordered tree is still ordered after insertion of an element is proven in Theorem 5.

**Delete**

1. map
2. delete : Pos # fsetAVL -> fsetAVL;
3. var
4. p, q : Pos;
5. l, r : fsetAVL;
6. h : Nat;
7. eqn
8. delete(p, empty) = empty;
9. delete(p, node(q, l, r, h)) =
10.   if(p < q,
11.      rebalance(node(q, delete(p, l), r, h)),
12.      %else
13.      if(p > q,
14.       rebalance(node(q, l, delete(p, r), h)),
15.       %else
16.       if(l == empty && r == empty,
17.        empty,
18.        %else
19.        if(l == empty && r != empty,
20.         r,
21.         %else
22.         if(r == empty && l != empty,
23.          l,
24.          %else
25.          rebalance(node(minimum(r), l, delete(minimum(r), r), h))))))));

Delete is the function that is used to remove an element from a finite set, and thus from a tree. It is mapped as a function with two parameters, a positive number and a finite set and gives a finite set as result. Just as the insert function, delete consists of two equations. The first of these equations says that deleting an element from an empty set yields an empty set again. The second of the equations specifies what happens when one tries to delete an element from a non-empty set.
The equation for deleting a positive number from a non-empty set starts with an if-statement (this equation starts in line 9), which first checks whether the element to be deleted, \( p \), is smaller than the key of the root node of the tree, \( q \). When this is the case, a recursive call of \texttt{delete} is done with the parameters \( p \) and the left sub tree \( l \). This recursive call is, like with the \texttt{insert} function, enveloped in a call of the \texttt{rebalance} function to make sure that the trees stays balanced.

When \( p \) is not smaller than \( q \), it is checked whether \( p \) is bigger than \( q \). When that is the case, again a recursive call of \texttt{delete} is done, but now it is done on the right sub tree. This recursive call is also enveloped by a call of the \texttt{rebalance} function.

The last case that can occur is that when \( p \) is equal to \( q \), i.e. the element to be deleted is found and can be deleted. When \( p \) is equal to \( q \), four cases are distinguished:

1. Both the left and the right sub tree of the node to be deleted are empty: it is safe to delete the element and the result yields empty (lines 16 and 17);

2. The left sub tree is empty and the right sub tree is not empty: because the left sub tree of the node to be deleted is empty, the node to be deleted will be replaced by its right sub tree (lines 19 and 20);

3. The left sub tree is not empty and the right sub tree is empty: because the right sub tree of the node to be deleted is empty, the node to be deleted is replaced by its left sub tree (lines 22 and 23);

4. The left sub tree is not empty and the right sub tree is not empty: in this case, the node to be deleted is replaced by the smallest element present in its right sub tree. This node gets as left sub tree the left sub tree of the deleted node and as right sub tree the right sub tree of the deleted node without the smallest element of that right sub tree (since that smallest element has become the new root and no duplicates are allowed) (line 25).

\texttt{Delete} is also a function that is expected to run within \( O(\log n) \) time because a recursive call of \texttt{delete} is done at most \( (\log n) + 1 \) times for an AVL tree with \( n \) elements added to that are the costs of finding a possible minimum which can also be done in \( O(\log(n)) \) time. Just as with \texttt{insert}, some overhead is present because recursive calls of \texttt{delete} are enveloped by a call of the \texttt{rebalance} function, which in its turn does a call to one of the four rotation functions when necessary.

For the delete function, proofs are given in section 8.2. Theorem 2 of that section states that whenever an element is deleted from an ordered tree, then that element cannot be found anymore after the delete. Theorem 6 says that if all elements in a tree are smaller than a certain element and one of those elements is deleted, then all elements in the resulting tree are still smaller than that value. In theorem 7 this fact is proven for the \texttt{allbigger} function, namely when all elements in a tree are bigger than some value, and one of the elements from the tree is deleted, then all elements from the resulting tree are
still bigger than that value. The last theorem about the delete function is Theorem 8 which states that after a delete from an element from an ordered tree, the resulting tree is still ordered.

Element test

1. map
element_in : Pos # fsetAVL -> Bool;
2. var
3. p, q : Pos;
4. l, r : fsetAVL;
5. h : Nat;
6. eqn
element_in(p, empty) = false;
element_in(p, node(q, l, r, h)) =
7. if(p == q,
8.   true,
9. %else
10.   if(p < q,
11.     element_in(p, l),
12. %else
13.     element_in(p, r));

The element test function element_in is used to search for an element in a finite set. It takes a positive number and a finite set as input and returns a Boolean value that is true when the specified element is present in the set and false when the specified element is not present in the set.

The first of the two equations from which element_in is built says that searching for an element in an empty set always yields false. The second equation starts with an if-statement and checks whether the element searched for is equal to the key of the root of the tree. When that is the case, we are done because the element is found. The second possibility is that those numbers are not equal. Then a check whether p is smaller than the key of the root node is done, in the form of another if-statement. When p actually is smaller than q, we know that a search through the left sub tree is sufficient because all elements in the right sub tree are bigger than q, so they are also bigger than p. So in this case a recursive call of element_in is done with p and the left sub tree l as its input. The third possibility is that p is bigger than q. Then a recursive call is done on the right sub tree. Element_in should also run in O(log n) time, because at most log n recursive calls of element_in are done for an AVL tree of n elements.

Ordered

1. map
2. ordered : fsetAVL -> Bool;
3. var
4. p, q : Pos;
5. l, r : fsetAVL;
6. h : Nat;
7. eqn
8. ordered(empty) = true;
9. ordered(node(q, l, r, h)) = ordered(l) && ordered(r) &&
   allsmaller(l, q) && allbigger(r, q);

Ordered is a function that checks whether all elements in the tree (and thus in the finite set) appear in the correct order. Ordered has two equations. The first one says that an empty set is always ordered. The second equation says that a non-empty set is ordered when:

1. Its left sub tree is ordered;
2. Its right sub tree is ordered;
3. All elements in the left sub tree are smaller than the key of the root node of the tree;
4. All elements in the right sub tree are bigger than the key of the root node of the tree.

Ordered is of $O(n)$ because both allsmaller and allbigger can be calculated in $O(n)$ time.

All smaller

map
2. allsmaller : fsetAVL # Pos -> Bool;
var
3. p, q : Pos;
4. l, r : fsetAVL;
5. h : Nat;
eqn
8. allsmaller(empty, p) = true;
9. allsmaller(node(q, l, r, h), p) = q < p && allsmaller(l, p) &&
   allsmaller(r, p);

Allsmaller is a function that takes a finite set and a positive number and returns a Boolean value. This value is true when all elements of the set are smaller than the given number. When comparing an empty set to a positive number, this yields true. Comparing a non-empty set to a positive number gives true when:

1. The key of the root node of the tree, q, is smaller than the given positive number p;
2. All elements in the left sub tree l of the tree are smaller than p;
3. All elements in the right sub tree r of the tree are smaller than p.

For the function allsmaller it will take at most n steps to check whether all elements from finite set of $n$ elements are smaller than a given value. This is because for each element it has to be checked whether that element is smaller than the given value against which is checked. Therefore, it is expected that allsmaller performs in $O(n)$ time.
All bigger

1. map
2.    allbigger : fsetAVL # Pos -> Bool;
3. var
4.    p, q : Pos;
5.    l, r : fsetAVL;
6.    h : Nat;
7. eqn
8.    allbigger(empty, p) = true;
9.    allbigger(node(q, l, r, h), p) = q > p && allbigger(l, p) &&
    allbigger(r, p);

Allbigger is the opposite of allsmaller and it checks whether all elements in a given tree are bigger than a given positive number. Therefore, allbigger takes a finite set argument and a positive number argument. When comparing an empty set to a number this yields true. Comparing a non-empty set to a given number gives true when:

1. The key of the root node of the tree, q, is bigger than the given positive number p;
2. All elements in the left sub tree l of the tree are bigger than p;
3. All elements in the right sub tree r of the tree are bigger than p.

For a finite set of n elements, the function allbigger takes at most n steps to check whether all elements in a given finite set are bigger than a certain value, because for each value in the finite set it has to be checked whether this value is bigger than the given value. Therefore, this function is expected to perform in \(O(n)\) time.

Key

1. map
2.    key : fsetAVL -> Pos;
3. var
4.    q : Pos;
5.    l, r : fsetAVL;
6.    h : Nat;
7. eqn
8.    key(node(q, l, r, h)) = q;

The function key is used to retrieve the key of a node. Therefore key takes a finite set as input and gives a positive number as output. Key only has one equation saying that the key of a node(q, l, r, h) is equal to q. The function key is expected to perform in \(O(1)\) time because it only asks for the value of parameter q of node(q, l, r, h).

Left

1. map
2.    left : fsetAVL -> fsetAVL;
3. var
4.    q : Pos;
5.    l, r : fsetAVL;
6. h : Nat;
7. eqn
8. left(empty) = empty;
9. left(node(q, l, r, h)) = l;

Left is a function that is used to retrieve the left sub tree of a given tree. It takes a finite set as input and delivers a finite set as output. The left of an empty node yields empty and the left of a node(q, l, r, h) yields l. Performance of this function is of $O(1)$ because the only thing that is done is that the value of parameter l of node(q, l, r, h) is asked for.

Right

1. map
2. right : fsetAVL -> fsetAVL;
3. var
4. q : Pos;
5. l, r : fsetAVL;
6. h : Nat;
7. eqn
8. right(empty) = empty;
9. right(node(q, l, r, h)) = r;

Right is the function that returns the right sub tree of a given tree. Therefore it takes a finite set as input and returns again a finite set. The right sub tree of an empty node is empty and the right of a node(q, l, r, h) is always equal to r. This function performs the same as the left function because it only asks for a value of a parameter of node(q, l, r, h), namely r. Therefore it performs in $O(1)$ time.

Minimum

1. map
2. minimum : fsetAVL -> Pos;
3. var
4. q : Pos;
5. l, r : fsetAVL;
6. h : Nat;
7. eqn
8. minimum(node(q, l, r, h)) =
9. if(l != empty,
10. minimum(l),
11. %else
12. q);

Minimum is used to find and return the smallest element present in the tree. It takes a finite set and returns the smallest element in the form of a positive number. Minimum has one equation that consists of an if-statement that checks whether the left sub tree of the given tree is empty. When the left sub tree is not empty, a recursive call of minimum on the left sub tree. When the left sub tree is empty then the root element of the tree is returned as result. The function minimum is expected to perform in $O(\log(n))$ time.
because for a finite set of $n$ elements, it takes $\log(n)$ steps at most to find the minimum value in the set.

**Height**

1. map
2.    height : fsetAVL \rightarrow \text{Nat};
3. var
4.    q : \text{Pos};
5.    l, r : fsetAVL;
6.    h : \text{Nat};
7. eqn
8.    height(\text{empty}) = 0;
9.    height(node(q, l, r, h)) = (\max(\text{height}(l), \text{height}(r))) + 1;

**Height** is a function that can be used to calculate the height of node, i.e. the longest path from that node to a leaf. **Height** takes a finite set and returns a natural number. The first of its two equations says that the height of an empty set is 0. The second equation calculates the height of a non-empty set by adding 1 to the maximum of the heights of the left and right sub tree. The function **height** is added to use in Theorem 9 of section 8.2 where the correctness of **get_height** is proven. **Height** operates in $O(\log(n))$ time because at most $\log(n)$ steps are needed to find the longest path from a node to a leaf.

**Get height**

1. map
2.    get_height : fsetAVL \rightarrow \text{Nat};
3. var
4.    q : \text{Pos};
5.    l, r : fsetAVL;
6.    h : \text{Nat};
7. eqn
8.    get_height(\text{empty}) = 0;
9.    get_height(node(q, l, r, h)) = h;

**Get height** is a function that one can retrieve the value of the height parameter that is present in each node. Retrieving the height of an empty set yields 0, retrieving the height of the non-empty set $\text{node}(q, l, r, h)$ yields the value $h$. **get_height** is expected to perform in $O(1)$ time for it only asks the value of $h$ that is stored in $\text{node}(q, l, r, h)$. 
5.3.3. **Finite set specific functionality**

In this section, all functionality that is specific for finite sets can be found. For each function the mCRL2 code will be given and will be described.

**Cardinality**

1. map
2. card : fsetAVL -> Nat;
3. var
4. l, r : fsetAVL;
5. q : Pos;
6. h : Nat;
7. eqn
8. card(empty) = 0;
9. card(node(q, l, r, h)) = (card(l) + card(r)) + 1;

For calculating the number of elements present in a finite set, the function `card` is used. This function takes a finite set as input and returns a natural number that specifies the total number of elements. The first of its two equations says that the number of elements in an empty set is 0. The second equation calculates the number of elements for a non-empty set by adding 1 to the total cardinality of the left and the right sub tree together. This 1 that is added represents the root node of the tree. In this way the cardinality is calculated recursively for the whole tree. The cost for calculating the cardinality of a finite set of \( n \) elements will be of \( O(n) \) because for each element, 1 will be added to the total.

**Subset**

1. map
2. subset : fsetAVL # fsetAVL -> Bool;
3. var
4. q : Pos;
5. l, r, s : fsetAVL;
6. h : Nat;
7. eqn
8. subset(empty, s) = true;
9. subset(node(q, l, r, h), s) =
   element_in(q, s) && subset(l, s) && subset(r, s);

The `subset` function checks whether one finite set is a subset of another finite set by taking two finite sets and returning a Boolean. `Subset` contains two equations: the first one says that the empty set is always a subset of any other finite set, the second equation says that a finite set `node(q, l, r, h)` is a subset of the finite set `s` when the key of the root node, `q`, is present in `s` and the left sub tree `l` is a subset of `s` and the right sub tree `r` is a subset of `s`. `subset` does \( n \) calls of the `element_in` function, so that yields a performance of \( O(n \cdot \log(n)) \).
Proper subset

1. map
2. proper_subset : fsetAVL # fsetAVL -> Bool;
3. var
4. s, t : fsetAVL;
5. eqn
6. proper_subset(s, t) = subset(s, t) && s != t;

The function \textit{proper_subset} calculates whether a finite set is a proper subset of another finite set. For a finite set \( A \) to be a proper subset of a finite set \( B \), the following should hold: \( A \subseteq B \land A \neq B \). This means that \( A \) is a subset of \( B \) but \( A \) is not allowed to be exactly equal to \( B \).

The \textit{proper_subset} function thus takes two finite sets as input and returns a Boolean. The equation from which \textit{proper_subset} is built specifies that a finite set \( s \) is a proper subset of the finite set \( t \) when \( s \) is a subset of \( t \) and is not equal to \( t \).

The performance of the \textit{proper_subset} function is equal to the performance of \textit{subset}, which is of \( O(n \times \log(n)) \).

Union

1. map
2. union : fsetAVL # fsetAVL -> fsetAVL;
3. var
4. q : Pos;
5. l, r, s : fsetAVL;
6. h : Nat;
7. eqn
8. union(empty, s) = s;
9. union(node(q, l, r, h), s) = insert(q, union(l, union(r, s)));

\textit{Union} calculates the union of two finite sets. Therefore it takes two finite sets as input and gives one finite set as result. The first of the two equations from which \textit{union} is built says that the \textit{union} of the empty set and a set \( s \) yields the set \( s \). The second equation calculates the \textit{union} of a non-empty set \textit{node}(\( q, l, r, h \)) and a set \( s \) by merging the \textit{union} of \( l \) with \( s \) and the \textit{union} of \( r \) with \( s \), and added to that the root element \( q \). In this way all elements from \textit{node}(\( q, l, r, h \)) and \( s \) are present in the resulting set without any duplicates. The costs for a union of two sets of \( n \) and \( m \) elements respectively add to a total cost of \( O(n \times \log(n)) \) because \( n \) inserts are done for the set with \( n \) elements.

Difference

1. map
2. difference : fsetAVL # fsetAVL -> fsetAVL;
3. var
4. q, q1 : Pos;
5. l, r, s, l1, r1 : fsetAVL;
6. h, h1 : Nat;
Another finite set-specific function is `difference`. The result of the `difference` between the finite sets \( A \) and \( B \) is the set with those elements \( x \) for which it holds that: \( x \in A \land x \notin B \). `Difference` thus takes two finite sets as input and yields one finite set as output. There are three equations that are used to calculate the difference between two finite sets.

The first one says that the `difference` of the finite set \( s \) and the empty set yields just the set \( s \). The second one says that the `difference` between the empty set and the set \( s \) is the empty set again. The last equation (starting in line 10) checks whether the root node element \( q \) of \( \text{node}(q, 1, r, h) \) is present in \( \text{node}(q1, 1l, r1, h1) \). If this is the case, then the `difference` is taken from the set \( \text{node}(q, 1, r, h) \) without \( q \), and \( \text{node}(q1, 1l, r1, h1) \). When \( q \) is not present in \( s \) then \( q \) is part of the resulting set. This is done by taking the difference of \( \text{node}(q, 1, r, h) \) without \( q \) and inserting \( q \) afterwards back into the tree, in this way the result yields always a correct, i.e. balanced, AVL tree. The performance of difference is of \( O(n \times \log(n)) \) because for a tree with \( n \) elements, the `element_in` function is called \( n \) times. Added to that are the costs of a possible `delete` or `insert`, where `delete` is of \( O(\log(n)) \) and `insert` also is of \( O(\log(n)) \).

### Intersection

1. map
2. intersection : fsetAVL \# fsetAVL \rightarrow fsetAVL;
3. var
4. \( q, q1 : \text{Pos}; \)
5. \( 1, r, s, 1l, r1 : \text{fsetAVL}; \)
6. \( h, h1 : \text{Nat}; \)
7. eqn
8. intersection(empty, s) = empty;
9. intersection(node(q, 1, r, h), node(q1, 1l, r1, h1)) =
10. if(element_in(q, node(q1, 1l, r1, h1)),
11. insert(q, intersection(delete(q, node(q, 1, r, h)),
12. node(q1, 1l, r1, h1)));
13. %else
14. intersection(delete(q, node(q, 1, r, h)),
15. node(q1, 1l, r1, h1)));

The `intersection` function calculates the intersection of two finite sets. When calculating the `intersection` for the sets \( A \) and \( B \) the resulting set contains those elements \( x \) for which it holds that: \( x \in A \land x \in B \), i.e. only those elements that are present
in both sets. As already mentioned intersection takes two finite sets as input, gives one finite set as result and it has two equations.

The first equation of intersection specifies that intersecting the empty set with another finite set yields the empty set. The second equation (line 9) starts with an if-statement that checks whether the element $q$ from $\text{node}(q, l, r, h)$ is present in the finite set $\text{node}(q_1, l_1, r_1, h_1)$. When it is, $q$ is part of the result by inserting $q$ into the intersection of $\text{node}(q, l, r, h)$ without $q$ and $\text{node}(q_1, l_1, r_1, h_1)$, in this way, the result is always a correct AVL tree. When $q$ is not present in $\text{node}(q_1, l_1, r_1, h_1)$, a recursive call of intersection is done on $\text{node}(q, l, r, h)$ without $q$, and $\text{node}(q_1, l_1, r_1, h_1)$.

For the performance of the intersection function this is $n$ times a check for element_in, at most $n$ times insert, and at most $n$ times the costs of a delete. This yields a performance of $O(n \times \log(n))$.

**Finite set to Bag**

1. map
2. fsetAVL2Bag : fsetAVL -> Bag(Pos);
3. var
4. q : Pos;
5. l, r : fsetAVL;
6. h : Nat;
7. eqn
8. fsetAVL2Bag(empty) = {};
9. fsetAVL2Bag(node(q, l, r, h)) = \{q:1\} + (fsetAVL2Bag(l) + fsetAVL2Bag(r));

The conversion function $fsetAVL2Bag$ creates a Bag of elements from a given finite set. Therefore it takes a finite set as argument and returns a Bag. The conversion of an empty set yields an empty Bag. When converting a finite set $\text{node}(q, l, r, h)$ to a Bag, the root element $q$ is added once to an empty Bag and added to this Bag are the results of the conversion of the left sub tree $l$ to a Bag and the right sub $r$ to a Bag. This function performs in $O(n)$ time because for a finite set of $n$ elements, each element has to be added to the resulting Bag by calling the Bag enumerator $\{}$.

**Finite set to ordered List**

1. map
2. fsetAVL2List : fsetAVL -> List(Pos);
3. var
4. q : Pos;
5. l, r : fsetAVL;
6. h : Nat;
7. eqn
8. fsetAVL2List(empty) = [];
9. fsetAVL2List(node(q, l, r, h)) = fsetAVL2List(l) ++
   (q |> fsetAVL2List(r));
The second of the three conversion functions available in the AVL tree implementation of finite sets is the conversion from a finite set to a List. This function is called \texttt{fsetAVL2List} and takes a finite set as input and gives a List as output. The conversion of an empty set to a List gives the empty List $\emptyset$. When converting a non-empty set \texttt{node(q, l, r, h)} to a List, the result of the conversion of the left sub tree $l$ is concatenated to the result of the conversion of the right sub tree $r$, with $q$ in between. In this way the conversion yields an ordered List. The performance of this function is $O(n)$ because of the cost of the concatenation function $++$ that adds the resulting Lists together, and $++$ is called $n$ times for a finite set of $n$ elements.

### Finite set to Set

1. map  
2. \texttt{fsetAVL2Set} : fsetAVL \to Set(Pos);  
3. var  
4. \hspace{1em} q : Pos;  
5. \hspace{1em} l, r : fsetAVL;  
6. \hspace{1em} h : Nat;  
7. eqn  
8. \hspace{1em} fsetAVL2Set(empty) = \emptyset;  
9. \hspace{1em} fsetAVL2Set(node(q, l, r, h)) = \{q\} + (fsetAVL2Set(l) + fsetAVL2Set(r));

The last conversion function creates a Set as it is used now in mCRL2 from a finite set. The function is called \texttt{fsetAVL2Set} and takes a finite set as input and delivers a Set as output. The conversion of an empty set yields the empty Set $\emptyset$. Converting the finite set \texttt{node(q, l, r, h)} is done by creating the Set with root element $q$ and adding the result of the conversion of the left sub tree and the right sub tree to it. \texttt{fsetAVL2Set} performs in $O(n)$ time because for a finite set of $n$ elements, $n$ elements have to be added to the Set that is the result of this function.

### Equal

1. map  
2. \texttt{equal} : fsetAVL \# fsetAVL \to Bool;  
3. var  
4. \hspace{1em} s, t : fsetAVL;  
5. eqn  
6. \hspace{1em} equal(s, t) = subset(s, t) \&\& subset(t, s);

The \texttt{equal} function is used to determine whether two finite sets are exactly the same. Therefore it takes two finite sets as input and returns a Boolean. \texttt{Equal} is built from one equation, which compares two finite sets. This equation checks whether the two finite sets are equal by determining whether the first finite set argument is a subset of the second finite set argument and vice versa. The function \texttt{equal} performs in $O(n \times \log(n))$ time because it takes twice the performance of the \texttt{subset} function.
Inequal

1. map
2. inequal : fsetAVL # fsetAVL -> Bool;
3. var
4. s, t : fsetAVL;
5. eqn
6. inequal(s, t) = !(equal(s, t));

Inequal is the opposite of equal. Inequal yields true when two set are not exactly the same. Inequal is built from the equal function. This means that for calculating the inequality of two sets, the equality is calculated and a negation is done on that result, so false becomes true and vice versa. The performance of the inequal function is the same as that of the equal function, namely $O(n \times \log(n))$ because inequal is built from the equal function.
6. Finite set as left-balanced tree

6.1. Unique tree with left-balance

After developing the mCRL2 specification in which a finite set was implemented as an AVL tree, I came to the conclusion that it could be necessary to keep a tree unique to minimize the number of states in the space. Because an AVL tree is not unique for a finite set, e.g. the set \{1, 2, 3, 4\} can be represented by different AVL trees, the number of states could increase dramatically. It was not only necessary that the tree was unique, but also that the tree was balanced. This balanced-ness of the tree is necessary to get a better performance for the tree and finite set operations.

To overcome these possible problems, I made another specification of a finite set as a tree. This tree is a tree that is unique for a finite set, i.e. every set has exactly one corresponding tree. To achieve this, the tree that is used is filled from the left side only. This means that the height of the right sub tree of a node never exceeds the height of the left sub tree of that node. The way in which the tree is filled makes sure that the tree remains balanced. An example of a left-balanced tree can be found in Figure 6.1.

![Figure 6.1 Example of a left-balanced tree](image_url)

In Figure 6.1, an left-balanced tree can be seen where the left sub tree is completely filled and the right sub tree is partly filled. What also can be seen is that the right sub tree is filled from left to right. In this way, the height of the right sub tree never exceeds the height of the right sub tree. The way in which the tree is filled is described in detail when explaining the insert function of this tree by means of the mCRL2 code and of visual examples. The next section of this chapter, section 6.2, will cover the mCRL2 specification of the left-balanced tree.
6.2. **mCRL2 specification**

This section describes the mCRL2 specification of the finite set as left-balanced tree. First the sort definition of the left-balanced tree will be described and after that the two most important functions of this tree, i.e. the insert and delete function.

6.2.1. **Sort definition**

The sort definition for the left-balanced tree is the following:

1. sort fsetLB = struct empty | node(Pos, fsetLB, fsetLB, Nat);

The sort $fsetLB$ consists of two constructors: either it is empty or it is a node with a key, a left child, a right child and a height. The key is specified as a positive number, the left and the right child are both finite sets and the height is a natural number.

6.2.2. **Tree-specific functionality**

The most important difference between this tree implementation and the AVL tree implementation is the way in which the insert and delete function are built. The insert function now takes care of the ‘rebalancing’ by adding elements from the left only. Therefore, only the insert and delete function are described in detail, for the remaining tree-specific functions see Appendix C.

**Insert**

1. map
2. insert : Pos # fsetLB -> fsetLB;
3. var
4. $p, q$ : Pos;
5. $l, r$ : fsetLB;
6. $h$ : Nat;
7. eqn
8. insert($p$, empty) = node($p$, empty, empty, 1);
9. insert($p$, node($q$, $l$, $r$, $h$)) =
10. if($p = q$),
11. node($q$, $l$, $r$, $h$),
12. %else
13. if($p < q$ && $l$ == empty,
14. node($q$, insert($p$, $l$), $r$, $h + 1$),
15. %else
16. if($p < q$ && ($cardl == maxnodes$) && ($cardr == maxnodes$),
17. node($q$, insert($p$, $l$), $r$, $h + 1$),
18. %else
19. if($p < q$ && ($cardl != maxnodes$),
20. node($q$, insert($p$, $l$), $r$, $h$),
With this insert function, we want to achieve that each finite set yields a unique tree, i.e. for every finite set there is exactly one corresponding balanced binary search tree. To achieve this, a value will only be inserted into the right sub tree when the left sub tree is completely filled.

A sub tree is completely filled when its cardinality (number of elements that are present in the sub tree) is equal to the maximum number of elements that can be present in a sub tree. The maximum number of elements that can be present in a sub tree is defined as $2^h - 1$, where $h$ is the height of the sub tree. E.g. take a sub tree with height 2, this means that there could be $2^2 - 1 = 3$ elements present at most. When the actual number of elements present is equal to 3, then the sub tree is completely filled, otherwise it isn’t.

The performance of this function is in the worst case of $O(2^n)$. This is because it takes $O(n)$ time to determine whether an if-clause holds, and this has to be done for each if-
clause. Since all these if-statements add together to the total performance, it comes to the order of $O(2^n)$ for an insert of an element into a tree with $n$ elements since it could be the fact that during the insert, a delete is done on the tree such as happens in line 26. In this way it is possible in the worst case that the performance becomes exponential.

I created a case distinction within the insert function based on the completeness of the left and the right sub tree and based on where the value should be inserted, i.e. into the left or into the right sub tree.

For convenience’s sake I will first clarify the where clause that is used in this function, because the equations from this clause occur throughout the specification of insert. As can be seen in the code example above, the where clause starts in the line with number 44. It contains the following five equations:

1. $\text{card}_l = \text{card}(l)$
   
   This line calculates the cardinality of the left sub tree $l$ and stores it in $\text{card}_l$. The cardinality is the number of elements present in $l$.

2. $\text{card}_r = \text{card}(r)$
   
   The second line calculates the cardinality of the right sub tree $r$ and stores it in $\text{card}_r$.

3. $\text{maxnodes} = \exp(2, \text{get_height}(l)) - 1$
   
   $\text{maxnodes}$ represents the maximum number of elements that can be present in a sub tree. Because the number of elements in the right sub tree can not exceed the number of elements in the left sub tree, the $\text{maxnodes}$ variable is calculated by taking $2^{\text{get_height}(l)} - 1$. E.g. for a node of height 3, the maximum number of elements in that tree is 7.

4. $\text{max}_l = \text{maximum}(l)$
   
   $\text{max}_l$ represents the element with the greatest value in the left sub tree $l$.

5. $\text{min}_r = \text{minimum}(r)$
   
   $\text{min}_r$ specifies the smallest element preset in the right sub tree $r$.

Now I will continue with the actual insert function. The equation in line 8 says that inserting a positive number $p$ into an empty tree yields a node with key $p$, an empty left sub tree, an empty right sub tree and a height of value 1.

The equation in line 9 specifies what happens when a value is inserted into a non-empty tree $\text{node}(q, l, r, h)$. This equation is built from a case distinction on whether the sub trees of this node are empty, completely filled or not completely filled. The first if-
statement in line 10 says that when \( p \) is equal to \( q \), i.e. \( p \) already exists in the tree, \( \text{node}(q, l, r, h) \) is returned as result. When \( p \) is smaller than \( q \) then the following cases can be distinguished:

1. Lines 13 and 14:

   ```
   if(p < q && l == empty,
       node(q, insert(p, l), r, h + 1)
   ```

   This line deals with the case that \( p \) is smaller than \( q \) and the left sub tree is empty. Then \( p \) can be inserted into the left sub tree, the right sub tree \( r \) remains the same and the height increases by 1. The visual example of this case can be found in Figure 6.2.

   ![Figure 6.2 Insert case 1](image)

2. Lines 16 and 17:

   ```
   if((p < q) && (cardl == maxnodes) && (cardr == maxnodes),
       node(q, insert(p, l), r, h + 1)
   ```

   The second case is the case where both the left sub tree and the right sub tree are completely filled to their maximum. When this happens, the element \( p \) can be inserted into the left sub tree by a recursive call of insert and the height of the tree will be increased by 1. See Figure 6.3 for a visual example of this case.

   ![Figure 6.3 Insert case 2](image)
3. Lines 19 and 20:

\[
\text{if}((p < q) \land (\text{cardl} \neq \text{maxnodes}), \\
\quad \text{node}(q, \text{insert}(p, l), r, h))
\]

In this case, there is room in the left sub tree for adding the element \( p \) because sub tree \( l \) has not reached the maximum number of elements that can be added. So a recursive call of insert will be done on the left sub tree. See also Figure 6.4.

![Figure 6.4 Insert case 3](image)

4. Lines 22-26:

\[
\text{if}((p < q) \land (\text{cardl} = \text{maxnodes}) \land (\text{cardr} \neq \text{maxnodes}), \\
\quad \text{if}(\text{element_in}(p, l), \\
\quad \quad \text{node}(q, l, r, h), \\
\quad \quad \%\text{else} \\
\quad \quad \quad \text{node}(\text{maxl}, \text{insert}(p, \text{delete}(\text{maxl}, l)), \text{insert}(q, r), h)),
\]

This line of code deals with the possibility that an element has to be inserted into the left sub tree, since \( p < q \), but the sub tree is \( l \) is completely filled. Suppose sub tree \( l \) has a height of 2, then the number of elements in the tree can be 3 at most, before adding a new level to the tree.

In this case, \( l \) is completely filled but an element has to be inserted. What is done is the following: first a check is done whether the element is already present in the left sub tree. When this is the case, no value from \( l \) has to be moved to \( r \) to make space for the new element.

When the value \( p \) isn’t present in \( l \) then the tree is rebuilt with as key the biggest value present in sub tree \( l \) (defined as \( \text{maxl} \)). This maximum value will be removed from sub tree \( l \) by a call of the delete function so there will be room for the new value \( p \). This value \( p \) will then be inserted into sub tree \( l \). The original root of the tree, \( q \), will be inserted into the right sub tree. Because \( l \) is completely filled and \( r \) isn’t, the height of the tree remains the same. An example of a tree in which line 22 of the code example holds can be found in Figure 6.5.
The insert function now continues in line 28 with the cases where $p > q$. For $p > q$, also four cases are distinguished:

5. Lines 28 and 29:

\[
\text{if}(p > q \land l == \text{empty}, \\
(p, \text{node}(q, \text{empty}, \text{empty}, l), r, h + 1)
\]

Here we see the case that a value $p$ has to be inserted that is bigger than the root value $q$ and that the left sub tree of $q$ is empty. What is done is the following: since $q$ is a tree consisting of one node with two empty sub trees, the positions of $q$ and the inserted $p$ are swapped. This means that $p$ becomes the new root of the tree, since $p$ is bigger than $q$, and the node with key $q$ becomes the key of the left sub tree of the tree and the height of the tree increases by 1. The tree in which line 28 of the insert function holds can be found in Figure 6.2.

6. Lines 31 and 32:

\[
\text{if}((p > q) \land \text{cardl} == \text{maxnodes}) \land \text{cardr} != \text{maxnodes}, \\
\text{node}(q, l, \text{insert}(p, r), h)
\]

This part of the nested if-statement deals with the case that $p > q$ and the left sub tree is completely filled and the right sub tree is not completely filled. In this case the value $p$ is just inserted into the right sub tree $r$ with a recursive call of the insert function. The tree in which line 31 holds is visualized in Figure 6.5.

7. Lines 34-38:

\[
\text{if}((p > q) \land \text{cardl} != \text{maxnodes}) \land \text{r} != \text{empty}, \\
\text{if}(\text{element\_in}(p, r), \\
\text{node}(q, l, r, h), \\
\text{%else} \\
\text{node}(\text{minr}, \text{insert}(q, l), \text{insert}(p, \text{delete}(\text{minr}, r)), h))
\]

---

Figure 6.5 Insert case 4
In this case, a value $p$ is inserted that is bigger than $q$, so $p$ should be inserted into the right sub tree. This is only possible when there is no room left in the left sub tree. This means that a value from $r$ should be moved to sub tree $l$ to completely fill that tree. First a check is done whether the element is not already present in $r$, because when it is, no element has to be moved from $r$ to $l$.

When the element is not present in $r$, then the minimum of $r$ becomes the new root, but only if sub tree $r$ is not empty. Then $q$ is inserted into the left sub tree, and $p$ is inserted into the right sub tree. See Figure 6.4 for an example of a tree in which lines 34-38 are applied.

8. Lines 40-43:

```
if(element_in(p, r),
  node(q, l, r, h),
%else
  node(minr, insert(q, l), insert(p, delete(minr, r)), h + 1)
```

This case does almost the same as case 7, but here both sub trees $l$ and $r$ are completely filled. This means that when trying to insert a value $p > q$ into sub tree $r$, first an element from $r$ has to be moved to $l$ to create space. So if the value $p$ doesn’t exist yet in sub tree $r$, the smallest value of $r$ becomes the new root, the original root is inserted into $l$ and the value $p$ is inserted into $r$ and the height is increased by 1.

Delete

```
1. map
2.   delete : Pos # fsetLB -> fsetLB;
3. var
4.   p, q : Pos;
5.   l, r : fsetLB;
6.   h : Nat;
7. eqn
8.   delete(p, empty) = empty;
9.   delete(p, node(q, l, r, h)) =
10.  if((p < q) && (cardl == maxnodes) && (cardr == maxnodes) &&
11.      (r != empty),
12.      if(element_in(p, l),
13.          node(minr, insert(q, delete(p, l)), delete(minr, r), h),
14.          %else
15.          node(q, l, r, h)),
16.      %else
17.      if((p < q) && (cardl != maxnodes),
18.          node(q, delete(p, l), r, max(get_height(delete(p, l)),
19.            get_height(r)) + 1),
20.      %else
21.      if((p < q) && (cardl == maxnodes) && (cardr != maxnodes) &&
22.        (r != empty),
23.        if(element_in(p, l),
24.          node(minr, insert(q, delete(p, l)),
25.          %else
26.          node(q, l, r, h))
```
With the delete function, it has to be possible to delete an element from the tree in such a way that the tree stays in its unique form. This means that it could be possible that the tree has to be reshaped after a delete. I created a case distinction within the delete
function based on the completeness of the left and the right sub tree and the fact from which sub tree the value should be deleted. The delete function uses exactly the same where clause as the insert function does, so a detailed description can be found in section 6.2.2, where the insert function is described.

The performance of this function is in the worst case of $O(2^n)$ since just as in the insert function. Each of the if-clauses is of $O(n)$ to determine whether it holds or not and it is possible that a call of insert is done during a delete. Therefore the worst performance of delete is of the order of $O(2^n)$ for a delete of an element from a tree with n elements.

The first equation of delete in line 8 specifies what happens when one tries to delete a value $p$ from an empty tree: this yields an empty tree again. The second equation, starting in line 9, specifies what happens when a value is deleted from a non-empty tree. This equation is built from a case distinction on whether the sub trees are empty, completely filled or not completely filled. For the case that $p$ is smaller than $q$, the following four sub cases can be distinguished:

1. Lines 10-14:

   ```
   if((p < q) && (cardl == maxnodes) && (cardr == maxnodes) &&
   (r != empty),
   if(element_in(p, l),
       node(minr, insert(q, delete(p, l)), delete(minr, r), h),
   %else
       node(q, l, r, h)),
   ```

   In this case, $p$ is smaller than $q$, $l$ is completely filled and $r$ is completely filled and $r$ is not empty. The clause $r$ is not empty is added because the clause cardr == maxnodes is also true when $r$ is empty, because then cardr == 0 and maxnodes == 0.

   When this clause holds, it is checked whether the element actually exists in the left sub tree. When this is the case the minimum of $r$ becomes the new root of the tree, the original root node $q$ is moved to the left sub tree and the minimum of $r$ is removed from $r$. When the value to be deleted doesn’t exist in $l$, node($q$, $l$, $r$, $h$) is returned as result. Figure 6.6 displays the tree for which case 1 holds.
2. Lines 16 and 17:

   \[
   \text{if}((p < q) \land \land (\text{cardl} \neq \text{maxnodes}), \\
   \quad \text{node}(q, \text{delete}(p, \text{l}), r, \\
   \quad \quad \max(\text{get_height}(\text{delete}(p, \text{l})), \text{get_height}(r)) + 1), \\
   \]

In this case, \( p \) is smaller than \( q \) and sub tree \( l \) is not completely filled. Now just a recursive call of delete is done on \( p \) and the left sub tree and the height of the tree is recalculated. A tree for which case 2 is applicable is shown in Figure 6.7.

\[
\text{Figure 6.7 Delete case 2}
\]

3. Lines 19-23:

   \[
   \text{if}((p < q) \land \land (\text{cardl} == \text{maxnodes}) \land \land (\text{cardr} \neq \text{maxnodes}) \land \land (r \neq \text{empty}), \\
   \quad \text{if(element_in}(p, \text{l}), \\
   \quad \quad \text{node}(\text{minr}, \text{insert}(q, \text{delete}(p, \text{l})), \text{delete}(\text{minr}, \text{r}), h), \\
   \quad \quad \%else \\
   \quad \quad \quad \text{node}(q, \text{l}, \text{r}, h)), \\
   \]

Here, \( p \) is smaller than \( q \) and \( l \) is completely filled, \( r \) is not completely filled but also not empty. If the element to be deleted is present in the left sub tree, then the minimum of \( r \) becomes the new root of the tree and this minimum is deleted from
the right sub tree. The original root $q$ will be moved to sub tree $l$. If the element to be deleted is not present in $l$ then nothing happens and the tree remains the same. See Figure 6.8 for a tree that corresponds to this case.

Figure 6.8 Delete case 3

4. Lines 25 and 26:

\[
\text{if}((p < q) \land (r == \text{empty}), \\
\quad \text{node}(q, \text{delete}(p, l), r, \text{Int2Nat}(h - 1)),
\]

Here $p$ is smaller than $q$ and $r$ is empty. Now this means that $l$ consists of only one element, which can be deleted and the height of the tree is decreased by 1.

Figure 6.9 Delete case 4
These were the cases where \( p \) was smaller than \( q \). I will now continue with describing the cases where \( p \) is bigger than \( q \). The following sub cases can be distinguished:

5. Lines 28 and 29:

\[
\text{if}( (p > q) \land \text{card}_l == \text{maxnodes} ) \land \text{card}_r == \text{maxnodes}, \\
\text{node}(q, l, \text{delete}(p, r), h),
\]

Here \( p \) is greater than \( q \), \( l \) is completely filled and \( r \) is completely filled. In this sub case, just a recursive call of delete is done to remove the element from \( r \). The visual example for this case can be found in Figure 6.6, but it can also look like Figure 6.10.

\[ \text{Figure 6.10 Delete case 5} \]

6. Lines 31-35:

\[
\text{if}( (p > q) \land \text{card}_l == \text{maxnodes} ) \land \text{card}_r != \text{maxnodes} \land (l != \text{empty}), \\
\text{if}( \text{element}_\text{in}(p, r), \\
\text{node}(\text{max}_l, \text{delete}(\text{max}_l, l), \text{insert}(q, \text{delete}(p, r)), h), \\
\text{%else} \\
\text{node}(q, l, r, h)),
\]

\( p \) is greater than \( q \), \( l \) is completely filled, \( r \) is completely filled and \( l \) is not empty \((l \) is not empty is added because the clause \( \text{card}_l == \text{maxnodes} \) is also true when \( l \) is empty, because then \( \text{card}_l == 0 \) and \( \text{maxnodes} == 0 \)).

If the element to be deleted is present in \( r \), then the greatest value in \( l \) (this value is present because of the clause \( l != \text{empty} \)) becomes the new root of the tree. This element is removed from \( l \), the original root \( q \) is moved to \( r \) and the element \( p \) is deleted from \( r \). An example of a tree for which case 6 holds can be found in Figure 6.8.

7. Lines 37-41:

\[
\text{if}( (p > q) \land \text{card}_l != \text{maxnodes} ) \land (l != \text{empty}), \\
\text{if}( \text{element}_\text{in}(p, r),
\]

- 66 -
When $p$ is greater than $q$ and $l$ is not completely filled but also not empty, a check is done whether the element $p$ occurs in $r$. When this is the case, the greatest value in $l$ becomes the new root, this value is deleted from $l$, the original root $q$ is moved to $r$ and $p$ is deleted from $r$. In this case, also the height has to be recalculated. This is necessary because it could be possible that the height decreases with 1. See Figure 6.7 for an example for a tree for which case 7 holds.

The last case in the delete function is that when $p$ is equal to $q$, i.e. the element to be deleted is found in the tree. For this case, four sub cases are present:

8. Lines 44 and 45:

   if($(l == empty) \&\& (r == empty)$,
   empty,

   Since both sub trees of the element to be deleted are empty, the element is deleted and replaced by an empty sub tree.

9. Lines 47 and 48:

   if($(l == empty) \&\& (r != empty)$,
   $r$,

   Here $l$ is empty and $r$ is not, this means that the value to be deleted is replaced by its right sub tree.

10. Lines 50 and 51:

    if($(l != empty) \&\& (r == empty)$,
    $l$,

    When $l$ is not empty and $r$ is empty, the deleted value is replaced by its left sub tree.

11. Line 55:

    node(minr, l, delete(minr, r), h)))

    When both sub trees are not empty, the deleted value is replaced by a tree that has the minimum value of $r$ as its root, a left sub tree $l$ (the original left sub tree of the deleted value), a right sub tree $r$ without the minimum of $r$, and a height $h$. 

    

    node(maxl, delete(maxl, l), insert(q, delete(p, r)),
    max(get_height(delete(maxl, l)),
    get_height(insert(q, delete(p, r)))) + 1),

    %else
    node(q, l, r, h)),
7. Finite set as unique tree

7.1. Unique tree using list conversion

For the fourth and last option to model a finite set in mCRL2 I chose to model a finite set as a unique tree that uses conversion to an ordered List. This means that upon inserting an element into a tree or upon deleting an element from a tree, this tree is first converted to an ordered List. Then the specified element is inserted into or deleted from the ordered List and the resulting List is converted back to a tree again. The tree is kept unique for each finite set by the way in which the ordered List is converted back into a tree; this conversion is described in detail in section 7.2.2. In the remainder of this chapter, the most important parts of the mCRL2 specification of the finite set as unique tree are described.

7.2. mCRL2 specification

In this section, the most important functionality of the mCRL2 specification is treated. The sort definition and the most important functionality are discussed. I chose not to describe all set-specific functionality in detail because that functionality is very similar to the functionality used in the AVL tree implementation. The complete specification of the finite set as unique tree can be found in Appendix D.

7.2.1. Sort definition

For the unique tree that uses list conversion, two sort definitions are created, one for a finite set as an ordered List as already used in the ordered List implementation and one for a finite set as a unique tree:

1. sort fsetUT = struct empty | node(Pos, fsetUT, fsetUT, Nat);
2. sort fsetList = List(Pos);

Line number 1 defines the sort fsetUT, which is a structured type containing two constructors: either it is empty or it is a node with a positive number as its root, a left subchild of the type fsetUT, a right subchild of the type fsetUT and a natural number representing the cardinality of the tree. Line number 2 defines that the sort fsetList is a List of positive numbers (this is in fact the same as the ordered List).
7.2.2. Tree specific functionality

The most important difference between the unique tree that uses List conversion and the tree that is left-balanced is the fact that before inserting an element into the tree, this tree is first converted to an ordered List. Then the element to be added to the tree is inserted into the ordered List and that result is converted back to a tree. In this section the functions \( fsetList2fsetTree \), \( preorderleft \), \( preorderright \), \( fsetTree2fsetList \), \( insert \) and \( delete \) are treated.

Finite set as tree to finite set as ordered List

1. map
2. \( fsetUT2fsetList : fsetList \# fsetUT \to fsetList; \)
3. var
4. \( base : fsetList; \)
5. \( q : Pos; \)
6. \( l, r : fsetUT; \)
7. \( c : Nat; \)
8. eqn
9. \( fsetUT2fsetList(base, empty) = base; \)
10. \( fsetUT2fsetList(base, node(q, l, r, c)) = \)
    \( fsetUT2fsetList(q \mid> fsetUT2fsetList(base, r), l); \)

Here we see the first conversion function that can be used to convert a finite set as tree to a finite set as ordered List. This function takes a \( fsetList \) and an \( fsetUT \) as parameters and returns an \( fsetList \). When using this function, the following happens: the \( fsetUT \) elements are added to the List represented by the \( fsetList \) parameter. This parameter is here called \( base \) here. By adding an empty tree to the List \( base \), nothing happens and \( base \) remains the same List. When adding a non-empty tree to \( base \), a recursive call of \( fsetUT2fsetList \) is done with as its parameters the \( fsetList \) represented by \( q \mid> fsetUT2fsetList(base, r) \) and the left sub child \( l \). In this way, the elements of the tree are processed in such a way that the resulting List is an ordered List. More information on tree processing as is done in this function can be found in [1]. The performance of this function is of \( O(n) \) since for each element \( q \) of \( node(q, l, r, c) \) the function \( \mid> \) is called on \( q \) and the remaining part of the node.

Pre-order left

1. map
2. \( preorderleft : fsetList \# Nat \to fsetList; \)
3. var
4. \( n : Nat; \)
5. \( i : fsetList; \)
6. eqn
7. \( n >= 1 \to preorderleft(i, n) = \)
    \( head(i) \mid> preorderleft(tail(i), Int2Nat(n - 1)); \)
The function `preorderleft` creates a sub List of an ordered List that contains the first \( n \) elements of that List. The function takes a parameter of the type `fsetList` and a natural number as input and it returns an ordered List. It contains one equation which says that when a List of at least length 1 is processed, then it makes a recursive call on the `tail` of \( l \) and \( n - 1 \). `Preorderleft` performs in \( O(n) \) time since for each \( n \), the functions `head(i), |>` and `tail(i)` are called upon.

**Pre-order right**

1. map
2. `preorderright : fsetList # Nat -> fsetList;`
3. var
4. `n : Nat;`
5. `i : fsetList;`
6. eqn
7. `n == 0 -> preorderright(i, n) = i;`
8. `n > 0 -> preorderright(i, n) = preorderright(tail(i), Int2Nat(n - 1));`

This function creates a sub List of an ordered List from the end of that List, i.e. it removes the first \( n \) elements of \( i \) and returns the rest. It contains two equations. The first one says that if \( n \) is 0, then the result is just the complete List \( i \). The second equation says that when \( n \) is bigger than 0, then it yields a sub List of \( i \) without the first \( n \) elements. `Preorderright` has a total performance of \( O(n) \).

**Finite set as ordered List to finite set as tree**

1. map
2. `fsetList2fsetUT : fsetList # Nat -> fsetUT;`
3. var
4. `p : Pos;`
5. `l : fsetList;`
6. `n : Nat;`
7. eqn
8. `n < 1 -> fsetList2fsetUT(i, n) = empty;`
9. `n > 0 && n < 2 -> fsetList2fsetUT(i, n) = node(head(i), empty, empty, 1);`
10. `n >= 2 -> fsetList2fsetUT(i, n) = node(i.(middle),`
   `fsetList2fsetUT(preorderleft(i, middle), middle),`
   `fsetList2fsetUT(preorderright(i, middle + 1),
    Int2Nat((n - middle) - 1)),
   n)`
11. whr
12. `middle = n div 2`
13. end;

Another important conversion function is `fsetList2fsetUT` in which an ordered List is converted to a unique tree. It is built in such a way that for each ordered List, there is exactly one tree representation. It uses three different equations for building a tree from a List. I will start with the first equation that starts in line 8 of the code example that can be
seen above. That equation says that when the natural number parameter \( n \) (which specifies the length of the List) is smaller than 1, the result is the empty tree.

The second equation (line 9), creates a tree from a List of length 1. This results in a node with the first element of the List as its key, an empty left sub child, an empty right sub child and a height of value 1.

The third and last equation, line 10, creates a tree from a List with a length of at least 2. The root of the resulting tree is represented by the element at position \( \text{middle} \), which is equal to \( n \div 2 \). The left sub tree is built recursively with the function \( \text{preorderleft} \), which creates a sub List of a number of elements, in this case a sub List of \( \text{middle} \) elements, i.e. \( n \div 2 \) elements. The right sub tree is built recursively with the sub List that is created with \( \text{preorderright} \). The tree that results has a total cardinality of \( n \).

\( \text{fsetList2fsetUT} \) has a total performance of \( O(n \times \log(n)) \) since for each \( n \) bigger than 1 it does a call on \( \text{preorderleft} \) and on \( \text{preorderright} \) which are both of \( O(n) \), and this call is done on \( \log(n) \) levels.

**Insert**

1. map
2. \( \text{insert} : \text{Pos} \rightarrow \text{fsetUT} \rightarrow \text{fsetUT} \);
3. var
4. \( p, q : \text{Pos} \);
5. \( l, r : \text{fsetUT} \);
6. \( c : \text{Nat} \);
7. eqn
8. \( \text{insert}(p, \text{empty}) = \text{node}(p, \text{empty}, \text{empty}, 1) \);
9. \( \text{insert}(p, \text{node}(q, l, r, c)) = \)
10. \( \text{if}(\text{element_in}(p, \text{node}(q, l, r, c)), \)
11. \( \text{node}(q, l, r, c), \)
12. \( \text{else} \rightarrow \)
13. \( \text{fsetList2fsetUT}(\text{insert}(p, \)
14. \( \text{fsetUT2fsetList}([], \)
15. \( \text{node}(q, l, r, c))), \)
16. \( \text{get_card}(\text{node}(q, l, r, c)) + 1)) \);

The insert function takes a positive number and an \( \text{fsetUT} \) as input an returns a new \( \text{fsetUT} \) is output. It consists of two equations. The first equation says that inserting a positive number \( p \) into an empty \( \text{fsetUT} \) yields a node with key \( p \), an empty left sub child, an empty right sub child and a cardinality of value 1. The second equation specifies what happens when an insert of a positive number \( p \) is done on a non-empty finite set. This equation starts in line 13. It first checks whether \( p \) already occurs in \( \text{node}(q, l, r, c) \). When this is the case, \( p \) already exists in the tree and \( \text{node}(q, l, r, c) \) is returned as result. When \( p \) is not present in \( \text{node}(q, l, r, c) \) it does the following: it calls the function \( \text{insert} \) (this is the \( \text{insert} \) belonging to \( \text{fsetList} \)) on \( p \) and the List representation of \( \text{node}(q, l, r, c) \). This List representation is obtained by calling \( \text{fsetUT2fsetList} \) on the empty List and on \( \text{node}(q, l, r, c) \). The result from the call of
the insert function is converted back to a tree by using \text{fsetList2fsetUT} with as parameters the List returned by \text{add} and the length of that List. Insert can be done in $O(n)$ time because for each element to be inserted, one call is done on \text{fsetList2fsetUT} ($O(n \times \log(n))$) and on \text{fsetUT2fsetList} (also $O(n)$), one call on \text{element_in} ($O(\log(n))$) and one on \text{get_card} ($O(1)$). This yields a total performance of $O(n \times \log(n))$.

Delete

1. map
2. \text{delete} : Pos \# fsetUT -> fsetUT;
3. var
4. \quad p, q : Pos;
5. \quad l, r : fsetUT;
6. \quad c : Nat;
7. eqn
8. \quad \text{delete}(p, \text{empty}) = \text{empty};
9. \quad \text{delete}(p, \text{node}(q, l, r, c)) =
10. \quad \text{if} (\text{element_in}(p, \text{node}(q, l, r, c)),
11. \quad \quad \text{fsetList2fsetUT} (\text{remove}(p, \text{fsetTree2fsetList}([],
12. \quad \quad \quad \text{node}(q, l, r, c))),
13. \quad \quad \text{Int2Nat} (\text{get_card}(\text{node}(q, l, r, c)) - 1)))
14. \quad \text{else}
15. \quad \text{node}(q, l, r, c));

Delete works in a similar way as insert. It doesn’t directly delete an element from the tree but it removes an element from the List representation of the tree. So when trying to delete a positive number from an fsetUT, two things are possible: either the fsetUT is empty, causing the result to yield empty, or fsetUT is not empty. Then, the following happens: if the element to be deleted is present in the tree then the tree is converted to an ordered List by using fsetUT2fsetList. The function delete removes \text{p} from the List representation of the tree. When delete returns, the resulting List is a parameter of the \text{fsetList2fsetUT} function that converts the List back to a tree. The second parameter is the length of the List that was returned by the delete function. The performance of the delete function can be calculated in the same way as the performance of the insert function and thus yields a total performance of $O(n \times \log(n))$. 
8. Proofs

8.1. Finite set as ordered List

In this section I will give proofs for a number of properties that should hold in the ordered List implementation of a finite set. This is done to provide a more theoretical foundation for the implementation. There are four theorems that I want to prove for the finite set as ordered List specification:

**Theorem 1:** \( \forall p: \text{Pos} \forall L: \text{fset} : (\text{element\_in}(p, \text{insert}(p, L))) \)

This theorem says that for every positive number and for every ordered List it should hold that whenever an element is inserted into an ordered List, it is present in the ordered List, i.e. when you search for an element after an insert, you should find it.

Proof: By induction on the structure of \( L \):

Base case: \( L == [] \)

\[
\text{element\_in}(p, \text{insert}(p, L)) \equiv \{ L == [] \}
\]

\[
\text{element\_in}(p, \text{insert}(p, [])) \equiv \{ \text{definition of insert} \}
\]

\[
\text{element\_in}(p, [p]) \equiv \{ \text{definition of element\_in} \}
\]

\( True \)

Induction step: \( L == [q \triangleright L'] \)

Induction Hypothesis: \( \text{element\_in}(p, \text{insert}(p, L')) \)

\[
\begin{align*}
\text{element\_in}(p, \text{insert}(p, L)) & \equiv \{ L == [q \triangleright L'] \} \\
\text{element\_in}(p, \text{insert}(p, [q \triangleright L'])) & \equiv \{ \text{case distinction on p, definition of insert} \} \\
& \quad 1: p < q : \text{element\_in}(p, [p \triangleright q \triangleright L']) \\
& \quad 2: p > q : \text{element\_in}(p, [q \triangleright \text{insert}(p, L')]) \\
& \quad 3: p == q : \text{element\_in}(p, [p \triangleright L'])
\end{align*}
\]
1: \( p < q \) : element _in\((p, [p \triangleright q \triangleright L'])\) 
\( \equiv \) \{definition of element _in\} 
True

2: \( p > q \) : element _in\((p, [q \triangleright insert(p, L'])])\) 
\( \equiv \) \{definition of element _in\} 
\( p == q \lor \) element _in\(\(p, insert(p, L')\)) \( \equiv \) \{Induction Hypothesis\} 
True

3: \( p == q \) : element _in\((p, [p \triangleright L'])\) 
\( \equiv \) \{definition of element _in\} 
True

\[\Box\]

**Theorem 2:** \( \forall p : Pos \forall L : fset : (ordered(L) \rightarrow \neg(element \_in(p, delete(p, L)))\)

Whenever an ordered List is ordered, it should hold that whenever an element is deleted, it cannot be found in the List anymore.

Proof: By induction on the structure of \( L \):

Base case: \( L == [] \)
Assumption 1: \( ordered(L) \)
Prove: \( \neg(element \_in(p, delete(p, L)))\)

\( \neg(element \_in(p, delete(p, L)))\) 
\( \equiv \{L == []\} \) 
\( \neg(element \_in(p, delete(p, [])))\) 
\( \equiv \{definition of delete\} \) 
\( \neg(element \_in(p, []))\) 
\( \equiv \{definition element \_in\} \) 
True

Induction step: \( L == [q \triangleright L'] \)
Induction Hypothesis: \( ordered(L') \rightarrow \neg(element \_in(p, delete(p, L')))\)
Assumption 1: \( ordered(L) \)
Prove: \( \neg(element \_in(p, delete(p, L)))\)
\neg (\text{element \_in}(p, \text{delete}(p, L)))
\equiv \{ L \triangleright= [q \triangleright L'] \}
\neg (\text{element \_in}(p, \text{delete}(p, [q \triangleright L'])))
\equiv \{ \text{case distinction on } p, \text{definition of } \text{delete} \}
1: p < q : \neg (\text{element \_in}(p, [q \triangleright L']))
2: p > q : \neg (\text{element \_in}(p, [q \triangleright \text{delete}(p, L')])))
3: p \triangleright= q : \neg (\text{element \_in}(p, [L'])))

1: p < q : \neg (\text{element \_in}(p, [q \triangleright L']))
\equiv \{ p < q, \text{definition of } \text{element \_in} \}
\text{True}

2: p > q : \neg (\text{element \_in}(p, [q \triangleright \text{delete}(p, L')])))
\equiv \{ p > q, \text{definition of } \text{element \_in} \}
\neg (\text{element \_in}(p, \text{delete}(p, L'))))
\equiv \{ \text{Induction Hypothesis} \}
\text{True}

3: p \triangleright= q : \neg (\text{element \_in}(p, [L'])))
\equiv \{ L \triangleright= [q \triangleright L'] \}
\text{True}

\hfill \Box

**Theorem 3:** \( \forall p : \text{Pos} \forall L : \text{fset} : (\text{ordered}(L) \rightarrow \text{ordered}(\text{insert}(p, L)))\)

The third theorem says that whenever an ordered List is ordered, it should still be ordered after the insertion of an element.

Proof: By induction on the structure of \( L \):

**Base case:** \( L \triangleright= [] \)

**Assumption 1:** \( \text{ordered}(L) \)

**Prove:** \( \text{ordered}(\text{insert}(p, L)) \)
\text{ordered}(\text{insert}(p, L))
\equiv \{ L == [] \}
\text{ordered}(\text{insert}(p,[]))
\equiv \{ \text{definition of insert} \}
\text{ordered}([p])
\equiv \{ \text{definition of ordered} \}
\text{True}

\text{Induction step: } L == [q |> L']
\text{Induction Hypothesis: } \text{ordered}(L) \rightarrow \text{ordered}(\text{insert}(p, L'))
\text{Assumption 1: } \text{ordered}(L)
\text{Prove: } \text{ordered}(\text{insert}(p, L))

\text{ordered}(\text{insert}(p, L))
\equiv \{ L == [q |> L'] \}
\text{ordered}(\text{insert}(p,[q |> L']))
\equiv \{ \text{case distinction on } p, \text{definition of insert} \}
1: p < q : \text{ordered}([p |> q |> L'])
2: p > q : \text{ordered}([q |> \text{insert}(p, L')])
3: p == q : \text{ordered}([p |> L'])

1: p < q : \text{ordered}([p |> q |> L'])
\equiv \{ \text{definition of ordered} \}
p < q \land \text{ordered}([q |> L'])
\equiv \{ p < q \}
\text{ordered}([q |> L'])
\equiv \{ \text{rewrite assumption 1 to ordered } ([q |> L']) \}
\text{True}

2: p > q : \text{ordered}([q |> \text{insert}(p, L')])
\equiv \{ \text{definition of ordered} \}
q < p \land \text{ordered}(\text{insert}(p, L'))
\equiv \{ q < p \}
\text{ordered}(\text{insert}(p, L'))
\equiv \{ \text{Induction Hypothesis} \}
\text{True}
3: \( p == q : \text{ordered}([p \triangleright L']) \)
\[\equiv \{ p == q \}
\text{ordered}([q \triangleright L'])\]
\[\equiv \{ \text{rewrite assumption 1 to ordered}([q \triangleright L']) \}\]

True

\[\Box\]

**Theorem 4:** \( \forall p: \text{Pos} \forall L: \text{fset}: (\text{ordered}(L) \rightarrow \text{ordered} (\text{delete}(p, L))) \)

The fourth and last theorem says that whenever an ordered List is ordered, it should still be ordered after deleting an element.

Proof by induction on the structure of \( L \):

Base case: \( L == [] \)
Assumption 1: \( \text{ordered}(L) \)
Prove: \( \text{ordered}(\text{delete}(p, L)) \)

\( \text{ordered}(\text{delete}(p, L)) \)
\[\equiv \{ L == [] \}\]
\( \text{ordered}(\text{delete}(p, []))\)
\[\equiv \{ \text{definition of delete} \}\]
\( \text{ordered}([])\)
\[\equiv \{ \text{definition of ordered} \}\]

True

Induction step: \( L == [q \triangleright L'] \)
Induction Hypothesis: \( \text{ordered}(L) \rightarrow \text{ordered}(\text{delete}(p, L')) \)
Assumption 1: \( \text{ordered}(L) \)
Prove: \( \text{ordered}(\text{delete}(p, L)) \)

\( \text{ordered}(\text{delete}(p, L)) \)
\[\equiv \{ L == [q \triangleright L'] \}\]
\( \text{ordered}(\text{delete}(p, [q \triangleright L']))\)
\[\equiv \{ \text{case distinction on p, definition of delete} \}\]
1: \( p < q : \text{ordered}([q \triangleright L']) \)
2: \( p > q : \text{ordered}([q \triangleright \text{delete}(p, L']) \)
3: \( p == q : \text{ordered}([L']) \)
1: \( p < q : \text{ordered}([q \triangleright L']) \)
\[ \equiv \{ \text{rewrite assumption 1 to ordered}([q \triangleright L']) \} \]
\( \text{True} \)

2: \( p > q : \text{ordered}([q \triangleright \text{delete}(p,L']) \)
\[ \equiv \{ \text{definition of ordered} \} \]
\( q < p \land \text{ordered} (\text{delete}(p,L')) \)
\[ \equiv \{ q < x \} \]
\( \text{ordered (delete(p,L'))} \)
\[ \equiv \{ \text{Induction Hypothesis} \} \]
\( \text{True} \)

3: \( p =\equiv q : \text{ordered}([L']) \)
\[ \equiv \{ \text{ordered } ([L]) \rightarrow \text{ordered } ([L']) \} \]
\( \text{True} \)

\[ \square \]

### 8.2. Finite set as AVL tree

To provide a more theoretical foundation for the implementation of finite sets as AVL trees, I will give proofs for a number of properties that should always hold in that implementation. There are 25 lemmas, numbered I to XXV, and 9 theorems, numbered 1 to 9, that I want to prove for this implementation. For readability purposes, the function `get_height()` will be abbreviated to `gh()`.

**Lemma I:** \( \forall p: \text{Pos} \forall T: \text{fset} : (\text{element \_ in}(p,T) \rightarrow \text{element \_ in}(p,\text{singleleft}(T))) \)

In this lemma it will be proven that whenever an element \( p \) is present in a tree \( T \), then that element \( p \) is also present when a single left rotation is applied to \( T \).

**Proof:** By case distinction on the structure of \( T \).

Assumption 1: \( \text{element \_ in}(p,T) \)

Prove: \( \text{element \_ in}(p,\text{singleleft}(T)) \)
\begin{verbatim}

element_in(p, singleleft(T))
≡ \{ T == node(q, l, r, h) \}

\begin{align*}
\text{element_in}(p, \text{singleleft}(\text{node}(q, l, r, h))) \\
& \equiv \{ \text{definition of singleleft} \} \\
\text{element_in}(p, \text{node}(\text{key}(r), \text{node}(q, l, \text{left}(r), H), \text{right}(r), H)) \\
& \equiv \{ \text{case distinction on } p, \text{definition of element_in} \} \\
1: & p < \text{key}(r) : \text{element_in}(p, \text{node}(q, l, \text{left}(r), H)) \\
2: & p > \text{key}(r) : \text{element_in}(p, \text{right}(r)) \\
3: & p == \text{key}(r) : \text{element_in}(p, \text{node}(p, \text{node}(q, l, \text{left}(r), H), \text{right}(r), H))
\end{align*}

1: p < \text{key}(r) : \text{element_in}(p, \text{node}(q, l, \text{left}(r), H))
≡ \{ \text{case distinction on } p, \text{definition of element_in} \} \\
\text{1a} : p < q : \text{element_in}(p, l) \\
\text{1b} : p > q : \text{element_in}(p, \text{left}(r)) \\
\text{1c} : p == q : \text{element_in}(p, \text{node}(p, l, \text{left}(r), H))

\text{1a} : p < q : \text{element_in}(p, l)
≡ \{ p < q, \text{rewrite assumption 1 to element_in}(p, l) \} \\
\text{True}

\text{1b} : p > q : \text{element_in}(p, \text{left}(r))
≡ \{ p < \text{key}(r) \land p > q, \text{definition of element_in} \} \\
\text{True}

\text{1c} : p == q : \text{element_in}(p, \text{node}(p, l, \text{left}(r), H))
≡ \{ \text{definition of element_in} \} \\
\text{True}

2: p > \text{key}(r) : \text{element_in}(p, \text{right}(r))
≡ \{ p > \text{key}(r), \text{definition of element_in} \} \\
\text{True}

3: p == \text{key}(r) : \text{element_in}(p, \text{node}(p, \text{node}(q, l, \text{left}(r), H), \text{right}(r), H))
≡ \{ \text{definition of element_in} \} \\
\text{True}

\square
\end{verbatim}
**Lemma II:** \( \forall p : \text{Pos}\ \forall T : \text{fset} : (\text{element\_in}(p,T) \rightarrow \text{element\_in}(p,\text{singleright}(T))) \)

The second lemma proves that whenever an element \( p \) is present in a tree \( T \), then \( p \) will also be present after applying a single right rotation on \( T \).

**Proof:** By case distinction on the structure of \( T \).

Assumption 1: \( \text{element\_in}(p,T) \)
Prove: \( \text{element\_in}(p,\text{singleright}(T)) \)

\[
\text{element\_in}(p,\text{singleright}(T))
\equiv \{ T == \text{node}(q,l,r,h) \}
\text{element\_in}(p,\text{singleright}(\text{node}(q,l,r,h)))
\equiv \{ \text{definition of singleright} \}
\text{element\_in}(p,\text{node}(\text{key}(l),\text{left}(l),\text{node}(q,\text{right}(l),r,H'),H))
\equiv \{ \text{case distinction on } p, \text{definition of } \text{element\_in} \}
1: p < \text{key}(l) : \text{element\_in}(p,\text{left}(l))
2: p > \text{key}(l) : \text{element\_in}(p,\text{node}(q,\text{right}(l),r,H'))
3: p == \text{key}(l) : \text{element\_in}(p,\text{node}(p,\text{left}(l),\text{node}(q,\text{right}(l),r,H'),H))
\]

1: \( p < \text{key}(l) : \text{element\_in}(p,\text{left}(l)) \)
\equiv \{ p < \text{key}(l), \text{definition of } \text{element\_in} \}
True

2: \( p > \text{key}(l) : \text{element\_in}(p,\text{node}(q,\text{right}(r),r,H')) \)
\equiv \{ \text{case distinction on } p, \text{definition of } \text{element\_in} \}
2a : p < q : \text{element\_in}(p,\text{right}(l))
2b : p > q : \text{element\_in}(p,r)
2c : p == q : \text{element\_in}(p,\text{node}(p,\text{right}(r),r,H'))

2a : p < q : \text{element\_in}(p,\text{right}(l))
\equiv \{ p > \text{key}(l) \land p < q, \text{definition of } \text{element\_in} \}
True

2b : p > q : \text{element\_in}(p,r)
\equiv \{ p > q, \text{rewrite assumption 1 to } \text{element\_in}(p,r) \}
True
2c : \( p \equiv q \) : element\_in\( (p, \text{node}(p, \text{right}(r), r, H')) \)
\[ \equiv \{ \text{definition of element\_in} \} \]
\( True \)

3 : \( p \equiv \text{key}(l) : \text{element\_in}(p, \text{node}(p, \text{left}(l), \text{node}(q, \text{right}(l), r, H'), H)) \)
\[ \equiv \{ \text{definition of element\_in} \} \]
\( True \)

\[ \square \]

**Lemma III:** \( \forall p : \text{Pos} \ \forall T : fset : (\text{element\_in}(p, T) \rightarrow \text{element\_in}(p, \text{doubleleft}(T))) \)

Lemma III gives a proof of the fact that when an element \( p \) is present in a tree \( T \), it will also be present after an application of a double left rotation on \( T \).

**Proof:** By case distinction on the structure of \( T \).

Assumption 1: \( \text{element\_in}(p, T) \)
Prove: \( \text{element\_in}(p, \text{doubleleft}(T)) \)

\( \text{element\_in}(p, \text{doubleleft}(T)) \)
\[ \equiv \{ T \equiv \text{node}(q, l, r, h) \} \]
\( \text{element\_in}(p, \text{doubleleft}(\text{node}(q, l, r, h))) \)
\[ \equiv \{ \text{definition of doubleleft} \} \]
\( \text{element\_in}(p, \text{singleleft}(\text{node}(q, l, \text{singleright}(r), h))) \)
\[ \equiv \{ \text{Lemma I} \} \]
\( \text{element\_in}(p, \text{node}(q, l, \text{singleright}(r), h)) \)
\[ \equiv \{ \text{case distinction on } p, \text{definition of element\_in} \} \]

1 : \( p < q : \text{element\_in}(p, l) \)
2 : \( p > q : \text{element\_in}(p, \text{singleright}(r)) \)
3 : \( p \equiv q : \text{element\_in}(p, \text{node}(p, l, \text{singleright}(r), h)) \)

1 : \( p < q : \text{element\_in}(p, l) \)
\[ \equiv \{ p < q, \text{rewrite assumption 1 to element\_in}(p, l) \} \]
\( True \)
2: \( p > q : \text{element \_in}(p, \text{singleright}(r)) \)
\[ \equiv \{ \text{Lemma II} \}\]
\( \text{element \_in}(p, r) \)
\[ \equiv \{ p > q, \text{rewrite assumption 1 to element \_in}(p, r) \}\]
True

3: \( p == q : \text{element \_in}(p, \text{node}(p, l, \text{singleright}(r), h)) \)
\[ \equiv \{ \text{definition of element \_in} \}\]
True

\[ \square \]

**Lemma IV:** \( \forall p : \text{Pos} \forall T : \text{fset} : (\text{element \_in}(p, T) \rightarrow \text{element \_in}(p, \text{doublerright}(T))) \)

This lemma says that an element \( p \) should be present in a tree \( T \) after a double right rotation of that tree when \( p \) was present in \( T \) before that rotation.

**Proof:** By case distinction on the structure of \( T \).

Assumption 1: \( \text{element \_in}(p, T) \)
Prove: \( \text{element \_in}(p, \text{doublerright}(T)) \)

\( \text{element \_in}(p, \text{doublerright}(T)) \)
\[ \equiv \{ T == \text{node}(q, l, r, h) \}\]
\( \text{element \_in}(p, \text{doublerright}(\text{node}(q, l, r, h))) \)
\[ \equiv \{ \text{definition of doublerright} \}\]
\( \text{element \_in}(p, \text{singleright}(\text{node}(q, \text{singleleft}(l), r, h))) \)
\[ \equiv \{ \text{Lemma II} \}\]
\( \text{element \_in}(p, \text{node}(q, \text{singleleft}(l), r, h)) \)
\[ \equiv \{ \text{case distinction on p, definition of element \_in} \}\]
1: \( p < q : \text{element \_in}(p, \text{singleleft}(l)) \)
2: \( p > q : \text{element \_in}(p, r) \)
3: \( p == q : \text{element \_in}(p, \text{node}(p, \text{singleleft}(l), r, h)) \)
1: \( p < q : element\_in(p, singleleft(l)) \)
\[\equiv \{ \text{Lemma I} \}\]
\(element\_in(p, l)\)
\[\equiv \{ p < q, rewrite \text{ assumption} 1 \text{ to } element\_in(p, l) \}\]
\(True\)

2: \( p > q : element\_in(p, r) \)
\[\equiv \{ p > q, rewrite \text{ assumption} 1 \text{ to } element\_in(p, r) \}\]
\(True\)

3: \( p == q : element\_in(p, node(p, singleleft(l), r, h)) \)
\[\equiv \{ \text{definition of } element\_in \}\]
\(True\)

\[\square\]

**Lemma V:** \( \forall p:\ Pos \forall T:\ fset : (element\_in(p,T) \rightarrow element\_in(p,rebalance(T))) \)

This lemma makes use of the previous four lemmas and proves that when an element \( p \) is present in a tree \( T \), then it is also present in the tree that is the result of rebalancing \( T \).

**Proof:** by case distinction on the structure of \( T \):

\( T == empty \):

Assumption 1: \( element\_in(p,T) \)
Prove: \( element\_in(p, rebalance(T)) \)

\( element\_in(p, rebalance(T)) \)
\[\equiv \{ T == empty \}\]
\(element\_in(p, rebalance(empty))\)
\[\equiv \{ \text{definition of rebalance} \}\]
\(element\_in(p, empty)\)
\[\equiv \{ rewrite \text{ assumption} 1 \text{ to } element\_in(p, empty) \}\]
\(True\)

\( T == node(q, l, r, h) : \)

Assumption 1: \( element\_in(p,T) \)
Prove: \( element\_in(p, rebalance(T)) \)
element_in(p, rebalance(T))
≡ \{ T == node(q, l, r, h) \}

element_in(p, rebalance(node(q, l, r, h)))
≡ \{ case distinction on gh(r) − gh(l), definition of rebalance \}

1: gh(r) − gh(l) > 1:
1a: gh(right(r)) − gh(left(r)) ≤ −1: element_in(p, doubleleft(node(q, l, r, h)))
1b: gh(right(r)) − gh(left(r)) > −1: element_in(p, singleleft(node(q, l, r, h)))

2: gh(r) − gh(l) < −1:
2a: gh(right(l)) − gh(left(l)) ≥ 1: element_in(p, doubleright(node(q, l, r, h)))
2b: gh(right(l)) − gh(left(l)) < 1: element_in(p, singleright(node(q, l, r, h)))

3: −1 ≤ gh(r) − gh(l) ≤ 1: element_in(p, node(q, l, r, max(gh(l), gh(r)) + 1))

1a: gh(right(r)) − gh(left(r)) ≤ −1: element_in(p, doubleleft(node(q, l, r, h)))
≡ \{ Lemma III \}
element_in(p, node(q, l, r, h))
≡ \{ rewrite assumption 1 to element_in(p, node(q, l, r, h)) \}
True

1b: gh(right(r)) − gh(left(r)) > −1: element_in(p, singleleft(node(q, l, r, h)))
≡ \{ Lemma I \}
element_in(p, node(q, l, r, h))
≡ \{ rewrite assumption 1 to element_in(p, node(q, l, r, h)) \}
True

2a: gh(right(l)) − gh(left(l)) ≥ 1: element_in(p, doubleright(node(q, l, r, h)))
≡ \{ Lemma IV \}
element_in(p, node(q, l, r, h))
≡ \{ rewrite assumption 1 to element_in(p, node(q, l, r, h)) \}
True

2b: gh(right(l)) − gh(left(l)) < 1: element_in(p, singleright(node(q, l, r, h)))
≡ \{ Lemma II \}
element_in(p, node(q, l, r, h))
≡ \{ rewrite assumption 1 to element_in(p, node(q, l, r, h)) \}
True
3: \(-1 \leq gh(r) - gh(l) \leq 1\): \(\text{element}_\text{in}(p, \text{node}(q, l, r, \text{max}(gh(l), gh(r)) + 1))\)
\(\equiv \{\text{definition of element}_\text{in}\}\)

\(\text{True}\)

\[\]

\textbf{Lemma VI:}
\(\forall p : \text{Pos} \forall T : \text{fset} : \neg(\text{element}_\text{in}(p, T)) \rightarrow \neg(\text{element}_\text{in}(p, \text{singleleft}(T)))\)

Lemma VI proves that when an element \(p\) doesn’t occur in a tree \(T\), then \(p\) also won’t occur after applying a single left rotation to \(T\).

\textbf{Proof:} By case distinction on the structure of \(T\).

Assumption 1: \(\neg(\text{element}_\text{in}(p, T))\)
Prove: \(\neg(\text{element}_\text{in}(p, \text{singleleft}(T)))\)

\(\neg(\text{element}_\text{in}(p, \text{singleleft}(T)))\)
\(\equiv \{T == \text{node}(q, l, r, h)\}\)
\(\neg(\text{element}_\text{in}(p, \text{singleleft}(\text{node}(q, l, r, h))))\)
\(\equiv \{\text{definition of singleleft}\}\)
\(\neg(\text{element}_\text{in}(p, \text{node}(\text{key}(r), \text{node}(q, l, \text{left}(r), H'), \text{right}(r), H)))\)
\(\equiv \{\text{case distinction on } p, \text{definition of element}_\text{in}\}\)
\(1: p < \text{key}(r) : \neg(\text{element}_\text{in}(p, \text{node}(q, l, \text{left}(r), H')))\)
\(2: p > \text{key}(r) : \neg(\text{element}_\text{in}(p, \text{right}(r)))\)

1: \(p < \text{key}(r) : \neg(\text{element}_\text{in}(p, \text{node}(q, l, \text{left}(r), H')))\)
\(\equiv \{\text{case distinction on } p, \text{definition of element}_\text{in}\}\)
\(1a: p < q : \neg(\text{element}_\text{in}(p, l))\)
\(1b: p > q : \neg(\text{element}_\text{in}(p, \text{left}(r)))\)

1a: \(p < q : \neg(\text{element}_\text{in}(p, l))\)
\(\equiv \{p < q, \text{rewrite assumption} 1 \text{ to } \neg(\text{element}_\text{in}(p, l))\}\)
\(\text{True}\)

1b: \(p > q : \neg(\text{element}_\text{in}(p, \text{left}(r)))\)
\(\equiv \{p > q, \text{rewrite assumption} 1 \text{ to } \neg(\text{element}_\text{in}(p, r)),\)
\(\neg(\text{element}_\text{in}(p, r)) \rightarrow \neg(\text{element}_\text{in}(p, \text{left}(r)))\}\)
\(\text{True}\)
Lemma VII:

\[ \forall p : Pos \forall T : fset : \neg (element\_in(p,T)) \rightarrow \neg (element\_in(p,singleright(T))) \]

The seventh lemma in this section proves that when an element \( p \) is not present in a tree \( T \) then it will also not be present after doing a single right rotation on \( T \).

**Proof:** By case distinction on the structure of \( T \).

Assumption 1: \( \neg (element\_in(p,T)) \)

Prove: \( \neg (element\_in(p,singleright(T))) \)

\[
\neg (element\_in(p,singleright(T)))
\equiv \{ T == node(q,l,r,h) \}
\neg (element\_in(p,singleright(node(q,l,r,h))))
\equiv \{ definition\ of\ singleright \}
\neg (element\_in(p,node(key(l),left(l),node(q,right(l),r,H'),H)))
\equiv \{ case\ distinction\ on\ p,definition\ of\ element\_in \}
1: p < key(l) : \neg (element\_in(p,left(l)))
2: p > key(l) : \neg (element\_in(p,node(q,right(l),r,H')))\]

**Proof:** By case distinction on the structure of \( T \).

Assumption 1: \( \neg (element\_in(p,T)) \)

Prove: \( \neg (element\_in(p,singleright(T))) \)

\[
\neg (element\_in(p,singleright(T)))
\equiv \{ T == node(q,l,r,h) \}
\neg (element\_in(p,singleright(node(q,l,r,h))))
\equiv \{ definition\ of\ singleright \}
\neg (element\_in(p,node(key(l),left(l),node(q,right(l),r,H'),H)))
\equiv \{ case\ distinction\ on\ p,definition\ of\ element\_in \}
1: p < key(l) : \neg (element\_in(p,left(l)))
2: p > key(l) : \neg (element\_in(p,node(q,right(l),r,H')))\]

**Proof:** By case distinction on the structure of \( T \).

Assumption 1: \( \neg (element\_in(p,T)) \)

Prove: \( \neg (element\_in(p,singleright(T))) \)

\[
\neg (element\_in(p,singleright(T)))
\equiv \{ T == node(q,l,r,h) \}
\neg (element\_in(p,singleright(node(q,l,r,h))))
\equiv \{ definition\ of\ singleright \}
\neg (element\_in(p,node(key(l),left(l),node(q,right(l),r,H'),H)))
\equiv \{ case\ distinction\ on\ p,definition\ of\ element\_in \}
1: p < key(l) : \neg (element\_in(p,left(l)))
2: p > key(l) : \neg (element\_in(p,node(q,right(l),r,H')))\]

**Proof:** By case distinction on the structure of \( T \).

Assumption 1: \( \neg (element\_in(p,T)) \)

Prove: \( \neg (element\_in(p,singleright(T))) \)

\[
\neg (element\_in(p,singleright(T)))
\equiv \{ T == node(q,l,r,h) \}
\neg (element\_in(p,singleright(node(q,l,r,h))))
\equiv \{ definition\ of\ singleright \}
\neg (element\_in(p,node(key(l),left(l),node(q,right(l),r,H'),H)))
\equiv \{ case\ distinction\ on\ p,definition\ of\ element\_in \}
1: p < key(l) : \neg (element\_in(p,left(l)))
2: p > key(l) : \neg (element\_in(p,node(q,right(l),r,H')))\]

**Proof:** By case distinction on the structure of \( T \).

Assumption 1: \( \neg (element\_in(p,T)) \)

Prove: \( \neg (element\_in(p,singleright(T))) \)

\[
\neg (element\_in(p,singleright(T)))
\equiv \{ T == node(q,l,r,h) \}
\neg (element\_in(p,singleright(node(q,l,r,h))))
\equiv \{ definition\ of\ singleright \}
\neg (element\_in(p,node(key(l),left(l),node(q,right(l),r,H'),H)))
\equiv \{ case\ distinction\ on\ p,definition\ of\ element\_in \}
1: p < key(l) : \neg (element\_in(p,left(l)))
2: p > key(l) : \neg (element\_in(p,node(q,right(l),r,H')))\]
2a: \( p < q \) : \( \neg (\text{element } \in (p, \text{right}(l))) \)

\[ \equiv \{ p < q, \text{rewrite assumption 1 to } \neg (\text{element } \in (p, l)), \]  
\[ \quad \quad \neg (\text{element } \in (p, l)) \rightarrow \neg (\text{element } \in (p, \text{right}(l))) \} \]

True

2b: \( p > q \) : \( \neg (\text{element } \in (p, r)) \)

\[ \equiv \{ p > q, \text{rewrite assumption 1 to } \neg (\text{element } \in (p, r)) \} \]

True

\[\Box\]

**Lemma VIII:**

\[ \forall p: \text{Pos} \forall T: \text{fset} : (\neg (\text{element } \in (p, T)) \rightarrow \neg (\text{element } \in (p, \text{doubleleft}(T)))) \]

Lemma VIII says that when \( p \) doesn’t occur in \( T \), it also doesn’t occur when a double left rotation is performed on \( T \).

**Proof:** By case distinction on the structure of \( T \).

Assumption 1: \( \neg (\text{element } \in (p, T)) \)
Prove: \( \neg (\text{element } \in (p, \text{doubleleft}(T))) \)

\[ \neg (\text{element } \in (p, \text{doubleleft}(T))) \]
\[ \equiv \{ T == \text{node}(q, l, r, h) \} \]
\[ \neg (\text{element } \in (p, \text{doubleleft}(\text{node}(q, l, r, h)))) \]
\[ \equiv \{ \text{definition of doubleleft} \} \]
\[ \neg (\text{element } \in (p, \text{singleleft}(\text{node}(q, l, \text{singleright}(r, h)))))) \]
\[ \equiv \{ \text{Lemma VI} \} \]
\[ \neg (\text{element } \in (p, \text{node}(q, l, \text{singleright}(r, h)))) \]
\[ \equiv \{ \text{case distinction on } p, \text{definition of element } \in \} \]
1: \( p < q \) : \( \neg (\text{element } \in (p, l)) \)
2: \( p > q \) : \( \neg (\text{element } \in (p, \text{singleright}(r))) \)

1: \( p < q \) : \( \neg (\text{element } \in (p, l)) \)
\[ \equiv \{ p < q, \text{rewrite assumption 1 to } \neg (\text{element } \in (p, l)) \} \]

True
2: \( p > q \) : \( \neg(\text{element \_ in}(p, \text{singleright}(r))) \)
\[ \equiv \{ \text{Lemma VII} \} \]
\( \neg(\text{element \_ in}(p, r)) \)
\[ \equiv \{ p > q, \text{rewrite assumption 1 to } \neg(\text{element \_ in}(p, r)) \} \]
True

□

**Lemma IX:**
\[ \forall p: \text{Pos} \forall T: \text{fset} : (\neg(\text{element \_ in}(p, T)) \rightarrow \neg(\text{element \_ in}(p, \text{doubleright}(T)))) \]

This lemma is similar to the previous lemma and it says that when \( p \) doesn’t occur in \( T \) then \( p \) will also not occur after a double right rotation of \( T \).

**Proof:** By case distinction on the structure of \( T \).

Assumption 1: \( \neg(\text{element \_ in}(p, T)) \)
Prove: \( \neg(\text{element \_ in}(p, \text{doubleright}(T))) \)

\[ \neg(\text{element \_ in}(p, \text{doubleright}(T))) \]
\[ \equiv \{ T == \text{node}(q, l, r, h) \} \]
\[ \neg(\text{element \_ in}(p, \text{doubleright}(\text{node}(q, l, r, h)))) \]
\[ \equiv \{ \text{definition of doubleright} \} \]
\[ \neg(\text{element \_ in}(p, \text{singleright}(\text{node}(q, \text{singleleft}(l), r, h)))) \]
\[ \equiv \{ \text{Lemma VII} \} \]
\[ \neg(\text{element \_ in}(p, \text{node}(q, \text{singleleft}(l), r, h))) \]
\[ \equiv \{ \text{case distinction on } p, \text{definition of } \text{element \_ in} \} \]
1: \( p < q \) : \( \neg(\text{element \_ in}(p, \text{singleleft}(l))) \)
2: \( p > q \) : \( \neg(\text{element \_ in}(p, r)) \)

1: \( p < q \) : \( \neg(\text{element \_ in}(p, \text{singleleft}(l))) \)
\[ \equiv \{ \text{Lemma VI} \} \]
\[ \neg(\text{element \_ in}(p, l)) \]
\[ \equiv \{ p < q, \text{rewrite assumption 1 to } \neg(\text{element \_ in}(p, l)) \} \]
True
2: \( p > q \) : \( \neg(\text{element } \_ \text{in}(p, r)) \)
\( \equiv \{ p > q, \text{rewrite assumption 1 to } \neg(\text{element } \_ \text{in}(p, r))\} \)

\( True \)

\[ \square \]

**Lemma X:**
\[ \forall p : Pos \forall T : \text{fset} : (\neg(\text{element } \_ \text{in}(p, T)) \rightarrow \neg(\text{element } \_ \text{in}(p, \text{rebalance}(T)))) \]

Lemma X makes use of the four previous lemmas and will prove that when an element \( p \) is not present in a tree \( T \), it is also not present after rebalancing \( T \).

**Proof:** by case distinction on the structure of \( T \):

\( T == \text{empty} \):

Assumption 1: \( \neg(\text{element } \_ \text{in}(p, T)) \)
Prove: \( \neg(\text{element } \_ \text{in}(p, \text{rebalance}(T))) \)

\( \neg(\text{element } \_ \text{in}(p, \text{rebalance}(T))) \)
\( \equiv \{ T == \text{empty} \} \)
\( \neg(\text{element } \_ \text{in}(p, \text{rebalance}(\text{empty}))) \)
\( \equiv \{ \text{definition of rebalance} \} \)
\( \neg(\text{element } \_ \text{in}(p, \text{empty})) \)
\( \equiv \{ \text{definition of element } \_ \text{in} \} \)

\( True \)

\( T == \text{node}(q, l, r, h) \)

Assumption 1: \( \neg(\text{element } \_ \text{in}(p, T)) \)
Prove: \( \neg(\text{element } \_ \text{in}(p, \text{rebalance}(T))) \)

- 89 -
\neg (\text{element\_in}(p, \text{rebalance}(T)))
\equiv \{ T == \text{node}(q, l, r, h) \}
\neg (\text{element\_in}(p, \text{rebalance}(\text{node}(q, l, r, h))))
\equiv \{ \text{case\ distinction\ on}\ gh(r) - gh(l),\ \text{definition\ of\ rebalance} \}

1: gh(r) - gh(l) > 1:

1a: gh(right(r)) - gh(left(r)) \leq -1: \neg (\text{element\_in}(p, \text{doubleleft}(\text{node}(q, l, r, h))))

1b: gh(right(r)) - gh(left(r)) > -1: \neg (\text{element\_in}(p, \text{singleleft}(\text{node}(q, l, r, h))))

2: gh(r) - gh(l) < -1:

2a: gh(right(l)) - gh(left(l)) \geq 1: \neg (\text{element\_in}(p, \text{doubleright}(\text{node}(q, l, r, h))))

2b: gh(right(l)) - gh(left(l)) < 1: \neg (\text{element\_in}(p, \text{singleright}(\text{node}(q, l, r, h))))

3: -1 \leq gh(r) - gh(l) \leq 1: \text{element\_in}(p, \text{node}(q, l, r, \max(gh(l), gh(r)) + 1))

1a: gh(right(r)) - gh(left(r)) \leq -1: \neg (\text{element\_in}(p, \text{doubleleft}(\text{node}(q, l, r, h))))
\equiv \{ \text{Lemma VIII} \}
\neg (\text{element\_in}(p, \text{node}(q, l, r, h)))
\equiv \{ \text{assumption 1} \}

True

1b: gh(right(r)) - gh(left(r)) > -1: \neg (\text{element\_in}(p, \text{singleleft}(\text{node}(q, l, r, h))))
\equiv \{ \text{Lemma VI} \}
\neg (\text{element\_in}(p, \text{node}(q, l, r, h)))
\equiv \{ \text{assumption 1} \}

True

2a: gh(right(l)) - gh(left(l)) \geq 1: \neg (\text{element\_in}(p, \text{doubleright}(\text{node}(q, l, r, h))))
\equiv \{ \text{Lemma IX} \}
\neg (\text{element\_in}(p, \text{node}(q, l, r, h)))
\equiv \{ \text{assumption 1} \}

True

2b: gh(right(l)) - gh(left(l)) < 1: \neg (\text{element\_in}(p, \text{singleright}(\text{node}(q, l, r, h))))
\equiv \{ \text{Lemma VII} \}
\neg (\text{element\_in}(p, \text{node}(q, l, r, h)))
\equiv \{ \text{assumption 1} \}

True
3: \(-1 \leq gh(r) - gh(l) \leq 1\): 
\((\text{element}_\text{in}(p, \text{node}(q, l, r, \max(gh(l), gh(r)) + 1)))\) 
\(\equiv \{\text{assumption\,1}\}\) 

\(\square\)

**Lemma XI:** \(\forall T: fset : (\text{ordered}(T) \rightarrow \text{ordered}(\text{singleleft}(T)))\)

Lemma XI specifies that whenever a tree \(T\) is ordered then \(T\) is still ordered after a single left rotation.

**Proof:** By case distinction on the structure of \(T\).

Assumption 1: \(\text{ordered}(T)\) 
Prove: \(\text{ordered}(\text{singleleft}(T))\)
ordered(singleleft(T))
≡ { T == node(q,l,r,h) }

ordered(singleleft(node(q,l,r,h)))
≡ { definition of singleleft }

ordered(node(key(r), node(q,l,left(r),H'),right(r),H))
≡ { definition of ordered }

ordered(node(q,l,left(r),H') ∧ ordered(right(r)) ∧
allsmaller(node(q,l,left(r),H'),key(r)) ∧ allbigger(right(r),key(r))
≡ { rewrite assumption 1 to ordered(l) ∧ ordered(r) ∧ allsmaller(l,q) ∧ allbigger(r,q),
   ordered(r) → ordered(right(r)) ∧ allbigger(right(r),key(r)) }

ordered(node(q,l,left(r),H') ∧ allsmaller(node(q,l,left(r),H'),key(r))
≡ { definition of ordered }

ordered(l) ∧ ordered(left(r)) ∧ allsmaller(l,q) ∧ allbigger(left(r),q) ∧
allsmaller(node(q,l,left(r),H'),key(r))
≡ { assumption 1, ordered(r) → ordered(left(r)), allbigger(r,q) → allbigger(left(r),q) }

allsmaller(node(q,l,left(r),H'),key(r))
≡ { definition of allsmaller }

q < key(r) ∧ allsmaller(l,key(r)) ∧ allsmaller(left(r),key(r))
≡ { allbigger(r,q) → key(r) > q }

allsmaller(l,key(r)) ∧ allsmaller(left(r),key(r))
≡ { ordered(r) → allsmaller(left(r),key(r)) }

allsmaller(l,key(r))
≡ { definition of allsmaller }

key(l) < key(r) ∧ allsmaller(left(l),key(r)) ∧ allsmaller(right(l),key(r))
≡ { key(l) < q ∧ q < key(r) → key(l) < key(r),
   ordered(l) ∧ q < key(r) → allsmaller(left(l),key(r)) ∧ allsmaller(right(l),key(r)) }

True

□
Lemma XII: \( \forall T : fset : (ordered(T) \rightarrow ordered(singleright(T))) \)

The twelfth lemma is similar to lemma XI, only lemma XII says that a tree \( T \) that is ordered will still be ordered after a single right rotation.

**Proof:** By case distinction on the structure of \( T \).

Assumption 1: \( ordered(T) \)

Prove: \( ordered(singleright(T)) \)
ordered(singleright(T))
≡ {T == node(q, l, r, h)}

ordered(singleright(node(q, l, r, h)))
≡ {definition of singleright}

ordered(node(key(l), left(l), node(q, right(l), r, H'), H))
≡ {definition of ordered}

ordered(left(l)) ∧ ordered(node(q, right(l), r, H')) ∧
alllsmaller(left(l), key(l)) ∧ allbigger(node(q, right(l), r, H'), key(l))
≡ {rewrite assumption 1 to ordered(l) ∧ ordered(r) ∧ allsmaller(l, q) ∧ allbigger(r, q),
ordered(l) → ordered(left(l)) ∧ allsmaller(left(l), key(l))}

ordered(node(q, right(l), r, H')) ∧ allbigger(node(q, right(l), r, H'), key(l))
≡ {definition of ordered}

ordered(right(l)) ∧ ordered(r) ∧ allsmaller(right(l), q) ∧ allbigger(r, q) ∧
alllbigger(node(q, right(l), r, H'), key(l))
≡ {assumption 1, ordered(l) → ordered(right(l)), allsmaller(l, q) → allsmaller(right(l), q)}
alllbigger(node(q, right(l), r, H'), key(l))
≡ {definition of allbigger}

q > key(l) ∧ allbigger(right(l), key(l)) ∧ allbigger(r, key(l))
≡ {allsmaller(l, q) → q > key(l)}
alllbigger(right(l), key(l)) ∧ allbigger(r, key(l))
≡ {ordered(l) → allbigger(right(l), key(l))}
alllbigger(r, key(l))
≡ {definition of allbigger}

key(r) > key(l) ∧ allbigger(left(r), key(l)) ∧ allbigger(right(r), key(l))
≡ {key(l) < q ∧ q < key(r) → key(l) < key(r),
ordered(r) ∧ q > key(l) → allbigger(left(r), key(l)) ∧ allbigger(right(r), key(l))}

True

□

Lemma XIII: ∀T : fset : (ordered(T) → ordered(doubleleft(T)))

This lemma says that a tree T that is ordered stays ordered after a double left rotation.

Proof: By case distinction on the structure of T.

Assumption 1: ordered(T)
Prove: \( \text{ordered}(\text{doubleleft}(T)) \)

\[
\begin{align*}
\text{ordered}(\text{doubleleft}(T)) &
\equiv [T \equiv \text{node}(q, l, r, h)] \\
\text{ordered}(\text{doubleleft}(\text{node}(q, l, r, h))) &
\equiv \{\text{definition of doubleleft}\} \\
\text{ordered}(\text{singleleft}(\text{node}(q, l, \text{singleright}(r), h))) &
\equiv \{\text{Lemma XI}\} \\
\text{ordered}(\text{node}(q, l, \text{singleright}(r), h)) &
\equiv \{\text{definition of ordered}\} \\
\text{ordered}(l) \land \text{ordered}(\text{singleright}(r)) \land \text{allsmaller}(l, q) \land \text{allbigger}(\text{singleright}(r), q) &
\equiv \{\text{rewrite assumption 1 to ordered}(l) \land \text{ordered}(r) \land \text{allsmaller}(l, q) \land \text{allbigger}(r, q)\} \\
\text{ordered}(\text{singleright}(r)) \land \text{allbigger}(\text{singleright}(r), q) &
\equiv \{\text{Lemma XII}\} \\
\text{ordered}(r) \land \text{allbigger}(\text{singleright}(r), q) &
\equiv \{\text{assumption 1, Lemma XXI}\} \\
\text{allbigger}(r, q) &
\equiv \{\text{assumption 1}\} \\
\text{True} &
\end{align*}
\]

\[\Box\]

**Lemma XIV:** \( \forall T : \text{fset} : (\text{ordered}(T) \rightarrow \text{ordered}(\text{doubleright}(T))) \)

This lemma is similar to the previous lemma; only this lemma says that an ordered tree \( T \) is still ordered after performing a double right rotation on it.

**Proof:** By case distinction on the structure of \( T \).

Assumption 1: \( \text{ordered}(T) \)

Prove: \( \text{ordered}(\text{doubleright}(T)) \)
ordered(doublerright(T))
≡ {T == node(q,l,r,h)}
ordered(doublerright(node(q,l,r,h)))
≡ {definition of doublerright}
ordered(singleright(node(q,singleleft(l),r,h))
≡ {Lemma XII}
ordered(node(q,singleleft(l),r,h))
≡ {definition of ordered}
ordered(singleleft(l)) ∧ ordered(r) ∧ allsmaller(singleleft(l),q) ∧ allbigger(r,q)
≡ {rewrite assumption 1 to ordered(l) ∧ ordered(r) ∧ allsmaller(l,q) ∧ allbigger(r,q)}
ordered(singleleft(l)) ∧ allsmaller(singleleft(l),q)
≡ {Lemma XI}
ordered(l) ∧ allsmaller(singleleft(l),q)
≡ {assumption 1, Lemma XVI}
allsmaller(l,q)
≡ {assumption 1}
True

□

Lemma XV: ∀T : fset : (ordered(T) → ordered(rebalance(T)))

An ordered tree T stays ordered after a rebalance operation. This lemma uses the proofs of the four previous lemmas in its sub cases.

Proof: by case distinction on the structure of T:

T == empty:

Assumption 1: ordered(T)
Prove: ordered(rebalance(T))
ordered(\text{rebalance}(T))
\equiv \{ T == empty \}

ordered(\text{rebalance}(empty))
\equiv \{ \text{definition of rebalance} \}

ordered(empty)
\equiv \{ \text{definition of ordered} \}

\text{True}

T == node(q,l,r,h):

Assumption 1: ordered(T)
Prove: ordered(\text{rebalance}(T))

ordered(\text{rebalance}(T))
\equiv \{ T == node(q,l,r,h) \}

ordered(\text{rebalance}(\text{node}(q,l,r,h)))
\equiv \{ \text{case distinction on } gh(r) - gh(l), \text{definition of rebalance} \}

1: gh(r) - gh(l) > 1:
  \begin{align*}
  1a : & \quad gh(\text{right}(r)) - gh(\text{left}(r)) \leq -1 : \text{ordered(\text{doubleleft}(\text{node}(q,l,r,h)))} \\
  1b : & \quad gh(\text{right}(r)) - gh(\text{left}(r)) > -1 : \text{ordered(\text{singleleft}(\text{node}(q,l,r,h)))} \\
  2 : & \quad gh(r) - gh(l) < -1:
  \end{align*}

  \begin{align*}
  2a : & \quad gh(\text{right}(l)) - gh(\text{left}(l)) \geq 1 : \text{ordered(\text{doubleright}(\text{node}(q,l,r,h)))} \\
  2b : & \quad gh(\text{right}(l)) - gh(\text{left}(l)) < 1 : \text{ordered(\text{singleright}(\text{node}(q,l,r,h)))} \\
  3 : & \quad -1 \leq gh(r) - gh(l) \leq 1 : \text{ordered(\text{node}(q,l,r,\max(gh(l), gh(r)) + 1))}
  \end{align*}

1a : gh(\text{right}(r)) - gh(\text{left}(r)) \leq -1 : \text{ordered(\text{doubleleft}(\text{node}(q,l,r,h)))}
\equiv \{ \text{Lemma XIII} \}

ordered(\text{node}(q,l,r,h))
\equiv \{ \text{assumption 1} \}

\text{True}

1b : gh(\text{right}(r)) - gh(\text{left}(r)) > -1 : \text{ordered(\text{singleleft}(\text{node}(q,l,r,h)))}
\equiv \{ \text{Lemma XI} \}

ordered(\text{node}(q,l,r,h))
\equiv \{ \text{assumption 1} \}

\text{True}
2a: gh(right(l)) − gh(left(l)) ≥ 1: ordered(doubleright(node(q,l,r,h)))
≡ {Lemma XIV}
ordered(node(q,l,r,h))
≡ {assumption 1}
True

2b: gh(right(l)) − gh(left(l)) < 1: ordered(singleright(node(q,l,r,h)))
≡ {Lemma XII}
ordered(node(q,l,r,h))
≡ {assumption 1}
True

3: −1 ≤ gh(r) − gh(l) ≤ 1: ordered(node(q,l,r,max(gh(l),gh(r)) + 1))
≡ {assumption 1}
True

□

Lemma XVI: ∀p : Pos ∀T : fset : (allsmaller(T, p) → allsmaller(singuleft(T), p))

When all elements in a tree T are smaller than a given element p, those elements are still smaller than p after performing a single left rotation on T.

Proof: by case distinction on the structure of T:

T == node(q,l,r,h)

Assumption 1: allsmaller(T, p)
Prove: allsmaller(singuleft(T), p)
allsmaller(singleleft(T), p)
≡ \{ T == node(q, l, r, h) \}

allsmaller(singleleft(node(q, l, r, h)), p)
≡ \{ definition of singleleft \}

allsmaller(node(key(r), node(q, l, left(r), H'), right(r), H), p)
≡ \{ definition of allsmaller \}

key(r) < p ∧ allsmaller(node(q, l, left(r), H'), p) ∧ allsmaller(right(r), p)
≡ \{ rewrite assumption \}

allsmaller(r, p) → key(r) < p ∧ allsmaller(right(r), p)

allsmaller(node(q, l, left(r), H'), p)
≡ \{ definition of allsmaller \}

q < p ∧ allsmaller(l, p) ∧ allsmaller(left(r), p)
≡ \{ assumption 1, allsmaller(r, p) → allsmaller(left(r), p) \}

True

Lemma XVII: \( ∀p : Pos \forall T : fset : (allsmaller(T, p) → allsmaller(singleright(T), p)) \)

When all elements in a tree \( T \) are smaller than a given element \( p \), those elements are still smaller than \( p \) after performing a single right rotation on \( T \).

Proof: by case distinction on the structure of \( T \):

\( T == node(q, l, r, h) \)

Assumption 1: allsmaller(T, p)
Prove: allsmaller(singleright(T), p)
allsmaller(singleright(T), p)
≡ \{ T \equiv node(q, l, r, h) \}

allsmaller(singleright(node(q, l, r, h)), p)
≡ \{ definition of singleright \}

allsmaller(node(key(l), left(l), node(q, right(l), r, H'), H), p)
≡ \{ definition of allsmaller \}
key(l) < p \land allsmaller(left(l), p) \land allsmaller(node(q, right(l), r, H'), p)
≡ \{ rewrite assumption \to q < p \land allsmaller(l, p) \land allsmaller(r, p) \}
  \quad \land allsmaller(l, p) \to key(l) < p \land allsmaller(left(l), p))

allsmaller(node(q, right(l), r, H'), p)
≡ \{ definition of allsmaller \}
q < p \land allsmaller(right(l), p) \land allsmaller(r, p)
≡ \{ assumption 1, allsmaller(l, p) \to allsmaller(right(l), p) \}

True

□

Lemma XVIII: \( \forall p : Pos \\forall T : fset : (allsmaller(T, p) \to allsmaller(doubleleft(T), p)) \)

Lemma XVIII says that whenever all elements in a tree \( T \) are smaller than a given element \( p \), all those elements of \( T \) are still smaller than \( p \) after a double left rotation of \( T \).

Proof: by case distinction on the structure of \( T \):

\( T \equiv node(q, l, r, h) \)

Assumption 1: allsmaller(T, p)
Prove: allsmaller(doubleleft(T), p)
allsmaller(doubleleft(T), p)
≡ \{ T == node(q,l,r,h) \}
allsmaller(doubleleft(node(q,l,r,h)), p)
≡ \{ definition of doubleleft \}
allsmaller(singleleft(node(q,l,singleright(r),h)), p)
≡ \{ Lemma XVI \}
allsmaller(node(q,l,singleright(r),h), p)
≡ \{ definition of allsmaller \}
q < p \land allsmaller(l, p) \land allsmaller(singleright(r), p)
≡ \{ rewrite assumption 1 to q < p \land allsmaller(l, p) \land allsmaller(r, p) \}
allsmaller(singleright(r), p)
≡ \{ Lemma XVII \}
allsmaller(r, p)
≡ \{ assumption 1 \}
True

\[\square\]

**Lemma XIX:** \( \forall p: Pos \forall T: fset : (allsmaller(T, p) \rightarrow allsmaller(doubleright(T), p)) \)

Lemma XIX says that whenever all elements in a tree \( T \) are smaller than a given element \( p \), all those elements of \( T \) are still smaller than \( p \) after a double right rotation of \( T \).

**Proof:** by case distinction on the structure of \( T \):

\( T == node(q,l,r,h) \)

Assumption 1: allsmaller(T, p)
Prove: allsmaller(doubleright(T), p)
allsmaller(doubleright(T), p)  
≡ \{ T == node(q, l, r, h) \}  

allsmaller(doubleright(node(q, l, r, h)), p)  
≡ \{ definition of doubleright \}  

allsmaller(singleright(node(q, singleleft(l), r, h)), p)  
≡ \{ Lemma XVII \}  

allsmaller(node(q, singleleft(l), r, h), p)  
≡ \{ definition of allsmaller \}  

q < p \land allsmaller(singleleft(l), p) \land allsmaller(r, p)  
≡ \{ rewrite assumption 1 to q < p \land allsmaller(l, p) \land allsmaller(r, p) \}  

allsmaller(singleleft(l), p)  
≡ \{ Lemma XVI \}  

allsmaller(l, p)  
≡ \{ assumption 1 \}  

True  

□  

Lemma XX  \forall p: \text{Pos} \forall T: \text{fset} : (\text{allbigger}(T, p) \rightarrow \text{allbigger}(singleleft(T), p))  

The lemma with number XX proves that when all elements of T are bigger than p, then they are also bigger than p after a single left rotation of T.  

Proof: by case distinction on the structure of T:  

T == node(q, l, r, h)  

Assumption 1: allbigger(T, p)  
Prove: allbigger(singleleft(T), p)
allbigger(singleleft($T$), $p$)
\[\equiv \{ T \equiv node(q, l, r, h) \}\]

allbigger(singleleft(node($q, l, r, h$)), $p$)
\[\equiv \{ \text{definition of singleleft} \}\]

allbigger(node(key($r$), node($q, l, left(r), H'$), right($r$), $H$), $p$)
\[\equiv \{ \text{definition of allbigger} \}\]

key($r$) > $p$ \& allbigger(node($q, l, left(r), H'$), $p$) \& allbigger(right($r$), $p$)
\[\equiv \{ \text{rewrite assumption 1 to } q > p \& allbigger(l, p) \& allbigger(r, p) \}\]

allbigger(node($q, l, left(r), H'$), $p$)
\[\equiv \{ \text{definition of allbigger} \}\]

$q > p \& allbigger(l, p) \& allbigger(left(r), p)$
\[\equiv \{ \text{assumption 1, allbigger(r, p) \rightarrow allbigger(left(r), p)} \}\]

True

\[\square\]

**Lemma XXI:** $\forall p \forall T : \text{Pos T : fset : allbigger(T, p) \rightarrow allbigger(singleright(T), p)}$

The twenty-first lemma states that when all elements of $T$ are bigger than $p$ it implies that after a single right rotation of $T$, the elements of $T$ will all still be bigger than $p$.

**Proof:** by case distinction on the structure of $T$:

$T \equiv node(q, l, r, h)$

Assumption 1: allbigger($T$, $p$)
Prove: allbigger(singleright($T$), $p$)
allbigger(singleright(T), p)  
\[\equiv \{ T == node(q, l, r, h) \} \]

allbigger(singleright(node(q, l, r, h)), p)  
\[\equiv \{ \text{definition of singleright} \} \]

allbigger(node(key(l), left(l), node(q, right(l), r, H'), H'), p)  
\[\equiv \{ \text{definition of allbigger} \} \]

key(l) > p \land allbigger(left(l), p) \land allbigger(node(q, right(l), r, H'), p)  
\[\equiv \{ \text{rewrite assumption 1 to q > p \land allbigger(l, p) \land allbigger(r, p)} \} \]

allbigger(l, p) \rightarrow key(l) > p \land allbigger(left(l), p)  
\[\equiv \{ \text{definition of allbigger} \} \]

q > p \land allbigger(right(l), p) \land allbigger(r, p)  
\[\equiv \{ \text{assumption 1, allbigger(l, p) \rightarrow allbigger(right(l), p)} \} \]

True

\[\square\]

Lemma XXII: \( \forall p : Pos \forall T : fset : (allbigger(T, p) \rightarrow allbigger(doubleleft(T), p)) \)

This lemma says that after a double left rotation of a tree \( T \), all elements of \( T \) are bigger than \( p \) when they were bigger before the double left rotation of \( T \).

Proof: by case distinction on the structure of \( T \):

\( T == node(q, l, r, h) \)

Assumption 1: \( allbigger(T, p) \)
Prove: \( allbigger(doubleleft(T), p) \)
allbigger(doubleleft(T), p)
≡ {T == node(q,l,r,h)}
allbigger(doubleleft(node(q,l,r,h)), p)
≡ {definition of doubleleft}
allbigger(singleleft(node(q,l,singleright(r),h)), p)
≡ {Lemma XX}
allbigger(node(q,l,singleright(r),h), p)
≡ {definition of allbigger}
q > p ∧ allbigger(l, p) ∧ allbigger(singleright(r), p)
≡ {rewrite assumption 1 to q > p ∧ allbigger(l, p) ∧ allbigger(r, p)}
allbigger(singleright(r), p)
≡ {Lemma XXI}
allbigger(r, p)
≡ {assumption 1}
True

□

Lemma XXIII: ∀p: Pos∀T: fset(allbigger(T, p) → allbigger(doubleright(T), p))

This lemma is similar to lemma XXII, only this lemma proves for a double right rotation that all elements of a tree are bigger than a value p when all these values were bigger than p before the rotation.

Proof: by case distinction on the structure of T:

T == node(q,l,r,h)

Assumption 1: allbigger(T, p)
Prove: allbigger(doubleright(T), p)
allbigger(doubleright(T), p)  
≡ {T == node(q,l,r,h)}  
allbigger(doubleright(node(q,l,r,h)), p)  
≡ {definition of doubleright}  
allbigger(singleright(node(q, singleleft(l), r), p), p)  
≡ {Lemma XXI}  
allbigger(node(q, singleleft(l), r), p)  
≡ {definition of allbigger}  
q > p ∧ allbigger(singleleft(l), p) ∧ allbigger(r, p)  
≡ {rewrite assumption 1 to q > p ∧ allbigger(l, p) ∧ allbigger(r, p)}  
allbigger(singleleft(l), p)  
≡ {Lemma XX}  
allbigger(l, p)  
≡ {assumption 1}  
True

□

Lemma XXIV: ∀ p: Pos ∃ T : fset : (allsmaller(T, p) → allsmaller(rebalance(T), p))

Lemma XXIV will prove that when all elements of T are smaller than p, all elements of T are still smaller than p after T is rebalanced.

Proof: by case distinction on the structure of T:

T == empty:

Assumption 1: allsmaller(T, p)  
Prove: allsmaller(rebalance(T), p)
\[ T \equiv \text{node}(q, l, r, h) : \]

Assumption 1: \( \text{allsmaller}(T, p) \)
Prove: \( \text{allsmaller}(\text{rebalance}(T), p) \)

\( \text{allsmaller}(\text{rebalance}(T), p) \)
\[ \equiv \{ T \equiv \text{node}(q, l, r, h) \} \]
\( \text{allsmaller}(\text{rebalance}(\text{node}(q, l, r, h), p) \)
\[ \equiv \{ \text{case distinction on } gh(r) - gh(l), \text{definition of rebalance} \} \]
1: \( gh(r) - gh(l) > 1 \):
   1a: \( gh(\text{right}(r)) - gh(\text{left}(r)) \leq -1: \text{allsmaller}(\text{doubleleft}(\text{node}(q, l, r, h), p) \)
   1b: \( gh(\text{right}(r)) - gh(\text{left}(l)) > -1: \text{allsmaller}(\text{singleleft}(\text{node}(q, l, r, h), p) \)
2: \( gh(r) - gh(l) < -1 \):
   2a: \( gh(\text{right}(l)) - gh(\text{left}(l)) \geq 1: \text{allsmaller}(\text{doubleright}(\text{node}(q, l, r, h), p) \)
   2b: \( gh(\text{right}(l)) - gh(\text{left}(l)) < 1: \text{allsmaller}(\text{singleright}(\text{node}(q, l, r, h), p) \)
3: \( -1 \leq gh(r) - gh(l) \leq 1: \text{allsmaller}(\text{node}(q, l, r, \max(gh(l), gh(r)) + 1), p \)

1a: \( gh(\text{right}(r)) - gh(\text{left}(r)) \leq -1: \text{allsmaller}(\text{doubleleft}(\text{node}(q, l, r, h), p) \)
\[ \equiv \{ \text{Lemma XVIII} \} \]
\( \text{allsmaller}(\text{node}(q, l, r, h), p) \)
\[ \equiv \{ \text{assumption 1} \} \]
\( \text{True} \)

1b: \( gh(\text{right}(r)) - gh(\text{left}(l)) > -1: \text{allsmaller}(\text{singleleft}(\text{node}(q, l, r, h), p) \]
\[ \equiv \{ \text{Lemma XVI} \} \]
\( \text{allsmaller}(\text{node}(q, l, r, h), p) \)
\[ \equiv \{ \text{assumption 1} \} \]
\( \text{True} \)

2a: \( gh(\text{right}(l)) - gh(\text{left}(l)) \geq 1: \text{allsmaller}(\text{doubleright}(\text{node}(q, l, r, h), p) \]
\[ \equiv \{ \text{Lemma XIX} \} \]
\( \text{allsmaller}(\text{node}(q, l, r, h), p) \)
\[ \equiv \{ \text{assumption 1} \} \]
\( \text{True} \)
Lemma XVII

\[
\forall l, r, h. \quad \text{allsmaller}(\text{node}(q, l, r, h), p) \equiv \text{assumption 1} \quad \text{True}
\]

Lemma XXV: \( \forall p : \text{Pos} \forall T : \text{fset} : (\text{allbigger}(T, p) \rightarrow \text{allbigger}(\text{rebalance}(T), p)) \)

Lemma XXV states that whenever all elements of a tree \( T \) are bigger than a given value \( p \), they will still be bigger than \( p \) after rebalancing \( T \).

Proof: by case distinction on the structure of \( T \):

\( T == \text{empty} : \)

Assumption 1: \( \text{allbigger}(T, p) \)
Prove: \( \text{allbigger}(\text{rebalance}(T), p) \)

\[
\text{allbigger}(\text{rebalance}(T), p) \equiv [ T == \text{empty} ]
\]

\[
\text{allbigger}(\text{rebalance}(\text{empty}), p) \equiv \text{definition of rebalance}
\]

\[
\text{allbigger}(\text{empty}, p) \equiv \text{definition of allbigger}
\]

True

\( T == \text{node}(q, l, r, h) : \)

Assumption 1: \( \text{allbigger}(T, p) \)
Prove: \( \text{allbigger}(\text{rebalance}(T), p) \)
allbigger(rebalance(T), p) 
≡ \{ T == node(q, l, r, h) \}

allbigger(rebalance(node(q, l, r, h)), p) 
≡ \{ case distinction on gh(r) − gh(l), definition of rebalance \}

1: gh(r) − gh(l) > 1:
   1a: gh(right(r)) − gh(left(r)) ≤ −1: allbigger(doubleleft(node(q, l, r, h)), p) 
   1b: gh(right(r)) − gh(left(r)) > −1: allbigger(singleleft(node(q, l, r, h)), p)

2: gh(r) − gh(l) < −1:
   2a: gh(right(l)) − gh(left(l)) ≥ 1: allbigger(doubleright(node(q, l, r, h)), p)
   2b: gh(right(l)) − gh(left(l)) < 1: allbigger(singleright(node(q, l, r, h)), p)

3: −1 ≤ gh(r) − gh(l) ≤ 1: allbigger(node(q, l, r, max(gh(l), gh(r)) + 1), p)

   1a: gh(right(r)) − gh(left(r)) ≤ −1: allbigger(doubleleft(node(q, l, r, h)), p) 
   ≡ \{ Lemma XXII \}

   allbigger(node(q, l, r, h), p) 
   ≡ \{ assumption 1 \}
   True

   1b: gh(right(r)) − gh(left(r)) > −1: allbigger(singleleft(node(q, l, r, h)), p) 
   ≡ \{ Lemma XX \}

   allbigger(node(q, l, r, h), p) 
   ≡ \{ assumption 1 \}
   True

   2a: gh(right(l)) − gh(left(l)) ≥ 1: allbigger(doubleright(node(q, l, r, h)), p) 
   ≡ \{ Lemma XXIII \}

   allbigger(node(q, l, r, h), p) 
   ≡ \{ assumption 1 \}
   True

   2b: gh(right(l)) − gh(left(l)) ≥ 1: allbigger(singleright(node(q, l, r, h)), p) 
   ≡ \{ Lemma XXI \}

   allbigger(node(q, l, r, h), p) 
   ≡ \{ assumption 1 \}
   True
3: \(-1 \leq gh(r) - gh(l) \leq 1: \text{allbigger}(\text{node}(q,l,r,\max(gh(l),gh(r))+1), p)\)
\[\equiv \{\text{assumption 1}\}\]
True

□

**Theorem 1:** \(\forall p: \text{Pos} \forall T: \text{fset}: (\text{element}_\text{in}(p,\text{insert}(p,T)))\)

This first theorem says that for every positive number and for every finite set as AVL tree it should hold that whenever an element is inserted into an AVL tree, it is present in the AVL tree, i.e. when you search for an element after an insert, and the element isn’t deleted in the meantime, you should find it.

Proof: by induction on the structure of \(T\):

Base case: \(T == \text{empty}\)

\[\text{element}_\text{in}(p,\text{insert}(p,T))\]
\[\equiv \{T == \text{empty}\}\]
\[\text{element}_\text{in}(p,\text{insert}(p,\text{empty}))\]
\[\equiv \{\text{definition of insert}\}\]
\[\text{element}_\text{in}(p,\text{node}(p,\text{empty},\text{empty},1))\]
\[\equiv \{\text{definition of element}_\text{in}\}\]
True

Induction step: \(T == \text{node}(q,l,r,h)\)

Induction Hypothesis 1: \(\text{element}_\text{in}(p,\text{insert}(p,l))\)

Induction Hypothesis 2: \(\text{element}_\text{in}(p,\text{insert}(p,r))\)

\[\text{element}_\text{in}(p,\text{insert}(p,T))\]
\[\equiv \{T == \text{node}(q,l,r,h)\}\]
\[\text{element}_\text{in}(p,\text{insert}(p,\text{node}(q,l,r,h)))\]
\[\equiv \{\text{case distinction on p, definition of insert}\}\]
1: \(p < q: \text{element}_\text{in}(p,\text{rebalance}(\text{node}(q,\text{insert}(p,l),r,h)))\)
2: \(p > q: \text{element}_\text{in}(p,\text{rebalance}(\text{node}(q,l,\text{insert}(p,r),h)))\)
3: \(p == q: \text{element}_\text{in}(p,\text{node}(p,l,r,h))\)
1: \( p < q : \text{element\_in}(p, \text{rebalance}(\text{node}(q, \text{insert}(p, l, r, h)))) \)
\[ \equiv \{ \text{Lemma V} \} \]
\[ \text{element\_in}(p, \text{node}(q, \text{insert}(p, l, r, h))) \]
\[ \equiv \{ p < q, \text{definition of element\_in} \} \]
\[ \text{element\_in}(p, \text{insert}(p, l)) \]
\[ \equiv \{ \text{Induction Hypothesis 1} \} \]
\[ \text{True} \]

2: \( p > q : \text{element\_in}(p, \text{rebalance}(\text{node}(q, \text{insert}(p, l, r, h)))) \)
\[ \equiv \{ \text{Lemma V} \} \]
\[ \text{element\_in}(p, \text{node}(q, l, \text{insert}(p, r, h))) \]
\[ \equiv \{ p > q, \text{definition of element\_in} \} \]
\[ \text{element\_in}(p, \text{insert}(p, r)) \]
\[ \equiv \{ \text{Induction Hypothesis 2} \} \]
\[ \text{True} \]

3: \( p == q : \text{element\_in}(p, \text{node}(p, l, r, h)) \)
\[ \equiv \{ \text{definition of element\_in} \} \]
\[ \text{True} \]

□

**Theorem 2:** \( \forall p : \text{Pos} \forall T : \text{fset} : (\text{ordered}(T) \rightarrow \neg(\text{element\_in}(p, \text{delete}(p, T)))) \)

Theorem two states that whenever an AVL tree is ordered, it should hold that whenever an element is deleted, it cannot be found in the AVL tree anymore.

Proof: by induction on the structure of \( T \):

Base case: \( T == \text{empty} \)
Assumption 1: \( \text{ordered}(T) \)
Prove: \( \neg(\text{element\_in}(p, \text{delete}(p, T))) \)
\neg(element\ _\ in(p, delete(p,T)))
\equiv \{T == empty\}
\neg(element\ _\ in(p, delete(p,empty)))
\equiv \{definition\ of\ delete\}
\neg(element\ _\ in(p,empty))
\equiv \{definition\ of\ element\ _\ in\}
True

Induction step: T == node(q,l,r,h)
Induction Hypothesis 1: \neg(element\ _\ in(p, delete(p,l)))
Induction Hypothesis 2: \neg(element\ _\ in(p, delete(p,r)))
Assumption 1: ordered(T)
Prove: \neg(element\ _\ in(p, delete(p,T)))

\neg(element\ _\ in(p, delete(p,T)))
\equiv \{T == node(q,l,r,h)\}
\neg(element\ _\ in(p, delete(p,node(q,l,r,h))))
\equiv \{case\ distibution\ on\ p, definition\ of\ delete\}
1: p < q : \neg(element\ _\ in(p, rebalance(node(q,delete(p,l),r,h))))
2: p > q : \neg(element\ _\ in(p, rebalance(node(q,l,delete(p,r),h))))
3: p == q : \neg(element\ _\ in(p, delete(p,node(p,l,r,h))))
\equiv \{case\ distinction\ on\ l\ and\ r, definition\ of\ delete\}
3a: l == empty \land r == empty : \neg(element\ _\ in(p,empty))
3b: l \neq empty \land r == empty : \neg(element\ _\ in(p,l))
3c: l == empty \land r \neq empty : \neg(element\ _\ in(p,r))
3d: l \neq empty \land r \neq empty : \neg(element\ _\ in(p,rebalance(node(min(r),l,delete(min(r),r),h))))

1: p < q : \neg(element\ _\ in(p, rebalance(node(q,delete(p,l),r,h))))
\equiv \{Lemma\ X\}
\neg(element\ _\ in(p,node(q,delete(p,l),r,h)))
\equiv \{p < q, definition\ of\ element\ _\ in\}
\neg(element\ _\ in(p, delete(p,l)))
\equiv \{Induction\ Hypothesis\ 1\}
True
2: \( p > q : \neg (\text{element}_\text{in}(p, \text{rebalance}(\text{node}(q,l, \text{delete}(p,r), h)))) \)
\[ \equiv \{ \text{Lemma X} \} \]
\( \neg (\text{element}_\text{in}(p, \text{node}(q,l, \text{delete}(p,r), h))) \)
\[ \equiv \{ p > q, \text{definition of element}_\text{in} \} \]
\( \neg (\text{element}_\text{in}(p, \text{delete}(p,r))) \)
\[ \equiv \{ \text{Induction Hypothesis 2} \} \]
True

3a: \( p == q \land l == \text{empty} \land r == \text{empty} : \neg (\text{element}_\text{in}(p, \text{empty})) \)
\[ \equiv \{ \text{definition of element}_\text{in} \} \]
True

3b: \( p == q \land l \neq \text{empty} \land r == \text{empty} : \neg (\text{element}_\text{in}(p, l)) \)
\[ \equiv \{ \text{rewrite assumption 1 to ordered }(l) \land \text{ordered }(r) \land \text{allsmaller}(l,q) \land \text{allbigger}(r,q), \]
allsmaller\((l,q) \rightarrow \neg (\text{element}_\text{in}(p, l)) \} \]
True

3c: \( l == \text{empty} \land r \neq \text{empty} : \neg (\text{element}_\text{in}(p, r)) \)
\[ \equiv \{ \text{rewrite assumption 1 to ordered }(l) \land \text{ordered }(r) \land \text{allsmaller}(l,q) \land \text{allbigger}(r,q), \]
allbigger\((r,q) \rightarrow \neg (\text{element}_\text{in}(p, r)) \} \]
True

3d: \( p == q \land l \neq \text{empty} \land r \neq \text{empty} : \)
\( \neg (\text{element}_\text{in}(p, \text{rebalance}(\text{node}(\text{min}(r),l, \text{delete}(\text{min}(r),r), h)))) \)
\[ \equiv \{ \text{Lemma X} \} \]
\( \neg (\text{element}_\text{in}(p, \text{node}(\text{min}(r),l, \text{delete}(\text{min}(r),r),h))) \)
\[ \equiv \{ p == q \rightarrow \neg (\text{element}_\text{in}(p, \text{node}(\text{min}(r),l, \text{delete}(\text{min}(r),r),h))) \} \]
True

\[ \square \]

**Theorem 3:**
\[ \forall p,q : \text{Pos} : \forall T : \text{fset} : (\text{allsmaller}(T,p) \rightarrow q < p \rightarrow \text{allsmaller}(\text{insert}(q,T),p)) \]

This third theorem says that when all elements in an AVL tree \( T \) are smaller than a value \( p \) and when a value \( q \) is smaller than \( p \), then after inserting \( q \) into \( T \), all elements in \( T \) are still smaller than \( p \).

Proof: by induction on the structure of \( T \):
Base case: $T == empty$
Assumption 1: $\text{allsmaller}(empty, p)$
Assumption 2: $q < p$
Prove: $\text{allsmaller}(insert(q, T), p)$

\[
\text{allsmaller}(insert(q, T), p) \\
\equiv \{ T == empty \} \\
\text{allsmaller}(insert(q, empty), p) \\
\equiv \{ \text{definition of insert} \} \\
\text{allsmaller}(node(q, empty, empty, 1), p) \\
\equiv \{ \text{definition of allsmaller} \} \\
q < p \land \text{allsmaller}(empty, p) \land \text{allsmaller}(empty, p) \\
\equiv \{ \text{assumption 2} \} \\
\text{allsmaller}(empty, p) \land \text{allsmaller}(empty, p) \\
\equiv \{ \text{definition of allsmaller} \} \\
\text{True}
\]

Induction step: $T == node(k, l, r, h)$
Induction Hypothesis 1: $\text{allsmaller}(l, p) \rightarrow \text{allsmaller}(insert(q, l), p)$
Induction Hypothesis 2: $\text{allsmaller}(r, p) \rightarrow \text{allsmaller}(insert(q, r), p)$
Assumption 1: $\text{allsmaller}(node(k, l, r, h), p)$
Assumption 2: $q < p$
Prove: $\text{allsmaller}(insert(q, T), p)$

\[
\text{allsmaller}(insert(q, T), p) \\
\equiv \{ T == node(k, l, r, h) \} \\
\text{allsmaller}(insert(q, node(k, l, r, h)), p) \\
\equiv \{ \text{case distinction on q, definition of insert} \} \\
1. q < k : \text{allsmaller}(rebalance(node(k, insert(q, l), r, h), p)) \\
2. q > k : \text{allsmaller}(rebalance(node(k, l, insert(q, r), h), p)) \\
3. q == k : \text{allsmaller}(node(q, l, r, h), p)
\]
1: \textit{allsmaller}(\textit{rebalance}(\textit{node}(k, \textit{insert}(q, l), r, h), p))
\equiv \{\textit{Lemma XXIV}\}
\textit{allsmaller}(\textit{node}(k, \textit{insert}(q, l), r, h), p)
\equiv \{\textit{definition of allsmaller}\}
k < p \land \textit{allsmaller}(\textit{insert}(q, l), p) \land \textit{allsmaller}(r, p)
\equiv \{\textit{rewrite assumption 1 to } k < p \land \textit{allsmaller}(l, p) \land \textit{allsmaller}(r, p)\}
\textit{allsmaller}(\textit{insert}(q, l), p)
\equiv \{\textit{Induction Hypothesis 1}\}
\textit{allsmaller}(l, p)
\equiv \{\textit{assumption 1}\}
\textit{True}

2: \textit{allsmaller}(\textit{rebalance}(\textit{node}(k, l, \textit{insert}(q, r), h), p))
\equiv \{\textit{Lemma XXIV}\}
\textit{allsmaller}(\textit{node}(k, l, \textit{insert}(q, r), h), p)
\equiv \{\textit{definition of allsmaller}\}
k < p \land \textit{allsmaller}(l, p) \land \textit{allsmaller}(\textit{insert}(q, r), p)
\equiv \{\textit{rewrite assumption 1 to } k < p \land \textit{allsmaller}(l, p) \land \textit{allsmaller}(r, p)\}
\textit{allsmaller}(\textit{insert}(q, r), p)
\equiv \{\textit{Induction Hypothesis 2}\}
\textit{allsmaller}(r, p)
\equiv \{\textit{assumption 1}\}
\textit{True}

3: \textit{allsmaller}(\textit{node}(q, l, r, h), p)
\equiv \{\textit{definition of allsmaller}\}
q < p \land \textit{allsmaller}(l, p) \land \textit{allsmaller}(r, p)
\equiv \{\textit{assumption 2}\}
\textit{allsmaller}(l, p) \land \textit{allsmaller}(r, p)
\equiv \{q = k, \textit{rewrite assumption 1 to } q < p \land \textit{allsmaller}(l, p) \land \textit{allsmaller}(r, p)\}
\textit{True}

□
Theorem 4:
\[ \forall p, q : \text{Pos} \implies \forall T : \text{fset} : (\text{allbigger}(T, p) \rightarrow q > p \rightarrow \text{allbigger}(\text{insert}(q, T), p)) \]

The fourth theorem says that when all elements in an AVL tree \( T \) are bigger than a value \( p \) and when a value \( q \) is bigger than \( p \), then after inserting \( q \) into \( T \), all elements in \( T \) are still bigger than \( p \).

Proof: by induction on the structure of \( T \):

Base case: \( T == \text{empty} \)
Assumption 1: \( \text{allbigger}((\text{empty}, p) \)\)
Assumption 2: \( q > p \)
Prove: \( \text{allbigger}(\text{insert}(q, T), p) \)

\[
\text{allbigger}(\text{insert}(q, T), p) \\
\equiv \{T == \text{empty}\} \\
\text{allbigger}(\text{insert}(q, \text{empty}), p) \\
\equiv \{\text{definition of insert}\} \\
\text{allbigger}(\text{node}(q, \text{empty}, \text{empty}, l), p) \\
\equiv \{\text{definition of allbigger}\} \\
q > p \land \text{allbigger}(\text{empty}, p) \land \text{allbigger}(\text{empty}, p) \\
\equiv \{\text{assumption 2}\} \\
\text{allbigger}(\text{empty}, p) \land \text{allbigger}(\text{empty}, p) \\
\equiv \{\text{definition of allbigger}\} \\
\text{True} \\
\]

Induction step: \( T == \text{node}(k, l, r, h) \)
Induction Hypothesis 1: \( \text{allbigger}(l, p) \rightarrow \text{allbigger}(\text{insert}(q, l), p) \)
Induction Hypothesis 2: \( \text{allbigger}(r, p) \rightarrow \text{allbigger}(\text{insert}(q, r), p) \)
Assumption 1: \( \text{allbigger}(\text{node}(k, l, r, h), p) \)
Assumption 2: \( q > p \)
Prove: \( \text{allbigger}(\text{insert}(q, T), p) \)
allbigger(insert(q,T), p) 
≡ {T == node(k,l,r,h)}
allbigger(insert(q,node(k,l,r,h)), p) 
≡ {case distinction on q, definition of insert}
1: q < k : allbigger(rebalance(node(k,insert(q,l),r,h), p))
2: q > k : allbigger(rebalance(node(k,l,insert(q,r),h), p))
3: q == k : allbigger(node(q,l,r,h), p)

1: q < k : allbigger(rebalance(node(k,insert(q,l),r,h), p))
≡ {Lemma XXV}
allbigger(node(k,insert(q,l),r,h), p) 
≡ {definition of allbigger}
k > p ∧ allbigger(insert(q,l), p) ∧ allbigger(r, p) 
≡ {rewrite assumption 1 to k > p ∧ allbigger(l, p) ∧ allbigger(r, p)}
allbigger(insert(q,l), p) 
≡ {Induction Hypothesis 1}
allbigger(l, p) 
≡ {assumption 1}
True

2: q > k : allbigger(rebalance(node(k,l,insert(q,r),h), p))
≡ {Lemma XXV}
allbigger(node(k,l,insert(q,r),h), p) 
≡ {definition of allbigger}
k > p ∧ allbigger(l, p) ∧ allbigger(insert(q,r), p) 
≡ {rewrite assumption 1 to k > p ∧ allbigger(l, p) ∧ allbigger(r, p)}
allbigger(insert(q,r), p) 
≡ {Induction Hypothesis 2}
allbigger(r, p) 
≡ {assumption 1}
True
3: \( q := k \ast \text{allbigger}(node(q,l,r,h), p) \)
\[\equiv \{\text{definition of allbigger}\} \]
\( q > p \land \text{allbigger}(l, p) \land \text{allbigger}(r, p) \)
\[\equiv \{\text{assumption 2}\} \]
\( \text{allbigger}(l, p) \land \text{allbigger}(r, p) \)
\[\equiv \{q := k, \text{rewrite assumption 1 to } q > p \land \text{allbigger}(l, p) \land \text{allbigger}(r, p)\} \]
True

□

Theorem 5: \( \forall p: Pos \forall T: fset : (\text{ordered}(T) \rightarrow \text{ordered}(\text{insert}(p,T))) \)

Theorem 5 states that a tree \( T \) that is ordered is still ordered after inserting a value \( p \) into it.

Proof: by induction on the structure of \( T \):

Base case: \( T == \text{empty} \)
Assumption 1: \( \text{ordered}(T) \)
Prove: \( \text{ordered}(\text{insert}(p,T)) \)

\( \text{ordered}(\text{insert}(p,T)) \)
\[\equiv \{T == \text{empty}\} \]
\( \text{ordered}(\text{insert}(p,\text{empty})) \)
\[\equiv \{\text{definition of insert}\} \]
\( \text{ordered}(\text{node}(p,\text{empty},\text{empty},1)) \)
\[\equiv \{\text{definition of ordered}\} \]
\( \text{ordered}(\text{empty}) \land \text{ordered}(\text{empty}) \land \text{allsmaller}(\text{empty}, p) \land \text{allbigger}(\text{empty}, p) \)
\[\equiv \{\text{definition of ordered, definition of allsmaller, definition of allbigger}\} \]
True

Induction step: \( T == \text{node}(q,l,r,h) \)
Induction Hypothesis 1: \( \text{ordered}(l) \rightarrow \text{ordered}(\text{insert}(p,l)) \)
Induction Hypothesis 2: \( \text{ordered}(r) \rightarrow \text{ordered}(\text{insert}(p,r)) \)
Assumption 1: \( \text{ordered}(T) \)
Prove: \( \text{ordered}(\text{insert}(p,T)) \)
ordered(insert(p,T))
≡ {T == node(q,l,r,h)}
ordered(insert(p,node(q,l,r,h)))
≡ {case distinction on p, definition of insert}
1: p < q : ordered(rebalance(node(q,insert(p,l),r,h)))
2: p > q : ordered(rebalance(node(q,l,insert(p,r),h)))
3: p == q : ordered(node(p,l,r,h))

1: p < q : ordered(rebalance(node(q,insert(p,l),r,h)))
≡ {Lemma XV}
ordered(node(q,insert(p,l),r,h))
≡ {definition of ordered}
ordered(insert(p,l)) ∧ ordered(r) ∧ allsmaller(insert(p,l),q) ∧ allbigger(r,q)
≡ {rewrite assumption 1 to ordered(l) ∧ ordered(r) ∧ allsmaller(l,q) ∧ allbigger(r,q)}
ordered(insert(p,l)) ∧ allsmaller(insert(p,l),q)
≡ {Induction Hypothesis 1}
ordered(l) ∧ allsmaller(insert(p,l),q)
≡ {assumption 1}
allsmaller(insert(p,l),q)
≡ {Theorem 3}
allsmaller(l,q)
≡ {assumption 1}
True
\begin{align*}
2: x &> q : \text{ordered}(\text{rebalance}(\text{node}(q,l,\text{insert}(x,r),h))) \\
&\equiv \{\text{Lemma XV}\} \\
&\text{ordered}(\text{node}(q,l,\text{insert}(x,r),h)) \\
&\equiv \{\text{definition of ordered}\} \\
&\text{ordered}(l) \land \text{ordered}(\text{insert}(x,r)) \land \text{allsmaller}(l,q) \land \text{allbigger}(\text{insert}(x,r),q) \\
&\equiv \{\text{rewrite assumption 1 to ordered}(l) \land \text{ordered}(r) \land \text{allsmaller}(l,q) \land \text{allbigger}(r,q)\} \\
&\text{ordered}(\text{insert}(x,r)) \land \text{allbigger}(\text{insert}(x,r),q) \\
&\equiv \{\text{Induction Hypothesis 2}\} \\
&\text{ordered}(r) \land \text{allbigger}(\text{insert}(x,r),q) \\
&\equiv \{\text{assumption 1}\} \\
&\text{allbigger}(\text{insert}(x,r),q) \\
&\equiv \{\text{Theorem 4}\} \\
&\text{allbigger}(r,q) \\
&\equiv \{\text{assumption 1}\} \\
&\text{True}
\end{align*}

\begin{align*}
3: x &== q : \text{ordered}(\text{node}(x,l,r,h)) \\
&\equiv \{\text{definition of ordered}\} \\
&\text{ordered}(l) \land \text{ordered}(r) \land \text{allsmaller}(l,x) \land \text{allbigger}(r,x) \\
&\equiv \{x == q\} \\
&\text{ordered}(l) \land \text{ordered}(r) \land \text{allsmaller}(l,q) \land \text{allbigger}(r,q) \\
&\equiv \{\text{rewrite assumption 1 to ordered}(l) \land \text{ordered}(r) \land \text{allsmaller}(l,q) \land \text{allbigger}(r,q)\} \\
&\text{True}
\end{align*}

\[\blacksquare\]

**Theorem 6:**
\[\forall p, q : \text{Pos} : \forall T : \text{fset} : (\text{allsmaller}(T, p) \rightarrow q < p \rightarrow \text{allsmaller}(\text{delete}(q,T), p))\]

The sixth theorem states that when all elements in an AVL tree \( T \) are smaller than a value \( p \) and when a value \( q \) is smaller than \( p \), then after deleting \( q \) from \( T \), all elements in \( T \) are still smaller than \( p \).

Proof: by induction on the structure of \( T \):

Base case: \( T == \text{empty} \)
Assumption 1: \( \text{allsmaller}(\text{empty}, p) \)
Assumption 2: \( q < p \)
Prove: \( \text{allsmaller}(\text{delete}(q, T), p) \)

\( \text{allsmaller}(\text{delete}(q, T), p) \)
\[ \equiv \{ T == \text{empty} \} \]
\( \text{allsmaller}(\text{delete}(q, \text{empty}), p) \)
\[ \equiv \{ \text{definition of delete} \} \]
\( \text{allsmaller}(\text{empty}, p) \)
\[ \equiv \{ \text{definition of allsmaller} \} \]

True

Induction step: \( T == \text{node}(k, l, r, h) \)

Induction Hypothesis 1: \( \text{allsmaller}(l, p) \rightarrow \text{allsmaller}(\text{delete}(q, l), p) \)

Induction Hypothesis 2: \( \text{allsmaller}(r, p) \rightarrow \text{allsmaller}(\text{delete}(q, r), p) \)

Assumption 1: \( \text{allsmaller}(\text{node}(k, l, r, h), p) \)

Assumption 2: \( q < p \)

Prove: \( \text{allsmaller}(\text{delete}(q, T), p) \)

\( \text{allsmaller}(\text{delete}(q, T), p) \)
\[ \equiv \{ T == \text{node}(k, l, r, h) \} \]
\( \text{allsmaller}(\text{delete}(q, \text{node}(k, l, r, h)), p) \)
\[ \equiv \{ \text{case distinction on } q, \text{definition of delete} \} \]

1: \( q < k : \text{allsmaller}(\text{rebalance}(\text{node}(k, \text{delete}(q, l), r, h), p)) \)

2: \( q > k : \text{allsmaller}(\text{rebalance}(\text{node}(k, l, \text{delete}(q, r), h), p)) \)

3: \( q == k : \text{allsmaller}(\text{delete}(q, \text{node}(q, l, r, h)), p) \)
\[ \equiv \{ \text{case distinction on } l \text{ and } r, \text{definition of delete} \} \]

3a: \( q == k \land r == \text{empty} \land l == \text{empty} : \text{allsmaller}(\text{empty}, p) \)

3b: \( q == k \land r == \text{empty} \land l \neq \text{empty} : \text{allsmaller}(l, p) \)

3c: \( q == k \land r \neq \text{empty} \land l == \text{empty} : \text{allsmaller}(r, p) \)

3d: \( q == k \land r \neq \text{empty} \land l \neq \text{empty} : \text{allsmaller}(\text{rebalance}(\text{node}(\text{min}(r), l, \text{delete}(\text{min}(r), r), h), p)) \)
1. \( q < k \) : \( \text{allsmaller}(\text{rebalance}(\text{node}(k, \text{delete}(q, l), r, h), p)) \)
\[\equiv \{\text{Lemma XXIV}\}\]
\[\text{allsmaller}(\text{node}(k, \text{delete}(q, l), r, h), p)\]
\[\equiv \{\text{definition of allsmaller}\}\]
\[k < p \land \text{allsmaller}(\text{delete}(q, l), p) \land \text{allsmaller}(r, p)\]
\[\equiv \{\text{rewrite assumption 1 to } k < p \land \text{allsmaller}(l, p) \land \text{allsmaller}(r, p)\}\]
\[\text{allsmaller}(\text{delete}(q, l), p)\]
\[\equiv \{\text{Induction Hypothesis 1}\}\]
\[\text{allsmaller}(l, p)\]
\[\equiv \{\text{assumption 1}\}\]
True

2. \( q > k \) : \( \text{allsmaller}(\text{rebalance}(\text{node}(k, l, \text{delete}(q, r), h), p)) \)
\[\equiv \{\text{Lemma XXIV}\}\]
\[\text{allsmaller}(\text{node}(k, l, \text{delete}(q, r), h), p)\]
\[\equiv \{\text{definition of allsmaller}\}\]
\[k < p \land \text{allsmaller}(l, p) \land \text{allsmaller}(\text{delete}(q, r), p)\]
\[\equiv \{\text{rewrite assumption 1 to } k < p \land \text{allsmaller}(l, p) \land \text{allsmaller}(r, p)\}\]
\[\text{allsmaller}(\text{delete}(q, r), p)\]
\[\equiv \{\text{Induction Hypothesis 2}\}\]
\[\text{allsmaller}(r, p)\]
\[\equiv \{\text{assumption 1}\}\]
True

3a : \( q == k \land r == \text{empty} \land l == \text{empty} \) : \( \text{allsmaller}() \)
\[\equiv \{\text{definition of allsmaller}\}\]
True

3b : \( q == k \land r == \text{empty} \land l \neq \text{empty} \) : \( \text{allsmaller}() \)
\[\equiv \{\text{rewrite assumption 1 to } k < p \land \text{allsmaller}(l, p) \land \text{allsmaller}(r, p)\}\]
True

3c : \( q == k \land r \neq \text{empty} \land l == \text{empty} \) : \( \text{allsmaller}() \)
\[\equiv \{\text{rewrite assumption 1 to } k < p \land \text{allsmaller}(l, p) \land \text{allsmaller}(l, p)\}\]
True
Lemma XXIV

\[\text{Lemma XXIV}\]
\[
\text{allsmaller}(\text{node}(\text{min}(r), l, \text{delete}(\text{min}(r), r), h), p)
\]
\[
\equiv \{\text{definition of allsmaller}\}
\]
\[
\text{min}(r) < p \land \text{allsmaller}(l, p) \land \text{allsmaller}(\text{delete}(\text{min}(r), r), p)
\]
\[
\equiv \{\text{rewrite assumption 1 to } k < p \land \text{allsmaller}(l, p) \land \text{allsmaller}(r, p)\}\]
\[
\text{min}(r) < p \land \text{allsmaller}(\text{delete}(\text{min}(r), r), p)
\]
\[
\equiv \{\text{allsmaller}(\text{node}(k, l, r, h), p) \rightarrow \text{min}(r) < p\}
\]
\[
\text{allsmaller}(\text{delete}(\text{min}(r), r), p)
\]
\[
\equiv \{\text{Induction Hypothesis 2}\}
\]
\[
\text{allsmaller}(r, p)
\]
\[
\equiv \{\text{assumption 1}\}
\]
\[\text{True}\]

\[\square\]

Theorem 7:
\[\forall p, q : \text{Pos} : \forall T : \text{fset} : (\text{allbigger}(T, p) \rightarrow q > p \rightarrow \text{allbigger}(\text{delete}(q, T), p))\]

Theorem 7 states that when all elements in a tree \( T \) are bigger than a given value \( p \) and there is a value \( q \) that is also bigger than \( p \), then after deleting \( q \) from \( T \), all elements in \( T \) are still bigger than \( p \).

Proof: by induction on the structure of \( T \) :

Base case: \( T == \text{empty} \)
Assumption 1: \( \text{allbigger}(\text{empty}, p) \)
Assumption 2: \( q < p \)
Prove: \( \text{allbigger}(\text{delete}(q, T), p) \)

\[\text{allbigger}(\text{delete}(q, T), p)\]
\[
\equiv \{T == \text{empty}\}
\]
\[\text{allbigger}(\text{delete}(q, \text{empty}), p)\]
\[
\equiv \{\text{definition of delete}\}
\]
\[\text{allbigger}(\text{empty}, p)\]
\[
\equiv \{\text{definition of allbigger}\}
\]
\[\text{True}\]
Induction step: $T = node(k, l, r, h)$

Induction Hypothesis 1: $allbigger(l, p) \rightarrow allbigger(delete(q, l), p)$

Induction Hypothesis 2: $allbigger(r, p) \rightarrow allbigger(delete(q, r), p)$

Assumption 1: $allbigger(node(k, l, r, h), p)$

Assumption 2: $q > p$

Prove: $allbigger(delete(q, T), p)$

\[ allbigger(delete(q, T), p) \]
\[ \equiv \{ T == node(k, l, r, h) \} \]

\[ allbigger(delete(q, node(k, l, r, h)), p) \]
\[ \equiv \{ case \ distinction \ on \ q, \ definition \ of \ delete \} \]

1. $q < k$ : $allbigger(rebalance(node(k, delete(q, l), r), h), p))$

2. $q > k$ : $allbigger(rebalance(node(k, l, delete(q, r), h), p))$

3. $q == k$ : $allbigger(delete(q, node(k, l, r, h)), p)$

\[ \equiv \{ case \ distinction \ on \ l \ and \ r, \ definition \ of \ delete \} \]

3a: $q == k \land r == empty \land l == empty : allbigger(empty, p)$

3b: $q == k \land r == empty \land l \not= empty : allbigger(l, p)$

3c: $q == k \land r == empty \land l == empty : allbigger(r, p)$

3d: $q == k \land r \not= empty \land l == empty : allbigger(rebalance(node(min(r), l, delete(min(r), r), h), p))$

1. $q < k$ : $allbigger(rebalance(node(k, delete(q, l), r, h), p))$

\[ \equiv \{ Lemma \ XXV \} \]

allbigger(node(k, delete(q, l), r, h), p)

\[ \equiv \{ definition \ of \ allsmaller \} \]

$k > p \land allbigger(delete(q, l), p) \land allbigger(r, p)$

\[ \equiv \{ rewrite \ assumption \ 1 \ to \ k > p \land allbigger(l, p) \land allbigger(r, p) \} \]

allbigger(delete(q, l), p)

\[ \equiv \{ Induction \ Hypothesis \ 1 \} \]

allbigger(l, p)

\[ \equiv \{ assumption \ 1 \} \]

True
2. \( q > k : \text{allbigger}(\text{rebalance}(\text{node}(k, l, \text{delete}(q, r), h), p)) \)

\[ \equiv \{ \text{Lemma XXV} \} \]

\[ \text{allbigger}(\text{node}(k, l, \text{delete}(q, r), h), p) \]

\[ \equiv \{ \text{definition of allbigger} \} \]

\[ k > p \land \text{allbigger}(l, p) \land \text{allbigger}(\text{delete}(q, r), p) \]

\[ \equiv \{ \text{rewrite assumption 1 to } k > p \land \text{allbigger}(l, p) \land \text{allbigger}(r, p) \} \]

\[ \text{allbigger}(\text{delete}(q, r), p) \]

\[ \equiv \{ \text{Induction Hypothesis 2} \} \]

\[ \text{allbigger}(r, p) \]

\[ \equiv \{ \text{assumption 1} \} \]

True

3a. \( q == k \land r == \text{empty} \land l == \text{empty} : \text{allbigger}(\text{empty}, p) \)

\[ \equiv \{ \text{definition of allbigger} \} \]

True

3b. \( q == k \land r == \text{empty} \land l \neq \text{empty} : \text{allbigger}(l, p) \)

\[ \equiv \{ \text{rewrite assumption 1 to } k > p \land \text{allbigger}(l, p) \land \text{allbigger}(r, p) \} \]

True

3c. \( q == k \land r \neq \text{empty} \land l == \text{empty} : \text{allbigger}(r, p) \)

\[ \equiv \{ \text{rewrite assumption 1 to } k > p \land \text{allbigger}(l, p) \land \text{allbigger}(l, p) \} \]

True
3d: q == k ∧ r ≠ empty ∧ l ≠ empty : allbigger(rebalance(node(min(r), l, delete(min(r), r), h), p))
≡ {Lemma XXV}
allbigger(node(min(r), l, delete(min(r), r), h), p)
≡ {definition of allbigger}
min(r) > p ∧ allbigger(l, p) ∧ allbigger(delete(min(r), r), p)
≡ {rewrite assumption 1 to k > p ∧ allbigger(l, p) ∧ allbigger(r, p)}
min(r) > p ∧ allbigger(delete(min(r), r), p)
≡ {allbigger(node(k, l, r, h), p) → min(r) > p}
allbigger(delete(min(r), r), p)
≡ {Induction Hypothesis 2}
allbigger(r, p)
≡ {assumption 1}
True

□

**Theorem 8:** ∀p : Pos : ∀T : fset : (ordered(T) → ordered(delete(p, T)))

Theorem number 8 says that it should hold for any tree T and any positive number p that an ordered tree T is still ordered after deleting an element p from it.

Proof: by induction on the structure of T:

Base case: T == empty
Assumption 1: ordered(T)
Prove: ordered(delete(p, T))

ordered(delete(p, T))
≡ {T == empty}
ordered(delete(p, empty))
≡ {definition of delete}
ordered(empty)
≡ {definition of ordered}
True

Induction step: T == node(q, l, r, h)
Induction Hypothesis 1: ordered(l) → ordered(delete(p, l))
Induction Hypothesis 2: \( \text{ordered}(r) \rightarrow \text{ordered}(\text{delete}(p, r)) \)
Assumption 1: \( \text{ordered}(T) \)
Prove: \( \text{ordered}(\text{delete}(p, T)) \)

\[
\text{ordered}(\text{delete}(p, T)) \\
\equiv \{T \equiv \text{node}(q, l, r, h)\} \\
\text{ordered}(\text{delete}(p, \text{node}(q, l, r, h))) \\
\equiv \{\text{case distinction on } p, \text{definition of } \text{delete}\} \\
1: p < q \triangleright \text{ordered}(\text{rebalance}(\text{node}(q, \text{delete}(p, l), r, h))) \\
2: p > q \triangleright \text{ordered}(\text{rebalance}(\text{node}(q, l, \text{delete}(p, r), h))) \\
3: p = q \triangleright \text{ordered}(\text{delete}(p, \text{node}(p, l, r, h))) \\
\equiv \{\text{case distinction on } l \text{ and } r, \text{definition of } \text{delete}\} \\
3a: l = \emptyset \land r = \emptyset \triangleright \text{ordered}(\emptyset) \\
3b: l = \emptyset \land r = \emptyset \triangleright \text{ordered}(l) \\
3c: l = \emptyset \land r \neq \emptyset \triangleright \text{ordered}(r) \\
3d: l \neq \emptyset \land r = \emptyset \triangleright \text{ordered}(\text{rebalance}(\text{node}(\text{min}(r), l, \text{delete}(\text{min}(r), r), h))) \\
1: p < q \triangleright \text{ordered}(\text{rebalance}(\text{node}(q, \text{delete}(p, l), r, h))) \\
\equiv \{\text{Lemma XV}\} \\
\text{ordered}(\text{node}(q, \text{delete}(p, l), r, h)) \\
\equiv \{\text{definition of } \text{ordered}\} \\
\text{ordered}(\text{delete}(p, l)) \land \text{ordered}(r) \land \text{allsmaller}(\text{delete}(p, l), q) \land \text{allbigger}(r, q) \\
\equiv \{\text{rewrite assumption 1 to } \text{ordered}(l) \land \text{ordered}(r) \land \text{allsmaller}(l, q) \land \text{allbigger}(r, q)\} \\
\text{ordered}(\text{delete}(p, l)) \land \text{allsmaller}(\text{delete}(p, l), q) \\
\equiv \{\text{Induction Hypothesis 1}\} \\
\text{ordered}(l) \land \text{allsmaller}(\text{delete}(p, l), q) \\
\equiv \{\text{assumption 1}\} \\
\text{allsmaller}(\text{delete}(p, l), q) \\
\equiv \{\text{Theorem 6}\} \\
\text{allsmaller}(l, q) \\
\equiv \{\text{assumption 1}\} \\
\text{True}
2: \( x > q : \text{ordered}(\text{rebalance}(\text{node}(q,l,\text{delete}(x,r),h))) \)
\[ \equiv \{ \text{Lemma XV} \} \]
\( \text{ordered}(\text{node}(q,l,\text{delete}(x,r),h)) \)
\[ \equiv \{ \text{definition of ordered} \} \]
\( \text{ordered}(l) \land \text{ordered}(\text{delete}(x,r)) \land \text{allsmaller}(l,q) \land \text{allbigger}(\text{delete}(x,r),q) \)
\[ \equiv \{ \text{rewrite assumption 1 to ordered}(l) \land \text{ordered}(r) \land \text{allsmaller}(l,q) \land \text{allbigger}(r,q) \} \]
\( \text{ordered}(\text{delete}(x,r)) \land \text{allbigger}(\text{delete}(x,r),q) \)
\[ \equiv \{ \text{Induction Hypothesis 2} \} \]
\( \text{ordered}(r) \land \text{allbigger}(\text{delete}(x,r),q) \)
\[ \equiv \{ \text{assumption 1} \} \]
\( \text{allbigger}(\text{delete}(x,r),q) \)
\[ \equiv \{ \text{Theorem 7} \} \]
\( \text{allbigger}(r,q) \)
\[ \equiv \{ \text{assumption 1} \} \]
\( \text{True} \)

3a: \( l == \text{empty} \land r == \text{empty} : \text{ordered}(\text{empty}) \)
\[ \equiv \{ \text{definition of ordered} \} \]
\( \text{True} \)

3b: \( l \neq \text{empty} \land r == \text{empty} : \text{ordered}(l) \)
\[ \equiv \{ \text{rewrite assumption 1 to ordered}(l) \land \text{ordered}(r) \land \text{allsmaller}(l,q) \land \text{allbigger}(r,q) \} \]
\( \text{True} \)

3c: \( l == \text{empty} \land r \neq \text{empty} : \text{ordered}(r) \)
\[ \equiv \{ \text{rewrite assumption 1 to ordered}(l) \land \text{ordered}(r) \land \text{allsmaller}(l,q) \land \text{allbigger}(r,q) \} \]
\( \text{True} \)
3d : \( l = \text{empty} \land r = \text{empty} : \text{ordered}(\text{rebalance}(\text{node}(\text{min}(r), l, \text{delete}(\text{min}(r), r), h))) \)

\[ \equiv \{ \text{Lemma XV} \} \]

\text{ordered}(\text{node}(\text{min}(r), l, \text{delete}(\text{min}(r), r), h))

\[ \equiv \{ \text{definition of ordered} \} \]

\text{ordered}(l) \land \text{ordered}(\text{delete}(\text{min}(r), r)) \land \text{allsmaller}(l, \text{min}(r)) \land \text{allbigger}(\text{delete}(\text{min}(r), r), \text{min}(r))

\[ \equiv \{ \text{rewrite assumption 1 to ordered}(l) \land \text{ordered}(r) \land \text{allsmaller}(l, q) \land \text{allbigger}(r, q) \} \]

\text{ordered}(\text{delete}(\text{min}(r), r)) \land \text{allsmaller}(l, \text{min}(r)) \land \text{allbigger}(\text{delete}(\text{min}(r), r), \text{min}(r))

\[ \equiv \{ \text{Induction Hypothesis 2} \} \]

\text{ordered}(r) \land \text{allsmaller}(l, \text{min}(r)) \land \text{allbigger}(\text{delete}(\text{min}(r), r), \text{min}(r))

\[ \equiv \{ \text{assumption 1} \} \]

\text{allsmaller}(l, \text{min}(r)) \land \text{allbigger}(\text{delete}(\text{min}(r), r), \text{min}(r))

\[ \equiv \{ \text{allsmaller}(l, q) \rightarrow \text{allsmaller}(l, \text{min}(r)) \} \]

True

\[ \square \]

**Theorem 9:** \( \forall T : \text{fset} : (\text{gh}(T) = \text{height}(T)) \)

This theorem says that the value of the height parameter of a node in an AVL tree should be equal to the value that you get when you *calculate* the height of that node.

Proof: by induction on the structure of \( T : \)

Base case: \( T = \text{empty} \)

\( \text{gh}(T) = \text{height}(T) \)

\[ \equiv \{ T = \text{empty} \} \]

\( \text{gh}(\text{empty}) = \text{height}(\text{empty}) \)

\[ \equiv \{ \text{definition of gh, definition of height} \} \]

\( 0 = 0 \)

\[ \equiv \]

True

Induction step: \( T = \text{node}(q, l, r, h) \)

Induction Hypothesis: \( h - 1 = \text{height}(l) \lor h - 1 = \text{height}(r) \)
1: \( \text{max}(\text{height}(l), \text{height}(r)) \equiv l : h \equiv \text{succ}(<\text{height}(l)>)) \\
\equiv \\
h - 1 \equiv \text{height}(l) \\
\equiv \{\text{Induction Hypothesis}\} \\
\text{True} \\

2: \( \text{max}(\text{height}(l), \text{height}(r)) \equiv r : h \equiv \text{succ}(<\text{height}(r)>)) \\
\equiv \\
h - 1 \equiv \text{height}(r) \\
\equiv \{\text{Induction Hypothesis}\} \\
\text{True} \\
\Box
9. Performance

This chapter will describe the performance tests I did for the four finite set specifications that I made. The first section will describe the behavior and the performance of the finite set as AVL tree under the four rewriters that are available in the mCRL2 toolset. The second section of this chapter will cover the behavior of the mCRL2 JITty c rewriter (where JIT stands for Just In Time and where c stands for compiled) for each of the four finite set specifications.

9.1. Finite set as AVL tree

For the finite set as AVL tree I created a process that inserts $n$ elements from 1 to $n$ into an empty AVL tree, thus in an empty finite set. This process is specified as follows:

1. proc $p(s : fset, m, n : Pos) =$
   $(m <= n) ->$ tau.$p($insert($m, s$), $m + 1$, $n$);
2. init $p$($empty$, 1, 2000);

The line with number 1 specifies a process called $p$ that has three parameters. The first parameter $s$ represents a finite set as an AVL tree; the second parameter $m$ specifies a positive number just as the last parameter $n$ does. The process $p$ does the following: as long as the value of parameter $m$ is smaller than, or equal to $n$, a $tau$ step is done and that step is followed by process $p$ with as first parameter the finite set $s$ into which element $m$ is inserted. The second parameter is the positive number $m$ increased by 1 and the last parameter is the positive number $n$ that stays the same throughout the whole process. In this way, all elements from 1 to $n$ will be inserted.

The initialization of the process parameters is done in line number 2 by specifying that the process starts with an empty set. It starts at position 1 and ends at position $n$. In this particular example $n$ is 2000, so the elements 1 to 2000 will be added to the empty finite set $empty$.

To measure the performance of the insert function for the finite set as AVL tree, the four different rewriters were used to generate state spaces for this process. This was done for each rewriter for $n = 100$ to $n = 1000$, and from $n = 1000$ to $n = 10000$, i.e. from 100 to 1000, $n$ was increased by 100 and from 1000 to 10000 $n$ was increased by 1000. So in total this yielded 19 test values per rewriter.

The process that was followed for each of the rewriters was that first a linear process specification of the mCRL2 specification was generated by using the tool mcrl22lps. From the resulting linear process specification a state space was generated by using the tool lps2lts. The following flags were used in the call of lps2lts to specify which rewriter was to be used: -Rinnerc for the innermost c rewriter, -Rjitty for the JITty rewriter and -Rjittyc for the JITty c rewriter. For the default rewriter, i.e. the innermost rewriter, no
flag had to be specified. The parameter for the lps2lts call was the file (with a linear process specification) from which a state space had to be generated.

For each rewriter, the time to generate a state space was calculated by using the command `time` that can be used under Linux. I created four equations to calculate a constant for the following orders: $O(\log(n))$, $O(n\log(n))$, $O(n^2)$ and $O(n^3)$. These equations are specified as follows:

1. $O(\log(n))$:
   \[ C = \frac{1}{\log(n)} \times \text{number of seconds} \]
2. $O(n\log(n))$:
   \[ C = \frac{1}{n\log(n)} \times \text{number of seconds} \]
3. $O(n^2)$:
   \[ C = \frac{1}{n^2} \times \text{number of seconds} \]
4. $O(n^3)$:
   \[ C = \frac{1}{n^3} \times \text{number of seconds} \]

Table 9.1 will display the results (i.e. the time in seconds to generate a state space) of the different rewrite strategies applied to the process as specified above. For the columns of innermost c and JITty c, the complete time is taken into account, i.e. the compilation time is included in the result.

<table>
<thead>
<tr>
<th>n</th>
<th>innermost</th>
<th>innermost c</th>
<th>JITty</th>
<th>JITty c</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>1,292</td>
<td>3,488</td>
<td>0,336</td>
<td>5,752</td>
</tr>
<tr>
<td>200</td>
<td>4,668</td>
<td>3,752</td>
<td>0,86</td>
<td>5,6</td>
</tr>
<tr>
<td>300</td>
<td>10,424</td>
<td>4,22</td>
<td>1,744</td>
<td>5,756</td>
</tr>
<tr>
<td>400</td>
<td>18,933</td>
<td>4,82</td>
<td>3,264</td>
<td>5,916</td>
</tr>
<tr>
<td>500</td>
<td>28,957</td>
<td>5,572</td>
<td>4,464</td>
<td>6,052</td>
</tr>
<tr>
<td>600</td>
<td>41,826</td>
<td>6,56</td>
<td>6,708</td>
<td>6,288</td>
</tr>
<tr>
<td>700</td>
<td>57,295</td>
<td>7,684</td>
<td>9,148</td>
<td>6,624</td>
</tr>
<tr>
<td>800</td>
<td>74,924</td>
<td>9,048</td>
<td>12,176</td>
<td>7,012</td>
</tr>
<tr>
<td>900</td>
<td>94,893</td>
<td>10,392</td>
<td>14,932</td>
<td>7,344</td>
</tr>
<tr>
<td>1000</td>
<td>117,003</td>
<td>12,612</td>
<td>18,061</td>
<td>7,848</td>
</tr>
<tr>
<td>2000</td>
<td>481,174</td>
<td>38,426</td>
<td>78,136</td>
<td>16,329</td>
</tr>
<tr>
<td>3000</td>
<td>1127,814</td>
<td>82,181</td>
<td>205,184</td>
<td>32,278</td>
</tr>
<tr>
<td>4000</td>
<td>2011,577</td>
<td>151,997</td>
<td>351,469</td>
<td>54,843</td>
</tr>
<tr>
<td>5000</td>
<td>3117,945</td>
<td>233,954</td>
<td>589,624</td>
<td>86,997</td>
</tr>
<tr>
<td>6000</td>
<td>4541,023</td>
<td>342,181</td>
<td>894,803</td>
<td>131,752</td>
</tr>
<tr>
<td>7000</td>
<td>6374,726</td>
<td>483,15</td>
<td>1195,134</td>
<td>177,607</td>
</tr>
<tr>
<td>8000</td>
<td>8207,024</td>
<td>619,27</td>
<td>1574,65</td>
<td>234,354</td>
</tr>
<tr>
<td>9000</td>
<td>10447,628</td>
<td>788,781</td>
<td>2040,155</td>
<td>301,306</td>
</tr>
<tr>
<td>10000</td>
<td>12919,327</td>
<td>975,712</td>
<td>2588,781</td>
<td>381,783</td>
</tr>
</tbody>
</table>

Table 9.1 Time results for AVL tree
From the graph in Figure 9.1, which displays the time in seconds it takes for each rewriter to generate a state space, it can be concluded that the JITty c rewriter yields the best performance results for the finite set as AVL tree. The worst performance for this finite set implementation is obtained by the innermost rewriter. This difference in time is due to the fact that the innermost rewriting strategy rewrites everything and the JITty c strategy only rewrites things when necessary. Based on these times, the four equations specified above are calculated for each rewriter and plotted in graphs that will come across in the next figures.

Starting with the first equation, the next graph, displayed in Figure 9.2, gives an overview of the equation: \( C = (1/\log(n)) \ast \text{number of seconds} \), which is calculated for each of the four rewriting strategies. The value of C is a value that should evolve into a constant value when the performance of the process from the linear process specification appears to be of \( O(\log(n)) \).
As can be seen in Figure 9.2, the performance of the insert function for the finite set as AVL tree doesn’t behave as expected, which would be in $O(\log(n))$ time. This can be seen in the graph: for each of the 4 rewriters, the result of the equation, the value of $C$, keeps growing when $n$ grows. This means that the equation never goes to a constant value and therefore the performance will not be of $O(\log(n))$. A possible cause for this can be the internal structure of the insert function, in which recursive calls are done and in which calls are done to other functions (such as rebalance, to make sure that the tree remains a valid AVL tree after inserting a new element).

The next equation that is calculated for each of the rewriters is the following: $C = (1/n \log(n)) \times \text{number of seconds}$. This equation tests whether the performance of the process is of $O(n \log(n))$. Results for this equation are plotted in the next graph, Figure 9.3.
As already said, the results of the equation \( C = \frac{1}{n \log(n)} \) \#number of seconds\ are displayed in Figure 9.3. Just as in Figure 9.2, the performance of the innermost rewriter is the worst of the four rewriters. From the graph can also be concluded that this equation doesn’t evolve into a value that remains constant and therefore it can be concluded that the insert for the AVL tree does not perform in \( O(n \log(n)) \) time. The next equation for which I tested the insert function is the following: \( C = \frac{1}{n^2} \) \#number of seconds\. First the graph for this equation is shown in Figure 9.4 and after that, a description of that graph is given.
Figure 9.4 displays the results for the equation $C = (1/n^2) \times \text{number of seconds}$ for 19 different values of $n$. As can be seen in the graph, the value of $C$ evolves into a constant value for each different rewriting strategy. This means that for inserting $n$ elements, the performance is of $O(n^2)$, which means that for inserting 1 element, the performance is of $O(n)$. This performance of $O(n)$ is in contrast with what one would expect from an insert into an AVL tree, because the actual performance of an insert should be of $O(\log(n))$. An explanation for this behavior is that the insert function for the AVL tree makes use of a number of recursive calls and calls to the rebalance function. This is because for each insert that is done; a call of the rebalance function is done, which can in its place make calls to singleleft, singleright, doubleleft or doubleright when rotation is necessary.
The last equation that was posed was the equation to calculate a possible constant value for $O(n^3)$. This equation was the following: $C = (1/n^3) \times \text{number of seconds}$. Though it is not very obvious from the graph, for $n^3$, the value of $C$ doesn’t evolve into a constant, i.e. the value of $C$ keeps decreasing while the value of $n$ gets bigger.

Now these four graphs have been described, it can be concluded that since the equation for $C = (1/n^2) \times \text{number of seconds}$ evolves into a constant value for $C$ while $n$ gets bigger, this means that the insert function for the AVL tree behaves in $n^2$ time for $n$ inserts and thus behaves in $O(n)$ time for 1 insert.

As already said, the difference in expected performance, $O(\log(n))$, and the actual performance $O(n)$, can be explained by the fact that insert makes use of recursive calls to itself and that each call of insert is enveloped by a call of rebalance to make sure that the AVL tree that is started with remains a correct AVL tree after the insertion of an element.

The next section of this chapter deals with the mCRL2 JITty c rewriter. It examines its behavior for each of the four different implementations of the finite set, i.e. the ordered List, the AVL tree, the unique tree with left-balance and the unique tree that uses list conversion.
9.2. mCRL2 JITty c rewriter

In this section, the performance of the four finite set implementations under the mCRL2 JITty c rewriter is described and reviewed. The mCRL2 JITty c rewriter is a rewriter that can be used while transforming a linear process specification into a labeled transition system. JIT in JITty c stands for Just In Time and this rewriter can be used by adding the flag \(-Rjittyc\) to the call of \textit{lps2lts}. More information about Just In Time rewriting can be found in [9] and [10].

For the ordered List implementation the following process was used:

1. \texttt{proc p(s : fset, m, n : Pos) =}
   
   \hspace{1cm} \begin{align*}
   & (m \leq n) \rightarrow \tau.p(\text{insert}(m, s), m + 1, n); \\
   & 2. \text{init } p([], 1, 7000);
   \end{align*}

For the AVL tree, the process specified in section 9.1 was used. For the unique tree that is left-balanced, the following process was specified:

1. \texttt{proc p(s : fset, m, n : Pos) =}
   
   \hspace{1cm} \begin{align*}
   & (m \leq n) \rightarrow \tau.p(\text{insert}(m, s), m + 1, n); \\
   & 2. \text{init } p(\text{empty}, 1, 300);
   \end{align*}

The last process used, the process for the unique tree with List conversion, was the following:

1. \texttt{proc p(s : fsetTree, m, n : Pos) =}
   
   \hspace{1cm} \begin{align*}
   & (m \leq n) \rightarrow \tau.p(\text{insert}(m, s), m + 1, n); \\
   & 2. \text{init } p(\text{empty}, 1, 100);
   \end{align*}

Each of these four processes starts with an empty finite set and inserts \(n\) consecutive elements starting from 1. Now I will start with showing a table with the times for generating state spaces for each of the four finite set implementations with the mCRL2 JITty c rewriter. These times can be found in Table 9.2.
The time in seconds to generate a state space is measured for 19 different values of $n$ starting at $n = 100$ and ending at $n = 10000$. All values of $n$ that are tested can be found in the table above. What already can be concluded from Table 9.2 is the fact that the unique tree that is left-balanced yields the worst performance, e.g. for the value $n = 10000$, this specification is a factor 4 times slower than the ordered List version. An explanation for this behavior is the fact that the insert function for this implementation makes use of multiple other functions which in turn also make recursive calls and calls to other functions.

The best performance under the JITty c rewriter is that of the AVL tree. This is due to the fact that the insert for an AVL tree is built from an if-statement which is not rewritten as a whole, but only the parts that are necessary. The insert for the other two unique trees are also built from if-statements, but their performance is slower due to the amount of calls to other functions. A graphical overview of the results from Table 9.2 can be found in Figure 9.6.
What was already concluded from the results in Table 9.2 can be seen much better in Figure 9.6. The performance of the left-balanced tree is the worst performance of the four specifications under the JITty c rewriter. The difference in time grows explosively while \( n \) grows. E.g., at \( n = 1000 \), the time for generating a state space for the left-balanced tree is almost 2.2 times slower than that of the best performance (AVL tree) at \( n = 5000 \), this is already 7.5 times slower and at \( n = 10000 \) this is still worse, namely 7.9 times slower.

Despite the fact that the AVL tree implementation of a finite set is not unique, i.e., for a finite set there is not 1 unique AVL tree, this implementation yields the best results for \( n \) consecutive inserts into an empty finite set.
10. Applications

In this section, 2 possible applications of the finite set implementation are described. Section 10.1 describes a simplified version of the Sliding Window Protocol as it is delivered with the mCRL2 toolset. The next section, section 10.2 gives an example of an adhoc network protocol that was translated from μCRL and adapted in such a way that finite sets are used.

10.1. Simplified Sliding Window Protocol

The sliding window protocol was adapted in such a way that data was removed and the Boolean buffer in the specification was represented by a finite set. In this section I will compare two versions of the Simplified Sliding Window Protocol. The first version uses a finite set as AVL tree and the second version uses a finite set as unique tree using List conversion. The performance of these two specifications was measured by measuring the time to generate a state space for varying sizes of the constant value \( n \) that is used in the protocol. Tests were done for \( n = 1 \) to \( n = 9 \). In this section, the performance of both of the finite set specifications under the mCRL2 JITty c rewriter is compared.

The most important difference between the orginal Simplified Sliding Window Protocol and the version that uses finite sets is that the Boolean buffer (see line 2 of the mCRL2 code below) is represented by a finite set instead of a \( \text{Nat} \to \text{Bool} \). The mCRL2 code for this protocol (using a finite set as an AVL tree) is the following (the code for the AVL tree itself is omitted here):

1. % Sort definition.
2. sort BBuf = fsetAVL;

3. % Mappings.
4. map
5.   n : Pos;
6.   nextempty_mod : Nat \# BBuf \# Nat \# Pos -> Nat;

7. % Variables.
8. var
9.   i, j, m: Nat;
10.  n':Pos;
11.  b : BBuf;

12. % Equations.
13. eqn
14.  n = 5;
15.  element_in(i mod n', b) && m>0 -> nextempty_mod(i, b, m, n') = 
       nextempty_mod((i+1) mod 2*n', b, Int2Nat(m-1), n');
16.  !(element_in(i mod n', b) && m>0) -> nextempty_mod(i, b, m, n') = 
       i mod 2*n';
17. % Actions.
18. act
19.   r1, s4;
20.   s2, r2, c2, s3, r3, c3 : Nat;
21.   s5, r5, c5, s6, r6, c6 : Nat;
22.   i;

23. % Process declaration.
24. proc
25.   S(l, m : Nat) =
26.     (m<n) -> r1.S(l, m+1) +
     sum k:Nat. (k<m) -> s2((l+k) mod 2*n).S(l, m) +
     sum k:Nat. r6(k).S(k, (m-k+1) mod 2*n);
27.   R(l':Nat, b:BBuf) =
28.     sum k:Nat. r3(k).(((k-l') mod (2*n) < n) ->
     R(l', insert(k mod n, b)) <> R(l', b)) +
     element_in(l' mod n, b) ->
     s4.R((l'+1) mod 2*n, delete(l' mod n, b)) +
     s5(nextempty_mod(l',b,n,n)).R(l', b);
29.   K = sum k:Nat. r2(k).(i.s3(k)+i).K;
30.   L = sum k:Nat. r5(k).(i.s6(k)+i).L;
31.   SWP =
32.     allow({c2, c3, c5, c6, i, r1, s4},
33.       comm({r2|s2->c2, r3|s3->c3, r5|s5->c5, r6|s6->c6},
34.       S(0, 0) || K || L || R(0, empty)));
35. % Initialization.
36. init SWP;
For both the AVL tree version and the version with a unique tree using List conversion, state spaces were generated for the values n = 1 to n = 9. This n is a constant value that is specified in line 14 in the code above. In Table 10.1, the time results of generating state spaces can be found.

<table>
<thead>
<tr>
<th>n</th>
<th>AVL tree</th>
<th>Unique tree List conversion</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>6,784</td>
<td>6,964</td>
</tr>
<tr>
<td>2</td>
<td>6,876</td>
<td>7,068</td>
</tr>
<tr>
<td>3</td>
<td>7,46</td>
<td>7,472</td>
</tr>
<tr>
<td>4</td>
<td>10,292</td>
<td>9,184</td>
</tr>
<tr>
<td>5</td>
<td>25,969</td>
<td>15,324</td>
</tr>
<tr>
<td>6</td>
<td>86,453</td>
<td>35,054</td>
</tr>
<tr>
<td>7</td>
<td>314,699</td>
<td>89,281</td>
</tr>
<tr>
<td>8</td>
<td>1361,669</td>
<td>251,439</td>
</tr>
<tr>
<td>9</td>
<td>5182,251</td>
<td>951,687</td>
</tr>
</tbody>
</table>

Table 10.1 Time results for SSWP

A graphical overview of Table 10.1 can be seen in Figure 10.1.
What can be concluded from both Table 10.1 and Figure 10.1 is that at the point where the constant value \( n \) is equal to 4, the performance of the unique tree using List conversion gets the advantage over the AVL tree implementation. What also can be concluded from these results and the fact that generating a state space for the original Simplified Sliding Window Protocol (i.e. the original Sliding Window Protocol without the use of data and databuffers) took already 1813,295 seconds for a value of 1 for \( n \). This shows that the finite set implementation performs better than the use of the \( \text{Nat} \rightarrow \text{Bool} \) structure that was originally used for the Boolean buffer.

### 10.2. Adhoc Network Protocol

This section describes an example of an adhoc network protocol that was originally written by Prof. Dr. J.C. van de Pol in µCRL. I translated this example to a mCRL2 example and added the functionality of finite sets. First the functionality of this protocol is described shortly. After that a part of the code will be shown in this section, and it will be made clear were and how finite sets are used in this protocol.

This protocol is a specification of a dynamic adhoc network protocol in which so-called hosts can move to each other or move from each other and so define a set of neighbours. Hosts are also able to send messages to neighbours, receive messages from neighbours and broadcast messages to neighbours.

Every host has a HostRecord in which a HostName, a MessageList and a HostList are present. These HostRecords together form a HostRecordList which is declared as a finite set as ordered List. MessageList and HostList are also represented as finite sets as ordered Lists.

The way in which the HostRecordList, the MessageList and the HostList are represented is the following:

1. sort HostList = fsetList;
2. sort HostRecordList = fsetListHRL;
3. sort MessageList = fsetListM;
4. sort fsetList = List(HostName);
5. sort fsetListHRL = List(HostRecord);
6. sort fsetListM = List(Message);

For each of these three sorts, the functionality for the finite set as ordered List was added to the code for the adhoc protocol. This functionality consisted for this protocol of the insert function, the delete function and the element_in function. Another part of code that was added, was an ordering for HostName, HostRecord and Message. Such an ordering is necessary for keeping an ordered List ordered.

The ordering for HostName will be shown below:
1. sort HostName = struct A | B | C;
2. map
3.    lt: HostName # HostName -> Bool; % Less than.
4.    gt: HostName # HostName -> Bool; % Greater than.
5. var
6.    H, I: HostName;
7. eqn
8.    lt(A,A) = false;
9.    lt(A,B) = true;
10.   lt(A,C) = true;
11.   lt(B,A) = false;
12.   lt(B,B) = false;
13.   lt(B,C) = true;
14.   lt(C,A) = false;
15.   lt(C,B) = false;
16.   lt(C,C) = false;
17.    gt(H, I) = lt(I, H);

As can be seen in the code sample above, the sort HostName consists of three possibilities: A, B and C. Starting in line 8, each of these HostNames is compared to each other for the less than function lt. For the greater than function gt, only one line (line 17) is necessary, which says that H is greater than I if I is less than H.

The adhoc network protocol is just an arbitrary example in which a finite set is usable but one of the solutions should be usable in more protocols where finite sets are desirable but where they are not used yet.
11. Conclusions and recommendations

In the last chapter of this document, the master project is evaluated. Also conclusions are drawn and recommendations are made on how to use the specifications that were made throughout the process. I will first give a project evaluation and then make some recommendations for future work on the topic of finite set in mCRL2.

11.1. Project evaluation

At the end of the master project it can be concluded that the main objectives as described in the project description of section 2.1 were reached with exception of the third point of the project description, which was to implement an ordered rewriter. The reasons for omitting this point were given in section 2.2. The two most important parts of the project, implement finite sets in mCRL2 and implement a finite set as an ordered List were both completed. Since these points were the most important part of the project, I will elaborate some more on them in this section.

Implement a finite set as a List

This point was actually the second point of the project description, but it nevertheless was the point with which I started. I chose to implement a finite set as an ordered List of positive numbers. The positive numbers were chosen because that data type was already implemented in mCRL2 including functionality for an ordering on the positive numbers, which was necessary to create an ordered List.

At the end of the project it can be concluded that the implementation of a finite set as an ordered List is a solution that can be easily used in mCRL2 especially because the List structure itself is already available. After completion of the ordered List implementation, the choice was made to develop some other finite set implementations. This part will be reviewed in the next section.

Implement finite sets in mCRL

The first point of the project description said that an implementation of a finite set had to be implemented in mCRL2 such that it was accepted by the mCRL2 parser and type checker. This point was not the point with which I started, but this point was done after completion of the ordered List implementation.

When that implementation was completed, we considered some other possibilities of finite set implementations and the first one that was created was the finite set implemented as an AVL tree. This AVL tree implementation has some advantages over the ordered List implementation because insertion of elements and removal of elements from the set should go more efficiently in the AVL tree implementation. When testing and comparing these two implementations with the example process as given in section
9.2 it can be seen that after 3000 consecutive inserts into a finite set, the AVL tree gets the advantage over the ordered List implementation. This is due to the fact that insertion into an ordered List operates in $O(n)$ time and in an AVL tree in $O(\log(n))$ time plus the added overhead for rebalancing the tree after every insert.

A possible disadvantage of the AVL tree is that an AVL tree is not a unique representation of a finite set, i.e. the set represented by \{1, 2, 3, 4\} can be represented by two different AVL trees. This can decrease the performance of the AVL tree implementation because more states than necessary will be generated when creating a state space for a process using a finite set as AVL tree. Therefore we decided to create another tree implementation, but this time a tree that was unique, i.e. for every finite set there will be exactly one tree representation.

The tree implementation that was the result of this decision was the left-balanced tree. An overview of this tree is already given in section 6. The benefit of this tree is that the generated state space contains fewer states than when an AVL tree is used. The main disadvantage of this tree is that its performance is worse than that of both the AVL tree and the ordered List implementation under the JITTy c rewriting strategy of mCRL2. This is due to the fact that an insert into a left-balanced tree brings much overhead because it could be necessary that the whole tree structure has to be changed to maintain a correct left-balanced tree.

For this last reason, the performance of the unique left-balanced tree, we created a fourth and last implementation of a finite set as a unique tree. This unique tree is a tree that makes use of list conversion, i.e. upon inserting an element into such a tree, first this tree will be converted to an ordered List, then the element is inserted into this List and then this List is converted back to a tree. When using this implementation in the Simplified Sliding Window Protocol of section 10.1, it can be seen that the unique tree using List conversion gains the advantage over the other three implementations when a constant of size 9 is used.

So at the end it can be concluded that when doing $n$ consecutive inserts, then the ordered List implementation performs best, but when using a finite set within the Simplified Sliding Window Protocol, then the unique tree that uses List conversion performs best. So for adding the data type finite set to the mCRL2 syntax it is best to implement both the ordered List version and the unique tree using List conversion in such a way that the ordered List can also be used separate from the unique tree, i.e. one can specify a finite set that is an ordered List, but one can also specify a finite set that is a unique tree.
11.2. **Future work**

**Efficiency**

An important issue when using finite sets in mCRL2 is the efficiency and performance of its implementation. Both these factors are important because it is desirable to generate state spaces as fast as possible and as small as possible. So it could be possible that some functions from the different finite set solutions could be more efficient than they are at the moment. This means that it is recommended that before implementation of a finite set solution, the mCRL2 code for that particular solution is reviewed and maybe improved when thought necessary.

What already can be concluded, for example from the Simplified Sliding Window Protocol, is that a finite set solution as an ordered List or a unique tree using List conversion, is already much more efficient than not using a finite set. E.g. for the Boolean buffer in the original SSWP specification, a construction of $\text{Nat} \rightarrow \text{Bool}$ is used. When the value is constant 1, this solution is already a factor 300 slower than when the ordered List or the unique tree is used for the Boolean buffer.

**Toolset extension and implementation**

To be able to use finite sets in mCRL2 in an easy way, it is necessary to extend the toolset with syntax for specifying a finite set. This means that it has to be possible to specify a finite set in mCRL2 and that all finite set functionality is available without having to specify each function when needed. Specifying a finite set of the sort D could for example be done in the following way: \texttt{fset(D)}.

Another useful extension involves the function names of the finite set structure. At the moment these functions are called \texttt{union}, \texttt{intersection}, etc. but it would be useful that these names would be the same as they are used in the infinite set implementation that is already available in mCRL2. This means that \texttt{union} can be used as $\cup$ (in mCRL2 code specified as +) and \texttt{intersection} as $\cap$ (in mCRL2 code specified as *).

What also would be useful is the notation of the functionality. The notations of the finite set functionality as I created are all prefix functions. It would be easy to have functions that can be used as infix notations. This would work in combination with the naming of the functions, i.e. the union of two sets $S$ and $T$ would not be written as \texttt{union}(S, T), but as $S + T$, which is much shorter and easier to write.

For the finite set implementation to work correctly, an ordering has to be generated automatically. This can be done according to the scheme given in section 3.2. This ordering has to be generated when a sort is used that doesn’t have an ordering already. This means that when a user creates a mCRL2 specification and linearises this specification, this user is able to use all finite set functionality in the correct way, so all...
finite set functions are generated and the ordering functions for the sort are also generated when necessary.

When looking at the four finite set implementations, the ordered List, the AVL tree, the left-balanced tree and the unique tree that uses List conversion, it can be concluded that it would be best to implement both the ordered List implementation and the unique tree implementation. The ordered List implementation because it is also part of the unique tree using List conversion, but also because the ordered List implementation has a fairly good performance when compared to the left-balanced tree and the unique tree using list conversion.

The most important reason for implementing the unique tree using List conversion instead of implementing the AVL tree solution is that the unique tree is unique and the AVL tree is not. When only doing consecutive inserts as is done in section 9.2, the AVL tree performs better than the unique tree implementation, but when using an example protocol such as the Simplified Sliding Window Protocol, then the performance of the unique tree is better when the value of constant n in that protocol reaches 4.

The left-balanced tree is in my view not a serious candidate for implementation in the mCRL2 toolset because it brings a lot of overhead when inserting or deleting an element into or from the tree. This means that also the finite set functions of that implementation bring a lot of overhead because they also make use of inserts and deletes.

A reason for not implementing the AVL tree solution would be the fact that the AVL tree is not a unique tree, i.e. a finite set can be represented by more than one AVL tree and that is not desirable because of the state space explosion problem.

So at the end of my master project I would recommend to implement the ordered List solution in the mCRL2 toolset and also the unique tree that uses List conversion. Because the ordered List is mandatory to use the unique tree using List conversion, it should be implemented anyway, so the implementation of both solutions could be done in such a way that they can also be used next to each other. This means that there would be two finite set implementations available in the mCRL2 language.
Bibliography

[1] Adams, S., 
*Implementing Sets Efficiently in a Functional Language*, CSTR 92-10, University of Southampton.


*Process Algebra and mCRL2*, IPA Basic Course on Formal Methods.


[7] Mathijssen, A., 
*Data types for mCRL2*.


[9] van de Pol, J.C., 

[10] van de Pol, J.C., 
Appendix A: Finite set as ordered List

This appendix contains the mCRL2 specification of the finite set as ordered List as it is specified in the file ‘List.mcrl2’.

1. % Sort definition.
2. sort fsetList = List(Pos);

3. % Mappings.
4. map
5.    insert : Pos # fsetList -> fsetList;
6.    delete : Pos # fsetList -> fsetList;
7.    element_in : Pos # fsetList -> Bool;
8.    ordered : fsetList -> Bool;
9.    card : fsetList -> Nat;
10. subset : fsetList # fsetList -> Bool;
11. proper_subset : fsetList # fsetList -> Bool;
12. union : fsetList # fsetList -> fsetList;
13. difference : fsetList # fsetList -> fsetList;
14. intersection : fsetList # fsetList -> fsetList;
15. fsetList2Bag : fsetList -> Bag(Pos);
16. fsetList2Set : fsetList -> Set(Pos);

17. % Variables.
18. var
19.    p, q : Pos;
20.    s, t : fsetList;

21. % Equations.
22. eqn

23. % Insert.
24.    insert(p, []) = p |> [];
25.    insert(p, q |> s) =
26.        if(p < q,
27.            p |> (q |> s),
28.        %else
29.            if(p > q,
30.                q |> (insert(p, s)),
31.            %else
32.                q |> s));

33. % Delete.
34.    delete(p, []) = [];
35.    delete(p, q |> s) =
36.        if(p < q,
37.            q |> s,
38.        %else
39.            if(p > q,
40.                q |> delete(p, s),
41.            %else
42.                s));

43. % Element test.
44.    element_in(p, []) = false;
finite_sets_in_mCRL2

45. \text{element_in}(p, q \mid s) = \\
46. \quad \text{if}(p = q, \\
47. \quad \quad \text{true}, \\
48. \quad \quad \%\text{else} \\
49. \quad \quad \text{if}(p < q, \\
50. \quad \quad \quad \text{false}, \\
51. \quad \quad \quad \%\text{else} \\
52. \quad \quad \quad \text{element_in}(p, s))); \\
53. \%\text{Ordered}. \\
54. \quad \text{ordered}([]) = \text{true}; \\
55. \quad \text{ordered}(p \mid \) \quad = \text{true}; \\
56. \quad \text{ordered}(p \mid \quad (q \mid s)) = (p < q) \&\& \text{ordered}(q \mid s); \\
57. \%\text{Cardinality}. \\
58. \quad \text{card}([]) = 0; \\
59. \quad \text{card}(q \mid s) = \text{card}(s) + 1; \\
60. \%\text{Subset}. \\
61. \quad \text{subset}([], s) = \text{true}; \\
62. \quad \text{subset}(q \mid t, s) = \text{element_in}(q, s) \&\& \text{subset}(t, s); \\
63. \%\text{Proper subset}. \\
64. \quad \text{proper_subset}(s, t) = \text{subset}(s, t) \&\& s \neq t; \\
65. \%\text{Union}. \\
66. \quad \text{union}([], s) = s; \\
67. \quad \text{union}(q \mid t, s) = \text{insert}(q, \text{union}(t, s)); \\
68. \%\text{Difference}. \\
69. \quad \text{difference}(s, []) = s; \\
70. \quad \text{difference}([], s) = []; \\
71. \quad \text{difference}(q \mid t, s) = \\
72. \quad \quad \text{if}(!\text{element_in}(q, s), \\
73. \quad \quad \quad \text{difference}(t, s), \\
74. \quad \quad \%\text{else} \\
75. \quad \quad \quad q \mid \text{difference}(t, s)); \\
76. \%\text{Intersection}. \\
77. \quad \text{intersection}(s, []) = []; \\
78. \quad \text{intersection}(s, q \mid t) = \\
79. \quad \quad \text{if}(!\text{element_in}(q, s), \\
80. \quad \quad \quad q \mid \text{intersection}(s, t), \\
81. \quad \quad \%\text{else} \\
82. \quad \quad \quad \text{intersection}(s, t)); \\
83. \%\text{fsetList to Bag}. \\
84. \quad \text{fsetList2Bag}([]) = \{}; \\
85. \quad \text{fsetList2Bag}(q \mid s) = \{q:1\} + \text{fsetList2Bag}(s); \\
86. \%\text{fsetList to Set}. \\
87. \quad \text{fsetList2Set}([]) = \{}; \\
88. \quad \text{fsetList2Set}(q \mid s) = \{q\} + \text{fsetList2Set}(s);
Appendix B: Finite set as AVL tree

This appendix contains the mCRL2 specification of the finite set as AVL tree as it is specified in the file ‘AVL.mcrl2’.

1. % Sort definition.  
2. sort fsetAVL = struct empty | node(Pos, fsetAVL, fsetAVL, Nat);

3. % Mappings.  
4. map  
5.    insert : Pos # fsetAVL -> fsetAVL;  
6.    delete : Pos # fsetAVL -> fsetAVL;  
7.    element_in : Pos # fsetAVL -> Bool;  
8.    card : fsetAVL -> Nat;  
9.    subset : fsetAVL # fsetAVL -> Bool;  
10.   proper_subset : fsetAVL # fsetAVL -> Bool;  
11.   union : fsetAVL # fsetAVL -> fsetAVL;  
12.   difference : fsetAVL # fsetAVL -> fsetAVL;  
13.   intersection : fsetAVL # fsetAVL -> fsetAVL;  
14.   equal : fsetAVL # fsetAVL -> Bool;  
15.   inequal : fsetAVL # fsetAVL -> Bool;  
16.   fsetAVL2Bag : fsetAVL -> Bag(Pos);  
17.   fsetAVL2List : fsetAVL -> List(Pos);  
18.   fsetAVL2Set : fsetAVL -> Set(Pos);  
19.   rebalance : fsetAVL -> fsetAVL;  
20.   singleleft : fsetAVL -> fsetAVL;  
21.   doubleleft : fsetAVL -> fsetAVL;  
22.   singleright : fsetAVL -> fsetAVL;  
23.   doubleright : fsetAVL -> fsetAVL;  
24.   key : fsetAVL -> Pos;  
25.   left : fsetAVL -> fsetAVL;  
26.   right : fsetAVL -> fsetAVL;  
27.   minimum : fsetAVL -> Pos;  
28.   height : fsetAVL -> Nat;  
29.   get_height : fsetAVL -> Nat;  
30.   ordered : fsetAVL -> Bool;  
31.   allsmaller : fsetAVL # Pos -> Bool;  
32.   allbigger : fsetAVL # Pos -> Bool;

33. % Variables.  
34. var  
35.    p, q, q1, qr, ql : Pos;  
36.    l, l1, lr, ll, r, rl, rr, rl, s, t : fsetAVL;  
37.    h, h1, hr, hl : Nat;

38. % Equations.  
39. eqn  
40. % Insert.  
41.    insert(p, empty) = node(p, empty, empty, 1);  
42.    insert(p, node(q, l, r, h)) =  
43.        if((p == q),  
44.            node(q, l, r, h),  
45.        %else  
46.            if((p < q),
47.             rebalance(node(q, insert(p, l), r, h)),
48.          %else
49.             rebalance(node(q, l, insert(p, r), h))});

50. % Delete.
51.    delete(p, empty) = empty;
52.    delete(p, node(q, l, r, h)) =
53.       if(p < q,
54.          rebalance(node(q, delete(p, l), r, h)),
55.          %else
56.          if(p > q,
57.             rebalance(node(q, l, delete(p, r), h)),
58.          %else
59.             if(l == empty && r == empty,
60.                empty,
61.                if(l == empty && r != empty,
62.                   r,
63.                   %else
64.                     if(r == empty && l != empty,
65.                        l,
66.                        %else
67.                          rebalance(node(minimum(r), l,
68.                          delete(minimum(r), r), h))))));

69. % Element test.
70.    element_in(p, empty) = false;
71.    element_in(p, node(q, l, r, h)) =
72.       if(p == q,
73.          true,
74.          %else
75.          if(p < q,
76.             element_in(p, l),
77.          %else
78.             element_in(p, r));

79. % Cardinality.
80.    card(empty) = 0;
81.    card(node(q, l, r, h)) = (card(l) + card(r)) + 1;

82. % Subset.
83.    subset(empty, s) = true;
84.    subset(node(q, l, r, h), s) = element_in(q, s) && subset(l, s)
85.      && subset(r, s);

85. % Proper subset.
86.    proper_subset(s, t) = subset(s, t) && s != t;

87. % Union.
88.    union(empty, s) = s;
89.    union(node(q, l, r, h), s) = insert(q, union(l, union(r, s)));

90. % Difference
91.    difference(s, empty) = s;
92.    difference(empty, s) = empty;
93.    difference(node(q, l, r, h), node(q1, l1, r1, h1)) =
if(element_in(q, node(q1, l1, r1, h1)),
    difference(delete(q, node(q, l, r, h)),
      node(q1, l1, r1, h1)),
%else
    insert(q, difference(delete(q, node(q, l, r, h)),
      node(q1, l1, r1, h1)));

% Intersection
intersection(empty, s) = empty;
intersection(node(q, l, r, h), node(q1, l1, r1, h1)) =
    if(element_in(q, node(q1, l1, r1, h1)),
        insert(q, intersection(delete(q, node(q, l, r, h)),
            node(q1, l1, r1, h1))),
%else
        intersection(delete(q, node(q, l, r, h)),
            node(q1, l1, r1, h1)));

% Equal.
equal(s, t) = subset(s, t) && subset(t, s);

% Inequal.
inequal(s, t) = !(equal(s, t));

% fsetAVL2Bag.
fsetAVL2Bag(empty) = {};
fsetAVL2Bag(node(q, l, r, h)) = {q:1} + (fsetAVL2Bag(l) +
  fsetAVL2Bag(r));

% fsetAVL2List.
fsetAVL2List(empty) = [];
fsetAVL2List(node(q, l, r, h)) = fsetAVL2List(l) ++ (q |>
   fsetAVL2List(r));

% fsetAVL2Set.
fsetAVL2Set(empty) = {};
fsetAVL2Set(node(q, l, r, h)) = {q} + (fsetAVL2Set(l) +
  fsetAVL2Set(r));

% Rebalance.
rebalance(empty) = empty;
rebalance(node(q, l, r, h)) =
    if((get_height(r) - get_height(l)) > 1,
        if((get_height(right(r)) - get_height(left(r))) <= -1,
            doubleleft(node(q, l, r, h)),
        %else
            singleleft(node(q, l, r, h))),
    %else
    if((get_height(r) - get_height(l)) < -1,
        if((get_height(right(l)) - get_height(left(l))) >= 1,
            doubleright(node(q, l, r, h)),
        %else
            singleright(node(q, l, r, h))),
    %else
        node(q, l, r, max(get_height(l), get_height(r)) + 1)));
134. % Single left rotation.
135.    singleleft(node(q, l, node(qr, lr, rr, hr), h)) =
136.       node(qr, new_l, rr, max(get_height(new_l),
137.             get_height(rr)) + 1)
138.       whr
139.       new_l = node(q, l, lr, max(get_height(l), get_height(lr)) + 1)
140.       end;

141. % Double left rotation.
142.    doubleleft(node(q, l, r, h)) =
143.       singleleft(node(q, l, singleright(r), h));

144. % Single right rotation.
145.    singleright(node(q, node(ql, ll, rl, hl), r, h)) =
146.       node(ql, ll, new_r, max(get_height(ll),
147.             get_height(new_r)) + 1)
148.       whr
149.       new_r = node(q, rl, r, max(get_height(rl), get_height(r)) + 1)
150.       end;

151. % Double right rotation.
152.    doubleright(node(q, l, r, h)) =
153.       singleright(node(q, singleleft(l), r, h));

154. % Key.
155.    key(node(q, l, r, h)) = q;

156. % Left.
157.    left(empty) = empty;
158.    left(node(q, l, r, h)) = l;

159. % Right.
160.    right(empty) = empty;
161.    right(node(q, l, r, h)) = r;

162. % Minimum.
163.    minimum(node(q, l, r, h)) =
164.       if(l != empty,
165.           minimum(l),
166.           if(r != empty,
167.             minimum(r),
168.             q));

169. % Get height.
170.    get_height(empty) = 0;
171.    get_height(node(q, l, r, h)) = h;

172. % Height.
173.    height(empty) = 0;
174.    height(node(q, l, r, h)) = (max(height(l), height(r))) + 1;

175. % Ordered.
176.    ordered(empty) = true;
177.    ordered(node(q, l, r, h)) = ordered(l) && ordered(r) &&
178.       allsmaller(l, q) && allbigger(r, q);

179. % Allsmaller.
176. allsmaller(empty, p) = true;
177. allsmaller(node(q, l, r, h), p) = q < p && allsmaller(l, p) && allsmaller(r, p);

178. % Allbigger.
179. allbigger(empty, p) = true;
180. allbigger(node(q, l, r, h), p) = q > p && allbigger(l, p) && allbigger(r, p);
Appendix C: Finite set as left-balanced tree

This appendix contains the mCRL2 specification of the left-balanced tree.

1. % Sort definition.
2. sort fsetLB = struct empty | node(Pos, fsetLB, fsetLB, Nat);
3. % Mappings.
4. map
5. insert : Pos # fsetLB -> fsetLB;
6. delete : Pos # fsetLB -> fsetLB;
7. element_in : Pos # fsetLB -> Bool;
8. card : fsetLB -> Nat;
9. subset : fsetLB # fsetLB -> Bool;
10. proper_subset : fsetLB # fsetLB -> Bool;
11. union : fsetLB # fsetLB -> fsetLB;
12. difference : fsetLB # fsetLB -> fsetLB;
13. intersection : fsetLB # fsetLB -> fsetLB;
14. equal : fsetLB # fsetLB -> Bool;
15. unequal : fsetLB # fsetLB -> Bool;
16. fsetLB2Bag : fsetLB -> Bag(Pos);
17. fsetLB2List : fsetLB -> List(Pos);
18. fsetLB2Set : fsetLB -> Set(Pos);
19. key : fsetLB -> Pos;
20. left : fsetLB -> fsetLB;
21. right : fsetLB -> fsetLB;
22. minimum : fsetLB -> Pos;
23. maximum : fsetLB -> Pos;
24. get_height : fsetLB -> Nat;
25. % Variables.
26. var
27. p, q : Pos;
28. l, r, s, t : fsetLB;
29. h : Nat;
30. % Equations.
31. eqn
32. % Insert.
33. insert(p, empty) = node(p, empty, empty, 1);
34. insert(p, node(q, l, r, h)) =
35. if((p == q),
36. node(q, l, r, h),
37. %else
38. if((p < q) && l == empty,
39. node(q, insert(p, l), r, h + 1),
40. %else
41. if((p < q) && (cardl == maxnodes) &&
        (cardr == maxnodes),
42. node(q, insert(p, l), r, h + 1),
43. %else
44. if((p < q) && (cardl != maxnodes),
45. node(q, insert(p, l), r, h),
46. %else

if((p < q) && (cardl == maxnodes) &&
   (cardr != maxnodes),
   if(element_in(p, l),
      node(q, l, r, h),
      %else
      node(maxl, insert(p, delete(maxl, l)),
           insert(q, r), h)),
else
    %else
    if(p > q && l == empty,
       node(p, node(q, empty, empty, 1), r, h + 1),
    %else
    if(p > q) && (cardl == maxnodes) &&
       (cardr != maxnodes),
       node(q, 1, insert(p, r), h),
    %else
    if((p > q) && (cardl != maxnodes) &&
       (r != empty),
       if(element_in(p, r),
          node(minr, insert(q, delete(p, l)), delete(minr, r),
               h),
       %else
       node(q, l, r, h),
    %else
    node(minr, insert(q, l),
       insert(p, delete(minr, r)),
       h)),
    %else
      if(element_in(p, r),
         node(q, l, r, h),
      %else
       node(minr, insert(q, l),
       insert(p, delete(minr, r)),
       h + 1)))))))))

whr

cardl = card(l),
cardr = card(r),
maxnodes = exp(2, get_height(l)) - 1,
maxl = maximum(l),
minr = minimum(r)
end;

% Delete.
delete(p, empty) = empty;
delete(p, node(q, l, r, h)) =
   if((p < q) && (cardl == maxnodes) && (cardr == maxnodes) &&
      (r != empty),
      if(element_in(p, l),
         node(minr, insert(q, delete(p, l)), delete(minr, r),
               h),
      %else
      node(q, l, r, h),
   %else
   node(q, delete(p, l), r, max(get_height(delete(p, l)),
      get_height(r)) + 1),
    %else
    if((p < q) && (cardl != maxnodes),
      node(q, delete(p, l), r, max(get_height(delete(p, l)),
         get_height(r)) + 1),
    %else
    if((p < q) && (cardl == maxnodes) &&
       (cardr != maxnodes) && (r != empty),

if(element_in(p, l),
   node(minr, insert(q, delete(p, l)),
   delete(minr, r), h),
   %else
   node(q, l, r, h)),
   %else
   if((p < q) && (r == empty),
      node(q, delete(p, l), r, Int2Nat(h - 1)),
   %else
   if((p > q) && (cardl == maxnodes) &&
      (cardr == maxnodes),
      node(q, l, delete(p, r), h),
   %else
   if((p > q) && (cardl == maxnodes) &&
      (cardr != maxnodes) && (l != empty),
      if(element_in(p, r),
         node(maxl, delete(maxl, l),
         insert(q, delete(p, r)), h),
      %else
         node(q, l, r, h)),
   %else
   if((p > q) && (cardl != maxnodes) &&
      (l != empty),
      if(element_in(p, r),
         node(maxl, delete(maxl, l),
         insert(q, delete(p, r)),
         max(get_height(delete(maxl, l)),
         get_height(insert(q, delete(p, r))))) + 1),
   %else
   node(q, l, r, h)),
  %else
   if((p == q),
      if((l == empty) && (r == empty),
         empty,
      %else
      if((l == empty) && (r != empty),
         r,
      %else
      if((l != empty) &&
         (r == empty),
         l,
      %else
      node(minr, l, delete(minr, r), h))),
   %else
   node(q, l, r, h))))))))
whr

cardl = card(l),
cardr = card(r),
maxnodes = exp(2, get_height(l)) - 1,
maxl = maximum(l),
minr = minimum(r)
end;

% Element test.
element_in(p, empty) = false;
if(p == q,
   true,
   %else
   if(p < q,
      element_in(p, l),
      %else
      element_in(p, r)));

% Cardinality.
card(empty) = 0;
card(node(q, l, r, h)) = (card(l) + card(r)) + 1;

% Subset.
subset(empty, s) = true;
subset(node(q, l, r, h), s) = element_in(q, s) && subset(l, s) && subset(r, s);

% Proper subset.
proper_subset(s, t) = subset(s, t) && s != t;

% Union.
union(empty, s) = s;
union(node(q, l, r, h), s) = insert(q, union(l, union(r, s)));

% Difference.
difference(s, empty) = s;
difference(empty, s) = empty;
difference(node(q, l, r, h), s) =
   if(element_in(q, s),
      difference(delete(q, node(q, l, r, h)), s),
      %else
      insert(q, union(difference(l, s), difference(r, s))));

% Intersection.
intersection(empty, s) = empty;
intersection(node(q, l, r, h), s) =
   if(element_in(q, s),
      insert(q, union(intersection(l, s), intersection(r, s))),
      %else
      intersection(delete(q, node(q, l, r, h)), s));

% Equal.
equal(s, t) = subset(s, t) && subset(t, s);

% Inequal.
inequal(s, t) = !(equal(s, t));

% fsetLB2Bag.
fsetLB2Bag(empty) = {};
fsetLB2Bag(node(q, l, r, h)) = {q:1} + (fsetLB2Bag(l) + fsetLB2Bag(r));

% fsetLB2List.
fsetLB2List(empty) = [];

- 161 -
177. \[ \text{fsetLB2List}(\text{node}(q, l, r, h)) = \text{fsetLB2List}(l) ++ (q |> \text{fsetLB2List}(r)); \]

178. \% fsetLB2Set.
179. \[ \text{fsetLB2Set}(\text{empty}) = \{\}; \]
180. \[ \text{fsetLB2Set}(\text{node}(q, l, r, h)) = \{q\} + (\text{fsetLB2Set}(l) + \text{fsetLB2Set}(r)); \]

181. \% Key.
182. \[ \text{key}(\text{node}(q, l, r, h)) = q; \]

183. \% Left.
184. \[ \text{left}(\text{node}(q, l, r, h)) = l; \]

185. \% Right.
186. \[ \text{right}(\text{node}(q, l, r, h)) = r; \]

187. \% Minimum.
188. \[ \text{minimum}(\text{empty}) = \text{Nat2Pos}(0); \]
189. \[ \text{minimum}(\text{node}(q, l, r, h)) = \]
190. \[ \text{if}(l \neq \text{empty}, \]
191. \[ \text{minimum}(l), \]
192. \%else
193. \[ q); \]

194. \% Maximum.
195. \[ \text{maximum}(\text{node}(q, l, r, h)) = \]
196. \[ \text{if}(r \neq \text{empty}, \]
197. \[ \text{maximum}(r), \]
198. \%else
199. \[ q); \]

200. \% Get height.
201. \[ \text{get\_height}(\text{empty}) = 0; \]
202. \[ \text{get\_height}(\text{node}(q, l, r, h)) = h; \]
Appendix D: Finite set as unique tree

This section contains the mCRL2 specification for the finite set as a unique tree that makes use of conversion to a finite set as an ordered list. The definition of the sort fsetList that is used throughout this specification can be found in Appendix A.

1. % Sort definitions.
2. sort fsetUT = struct empty | node(Pos, fsetUT, fsetUT, Nat);

3. % Mappings.
4. map
5.   insert : Pos # fsetUT -> fsetUT;
6.   delete : Pos # fsetUT -> fsetUT;
7.   element_in : Pos # fsetUT -> Bool;
8.   card : fsetUT -> Nat;
9.   get_card : fsetUT -> Nat;
10. subset : fsetUT # fsetUT -> Bool;
11. proper_subset : fsetUT # fsetUT -> Bool;
12. union : fsetUT # fsetUT -> fsetUT;
13. difference : fsetUT # fsetUT -> fsetUT;
14. intersection : fsetUT # fsetUT -> fsetUT;
15. equal : fsetUT # fsetUT -> Bool;
16. inequal : fsetUT # fsetUT -> Bool;
17. fsetUT2Bag : fsetUT -> Bag(Pos);
18. fsetUT2fsetList : fsetList # fsetUT -> fsetList;
19. fsetUT2Set : fsetUT -> Set(Pos);

20. % Additional mappings for fsetList.
21.   fsetList2fsetUT : fsetList # Nat -> fsetUT;
22.   preorderleft : fsetList # Nat -> fsetList;
23.   preorderright : fsetList # Nat -> fsetList;

24. % Variables.
25. var
26.   p, q : Pos;
27.   l, r, s, t : fsetUT;
28.   c, n : Nat;
29.   base, i, j, k : fsetList;

30. % Equations.
31. eqn
32. % Insert.
33.   insert(p, empty) = node(p, empty, empty, 1);
34.   insert(p, node(q, l, r, c)) =
35.       if(element_in(p, node(q, l, r, c)),
36.           node(q, l, r, c),
37.           %else
38.           fsetList2fsetUT(insert(p,
39.             fsetUT2fsetList([],
40.             node(q, l, r, c))),
41.             get_card(node(q, l, r, c)) + 1));

39. % Delete.
40. delete(p, empty) = empty;
41. delete(p, node(q, l, r, c)) =
42.   if(element_in(p, node(q, l, r, c)),
43.     fsetList2fsetUT(delete(p,
44.         fsetUT2fsetList([],
45.         node(q, l, r, c)));
46. % Element test.
47. element_in(p, empty) = false;
48. element_in(p, node(q, l, r, c)) =
49.   if(p == q,
50.     true,
51.     if(p < q,
52.       if(p < q,
53.         element_in(p, l),
54.       else
55.         element_in(p, r)));
56. % Cardinality.
57. card(empty) = 0;
58. card(node(q, l, r, c)) = (card(l) + card(r)) + 1;
59. % Get cardinality.
60. get_card(node(q, l, r, c)) = c;
61. % Subset.
62. subset(empty, s) = true;
63. subset(node(q, l, r, c), s) = element_in(q, s) && subset(l, s)
64.     && subset(r, s);
65. % Proper subset.
66. proper_subset(s, t) = subset(s, t) && s != t;
67. % Union.
68. union(empty, s) = s;
69. union(node(q, l, r, c), s) = insert(q, union(l, union(r, s)));
70. % Difference.
71. difference(s, empty) = s;
72. difference(empty, s) = empty;
73. difference(node(q, l, r, c), s) =
74.   if(element_in(q, s),
75.     difference(delete(q, node(q, l, r, c)), s),
76.   else
77.     insert(q, union(difference(l, s), difference(r, s)));
78. % Intersection.
79. intersection(empty, s) = empty;
80. intersection(node(q, l, r, c), s) =
81.   if(element_in(q, s),
82.     insert(q, union(intersection(l, s), intersection(r, s))),
83.   else
84.     intersection(delete(q, node(q, l, r, c)), s));
% Equal.
84. equal(s, t) = subset(s, t) && subset(t, s);
85. % Inequal.
86. inequal(s, t) = !(equal(s, t));
87. % fsetUT2Bag.
88. fsetUT2Bag(empty) = {};
89. fsetUT2Bag(node(q, l, r, c)) = {q:1} + (fsetUT2Bag(l) + fsetUT2Bag(r));
90. % fsetUT2fsetList;
91. fsetUT2fsetList(base, empty) = base;
92. fsetUT2fsetList(base, node(q, l, r, c)) =
   fsetUT2fsetList(q |> fsetUT2fsetList(base, r), 1);
93. % fsetUT2Set.
94. fsetUT2Set(empty) = {};
95. fsetUT2Set(node(q, l, r, c)) = {q} + (fsetUT2Set(l) + fsetUT2Set(r));
96. % Additional fsetList equations.
97. % fsetList to fsetUT.
98. n < 1 -> fsetList2fsetUT(i, n) = empty;
99. n > 0 && n < 2 -> fsetList2fsetUT(i, n) =
   node(head(i), empty, empty, 1);
100. n >= 2 -> fsetList2fsetUT(i, n) =
   node(i.(middle),
      fsetList2fsetUT(preorderleft(i, middle), middle),
      fsetList2fsetUT(preorderright(i, middle + 1),
      Int2Nat((n - middle) - 1)),
      n);
101. whr
102. middle = n div 2
103. end;
104. % Preorderleft
105. n >= 1 -> preorderleft(i, n) =
   head(i) |> preorderleft(tail(i), Int2Nat(n - 1));
106. % Preorderright
107. n == 0 -> preorderright(i, n) = i;
108. n > 0 -> preorderright(i, n) =
   preorderright(tail(i), Int2Nat(n - 1));
Appendix E: Original project description

Implementation of finite sets in mCRL2

Introduction

Currently, in mCRL2, built-in sets are functions and this may not be the best solution for much applications. The goal is now to do research after the implementation of finite sets in mCRL2. To achieve this, it is possible that an extension of the language has to be developed in which finite sets appear as a subset of infinite sets. It is also possible that an extension of the language has to be developed in which each type gets its own total ordering. This could be necessary for the use of ordered lists.

The duration of this project will be approximately 8 months; the following phases will be divided over these 8 months (each phase is divided in a set of goals):

Phase 1 Implementation of finite sets

Goals:
- Explore mCRL2 as it is currently, the existing mCRL2 language has to be explored to get a good overview of the language and its possibilities.
- Do research after implementation of finite sets, a strategy has to be found with which finite sets can be implemented as a subset of infinite sets, this could be done by creating an extension of the language.
- Discuss the possible solutions with the mCRL2 group for the acceptance of the chosen language extensions. This can be done by showing examples to illustrate the possibilities.
- Implement the finite sets in mCRL2 in the way that was chosen. This can be either as a subset of infinite set or as an extension of the language.
- Test the implementation of finite sets.

Risks:
- Acceptance of the language extensions by the mCRL2 group, could take some time to discuss these extensions before acceptance.
- Based on the discussions with the mCRL2 group, it could be possible that some notations have to be adapted.
- Test results are not satisfying and some work on the implementation has to be redone.

Deliverable:
An implementation of finite sets that is accepted by the mCRL2 parser and type checker.

Estimated time:
2 months.
Phase 2 Implementation of finite sets as lists

Goals:
- Do research after the necessity of creating an extension of the language in which each type gets its own total ordering, this could be necessary for the use of ordered lists.
- Do research after the possibilities to implement finite sets as lists.
- Make an implementation of the solution found for representing finite sets as lists.
- Start with writing master thesis.

Deliverable:
mCRL2 code for finite sets as lists.

Estimated time:
2 months.

Phase 3 Implementation of ordered rewriting

Goal:
- Do research after a strategy for ordered rewriting, this is necessary to transform the finite sets as lists, as implemented in phase 2, to ordered lists.
- Implement a solution for the use of ordered lists with the help of ordered rewriting. When necessary, implement an extension of the language in which each type gets its own total ordering.
- Test the implementation of the ordered rewriter and the ordered lists.
- Continue working on master thesis.

Risks:
- Test results are not satisfying and some work on the implementation has to be redone.

Deliverable:
An implemented ordered rewriter for rewriting finite sets as lists to ordered lists.

Estimated time:
3 months.

Phase 4 Case study

Goals:
- Test the implementation of the finite sets and test mCRL2 on existing security protocols that are currently available in mCRL2 (and in μCRL) and obtain satisfying test results.
- Complete master thesis and graduation presentation.
**Risks:**
- Test results are not satisfying and work of the previous phases has to be reviewed and maybe (partly) redone.

**Deliverable:**

**Estimated time:**
1 month.