

iiiiii HEAD	=====	LLLLLLL
d795feb6625577e42c080e911fbf2815a82c6857		

# Hogebomen

Lisette de Schipper (s1396250) en Micky Faas (s1407937)

## Abstract

Blablabla

## 1 Inleiding

AVL-bomen, splay-bomen en treaps zijn klassieke datastructuren die ingezet worden om een verzameling gegevens te faciliteren. Het zijn zelfbalancerende binaire zoekbomen die elk een vorm van ruimte en/of tijd-efficiëntie aanbieden. Er worden experimenten verricht om de prestatie van deze zelf-balancerende zoekbomen te vergelijken aan de hand van ophaaltijd van data, mate van herstructurering en het verwijderen van knopen. Ook wordt de prestatie van deze zoekbomen uitgezet tegen de ongebalanceerde tegenhanger, de binaire zoekboom.

## 2 Werkwijze

De vier bomen zijn conceptueel eenvoudig en relatief makkelijk te implementeren. Voor alle vier de bomen wordt dezelfde zoekmethode gebruikt. Deze is in het slechtste geval  $O(\log n)$ .

### 2.1 Implementatie binaire zoekboom

De binaire zoekboom (BST) vormt de basis voor alle zogeheten *zelf-organiserende bomen*, zoals de AVL- of SplayTree. Aan de grondslag van de BST ligt de *binaire-zoekboom-eigenschap*, die zorgt dat de boom op de “gretige” manier kan worden doorzocht in plaats van een *exhaustive search*. Hierdoor is het mogelijk om een knoop in hooguit  $O(\log(n))$  stappen te vinden. Kort samengevat houdt de bst-eigenschap het volgende in:

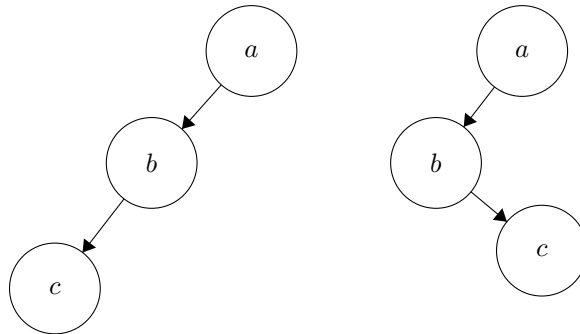
- Linker-kindknopen hebben altijd een kleinere waarde dan hun ouder, rechter-kindknopen altijd een grotere.
- Bij een MIN-boom is dit omgekeerd. Onze implementatie is enkel een MAX-boom.
- Toevoegen kan zonder verwisselen worden uitgevoerd (in tegenstelling tot bijv. een heap).
- Voor verwijderen of vervangen moet afhankelijk van de plaats van de knoop wel worden verwisseld.

*Nu ben ik moe*

## 2.2 Implementatie AVL-bomen

Knopen van een AVL-boom hebben een *balansfactor*, die altijd -1, 0 of 1 moet zijn. In deze implementatie is de balansfactor de hoogte van de rechtersubboom min de hoogte van de linkersubboom. Dit houdt dus in dat de hoogte van de linkersubboom van de wortel met maar 1 knoop kan verschillen van de hoogte van de rechtersubboom van de wortel. Het moment dat de balansfactor van een knoop minder dan -1 of meer dan 1 wordt, moet de boom geherstructureerd worden, om deze eigenschap te herstellen.

Om de balansfactor voor elke knoop te berekenen, houdt elke knoop zijn eigen hoogte bij. De balansfactor van een knoop wordt hersteld door rotaties. De richting en de hoeveelheid van de rotaties hangt af van de vorm van de betreffende (sub)boom. De volgende twee vormen en hun spiegelbeelden kunnen voorkomen bij het verwijderen of toevoegen van een knoop:



In het eerste geval moet de wortel naar rechts worden geroteerd. In het tweede geval moeten we eerst naar de staat van de eerste subboom komen, door b naar links te roteren. Voor de spiegelbeelden van deze twee vormen geldt hetzelfde alleen in spiegelbeeld.

In deze implementatie van een AVL-boom bedraagt het toevoegen van een knoop in het ergste geval  $O(\log n)$  tijd, waarbij  $n$  staat voor de hoogte van de boom. Eerst moet er gekeken worden of de data niet al in de boom voorkomt ( $O(\log n)$ ) en vervolgens moet de boom op basis van de toevoeging geherstructureerd worden. Dit laatste is in het ergste geval  $O(\log n)$ , omdat dan de gehele boom tot de wortel moeten worden nagelopen.

De complexiteitsgraad van het verwijderen van een knoop is gelijk aan die van het toevoegen van een knoop. In deze implementatie zoeken we in de rechtersubboom het kleinste kind en vervangen we de te verwijderen knoop met deze knoop. Dit heeft een duur van  $O(\log n)$ . Als hij geen rechtersubboom heeft, wordt de node weggegooid en wordt zijn linkersubboom de nieuwe boom.

## 2.3 Implementatie Splay-bomen

De Splay-boom is een simpele binaire zoekboom die zichzelf herorganiseert na elke operatie, ook na operaties die alleen lezen, zoals `find()`. Deze herorgan-

isatiestap heet “splay” (vandaar de naam) en heeft ten doel de laatst aangesproken knoop bovenaan te zetten. Dit wordt dus de wortel. Hieronder is het gedrag kort samengevat:

- Bij zoeken wordt de gevonden knoop de wortel, mits er een zoekresultaat is.
- Bij toevoegen wordt de toegevoegde knoop de wortel
- Bij vervangen wordt de vervangen knoop de wortel
- Bij verwijderen wordt de te verwijderen knoop eerst de wortel, dan wordt deze verwijderd.

Het idee achter dit gedrag is, dat vaak gebruikte knopen hoger in de boom terechtkomen en daarom sneller toegankelijk zijn voor volgende operaties. De splay-operatie zorgt er bovendien voor dat knoop die dicht in de buurt van de *gesplayde* knoop zitten, ook hoger in de boom worden geplaatst. Dit effect ontstaat doordat *splay* eigenlijk een serie boom rotaties is. Als men deze rotaties consequent uitvoert blijft bovendien de binary-zoekboom-eigenschap behouden.

### 2.3.1 Splay

De splay-operatie bestaat uit drie operaties en hun spiegelbeelden. We gaan uit van een knoop  $n$ , zijn ouderknoop  $p$  en diens ouderknoop  $g$ . Welke operatie wordt uitgevoerd is afhankelijk van het feit of  $n$  en  $p$  linker- of rechterkind zijn. We definiëren:

- De *Zig* stap. Als  $n$  linkerkind is van  $p$  en  $p$  de wortel is, doen we een rotate-right op  $p$ .
- Het spiegelbeeld van *Zig* is *Zag*.
- De *Zig-Zig* stap. Als  $n$  linkerkind is van  $p$  en  $p$  linkerkind is van  $g$ , doen we eerst een rotate-right op  $g$  en dan een rotate-right op  $p$ .
- Het spiegelbeeld van *Zig-Zig* is *Zag-Zag*
- De *Zig-Zag* stap. Als  $n$  rechterkind is van  $p$  en  $p$  linkerkind is van  $g$ , doen we eerst een rotate-left op  $p$  en dan een rotate-right op  $g$ .
- De omgekeerde versie heet *Zag-Zig*

Onze implementatie splayt op `insert()`, `replace()`, `remove()` en `find()`. De gebruiker kan eventueel zelf de splay-operatie aanroepen na andere operaties dmv de functie `splay()`.

## 2.4 Implementatie Treaps

Treap lijkt in veel opzichten op een AVL-boom. De balansfactor per knoop heeft echter plaats gemaakt voor een prioriteit per knoop. Deze prioriteit wordt bij het toevoegen van een knoop willekeurig bepaald. De complexiteit voor het toevoegen en verwijderen van een knoop is hetzelfde als bij de AVL-boom.

Bij het toevoegen van een knoop moet er nog steeds omhoog gelopen worden in de boom, totdat de prioriteit van de toegevoegde knoop kleiner is dan de prioriteit van de ouder. Als dit niet het geval is, blijft de toegevoegde knoop omhoog roteren. In het ergste geval kan het dus weer zo zijn dat we tot de wortel door moeten blijven lopen.

Bij het verwijderen van een knoop blijven we de betreffende knoop roteren naar het kind met de grootste prioriteit. Uiteindelijk belanden we dan in de situatie dat de knoop maar een of geen kinderen heeft. In het eerste geval verwijderen we de knoop en plakken zijn subboom terug aan de boom op zijn plek en in het tweede geval verwijderen we de knoop. In het slechtste geval duurt dit dus ook  $O(\log n)$  tijd.

### 3 Onderzoek

Een praktisch voorbeeld van binair zoeken in een grote boom is de spellingscontrole. Een spellingscontrole moet zeer snel voor een groot aantal strings kunnen bepalen of deze wel of niet tot de taal behoren. Aangezien er honderduizenden woorden in een taal zitten, is lineair zoeken geen optie. Voor onze experimenten hebben wij dit als uitgangspunt genomen en hieronder zullen we kort de experimenten toelichten die wij hebben uitgevoerd. In het volgende hoofdstuk staan vervolgens de resultaten beschreven.

#### 3.1 Hooiberg

“Hooiberg” is de naam van het testprogramma dat we hebben geschreven speciaal ten behoeven van onze experimenten. Het is een klein console programma dat woorden uit een bestand omzet tot een boom in het geheugen. Deze boom kan vervolgens worden doorzocht met de input uit een ander bestand: de “naalden”. De syntax is als volgt:

```
hooiberg type hooiberg.txt naalden.txt [treap-random-range]
```

Hierbij is **type** één van **bst**, **avl**, **splay**, **treap**, het eerste bestand bevat de invoer voor de boom, het tweede bestand een verzameling strings als zoekopdracht en de vierde parameters is voorbehouden voor het type **treap**. De bestanden kunnen woorden of zinnen bevatten, gescheiden door regeleinden. De binaire bomen gebruiken lexicografische sortering die wordt geleverd door de operatoren **<** en **>** van de klasse `std::string`. Tijdens het zoeken wordt een exacte match gebruikt (case-sensitive, non-locale-aware).

#### 3.2 Onderzoeks(deel)vragen

Met onze experimenten hebben we gepoogd een aantal eenvoudige vragen te beantwoorden over het gebruik van de verschillende binaire en zelf-organiserende bomen, te weten:

- Hoeveel meer rekenkracht kost het om grote datasets in te voegen in zelf-organiserende bomen tov binaire bomen?

- Levert een zelf-organiserende boom betere zoekprestaties en onder welke opstandigheden?
- Hoeveel extra geheugen kost een SOT?
- Wat is de invloed van de random-factor bij de Treap?

### 3.3 Meetmethoden

Om de bovenstaande vragen te toetsen, hebben we een aantal meetmethoden bedacht.

- Rekenkracht hebben we gemeten in milliseconden tussen aanvang en termineren van een berekening. We hebben de delta's berekend rond de relevante code blokken dmv de C++11 `chrono` klassen in de Standard Template Library. Alle test zijn volledig sequentieel en single-threaded uitgevoerd. Deze resultaten zijn representatie voor één bepaald systeem, vandaar dat we aantal % 'meer rekenkracht' als eenheid gebruiken.
- Zoekprestatie hebben we zowel met rekenkracht als zoekdiepte gemeten. De zoekdiepte is het aantal stappen dat vanaf de wortel moet worden gemaakt om bij de gewenste knoop te komen. We hebben hierbij naar het totaal aantal stappen gekeken en naar de gemiddelde zoekdiepte.
- Geheugen hebben we gemeten met de `valgrind` memory profiler. Dit programma wordt gebruikt voor het opsporen van geheugen lekken en houdt het aantal allocaties op de heap bij. Dit is representatie voor het aantal gealloceerde nodes. Aangezien hooiberg nauwelijks een eigen geheugen-voetafdruk heeft, zijn deze waarden representatief.

### 3.4 Input data

Voor ons experiment hebben we een taalbestand gebruikt van OpenTaal.org met meer dan 164.000 woorden. Dit is een relatief klein taalbestand, maar voldoende om verschillen te kunnen zien. We hebben een aantal testcondities gebruikt:

- Voor het inladen een wel of niet alfabetisch gesorteerd taalbestand gebruiken.
- Als zoekdocument hebben we een gedicht met 62 woorden gebruikt. Er zitten een aantal dubbele woorden in alsook een aantal woorden die niet in de woordenlijst voorkomen (werkwoordsvervoegingen).
- We hebben ook een conditie waarbij we alle woorden gezocht hebben, zowel in dezelfde, als in een andere volgorde dan dat ze zijn ingevoerd.
- We hebben één conditie waarbij we de random-range van de Treap hebben gevarieerd.

### 3.5 Hypothesen

- De binary search tree zal vermoedelijk het snelst nieuwe data toevoegen. De splay tree heeft veel ingewikkelde rotatie bij een insert, dus deze zal het traagst zijn.
- Bij het gedicht zal de splay boom waarschijnlijk het snelst zijn omdat deze optimaliseert voor herhalingen.
- ...
- De bomen die een aparte node-klasse gebruiken (avl en treap) gebruiken het meeste geheugen.
- De meest efficiënte randomfactor is afhankelijk van de grootte van de boom die geïmplementeerd gaat worden. Bij een kleine boom volstaat een kleine randomfactor, bij een grote boom volstaat een grote randomfactor.

## 4 Resultaten

## 5 Conclusies

## 6 Appendix

### 6.1 main.cc

```
1  /**
2   * main.cc:
3   *
4   * @author Micky Faas (s1407937)
5   * @author Lisette de Schipper (s1396250)
6   * @file main.cc
7   * @date 26-10-2014
8   */
9
10 #include <iostream>
11 #include "BinarySearchTree.h"
12 #include "Tree.h"
13 #include "AVLTree.h"
14 #include "SplayTree.h"
15 #include "Treap.h"
16 #include <string>
17
18 using namespace std;
19
20 // Makkelijk voor debuggen, moet nog beter
21 template<class T> void printTree( Tree<T> tree, int rows ) {
22     typename Tree<T>::nodelist list = tree.row( 0 );
23     int row = 0;
24     while( !list.empty( ) && row < rows ) {
25         string offset;
26         for( int i = 0; i < ( 1 << ( rows - row ) ) - 1 ; ++i )
```

```

27         offset += ' ';
28
29
30         for( auto it =list.begin( ); it != list.end( ); ++it ) {
31             if( *it )
32                 cout << offset << (*it)->info() << " " << offset;
33             else
34                 cout << offset << ". " << offset;
35         }
36         cout << endl;
37         row++;
38         list =tree.row( row );
39     }
40 }
41
42 int main ( int argc, char **argv ) {
43
44     /* BST hieronder */
45
46     cout << "BST:" << endl;
47     BinarySearchTree<int> bst;
48
49     /* auto root =bst.pushBack( 10 );
50     bst.pushBack( 5 );
51     bst.pushBack( 15 );
52
53     bst.pushBack( 25 );
54     bst.pushBack( 1 );
55     bst.pushBack( -1 );
56     bst.pushBack( 11 );
57     bst.pushBack( 12 );*/
58
59     Tree<int>* bstP =&bst; // Dit werkt gewoon :-)
60
61     auto root =bstP->pushBack( 10 );
62     bstP->pushBack( 5 );
63     bstP->pushBack( 15 );
64
65     bstP->pushBack( 25 );
66     bstP->pushBack( 1 );
67     bstP->pushBack( -1 );
68     bstP->pushBack( 11 );
69     bstP->pushBack( 12 );
70
71     //printTree<int>( bst, 5 );
72
73
74     //bst.remove( bst.find( 0, 15 ) );
75     //bst.replace( -2, bst.find( 0, 5 ) );
76
77
78     printTree<int>( bst, 5 );
79
80     bst.remove( root );

```



```

81
82
83     printTree<int>( bst, 5 );
84
85     /* Splay Trees hieronder */
86
87     cout << "Splay Boom:" << endl;
88     SplayTree<int> splay;
89
90     splay.pushBack( 10 );
91     auto a =splay.pushBack( 5 );
92     splay.pushBack( 15 );
93
94     splay.pushBack( 25 );
95     auto b =splay.pushBack( 1 );
96     splay.pushBack( -1 );
97     auto c =splay.pushBack( 11 );
98     splay.pushBack( 12 );
99
100    //printTree<int>( splay, 5 );
101
102    //a->swapWith( b );
103    //splay.remove( splay.find( 0, 15 ) );
104    //splay.replace( -2, splay.find( 0, 5 ) );
105
106
107    printTree<int>( splay, 5 );
108
109    //splay.remove( root );
110
111    splay.splay( c );
112
113    printTree<int>( splay, 5 );
114
115    // Test AVLTree //
116
117    AVLTree<char> test;
118    test.insert( 'a' );
119    auto d =test.insert( 'b' );
120    test.insert( 'c' );
121    test.insert( 'd' );
122    test.insert( 'e' );
123    test.insert( 'f' );
124    test.insert( 'g' );
125    cout << "AVL Boompje:" << endl;
126    printTree<char>( test, 5 );
127    cout << d->info( ) << " verwijderen: " << endl;
128    test.remove( d );
129    printTree<char>( test, 5 );
130
131    // Test Treap //
132
133    cout << "Treap" << endl;
134

```

```

135     Treap<int> testTreap(5);
136     testTreap.insert(2);
137     testTreap.insert(3);
138     auto e =testTreap.insert(4);
139     testTreap.insert(5);
140     printTree<int>( testTreap, 5 );
141     testTreap.remove(e);
142     printTree<int>( testTreap, 5 );
143
144     return 0;
145 }

```

## 6.2 hooiberg.cc

```

1  /**
2   * hooiberg.cc:
3   *
4   * @author Micky Faas (s1407937)
5   * @author Lisette de Schipper (s1396250)
6   * @file helehogebomen.cc
7   * @date 10-12-2014
8   */
9
10 #include "BinarySearchTree.h"
11 #include "Tree.h"
12 #include "AVLTree.h"
13 #include "SplayTree.h"
14 #include "Treap.h"
15
16 #include <iostream>
17 #include <string>
18 #include <fstream>
19 #include <vector>
20 #include <chrono>
21
22 // Only works on *nix operating systems
23 // Needed for precision timing
24 #include <sys/time.h>
25
26 using namespace std;
27
28 // Makkelijk voor debuggen, moet nog beter
29 template<class T> void printTree( Tree<T> tree, int rows ) {
30     typename Tree<T>::nodelist list =tree.row( 0 );
31     int row =0;
32     while( !list.empty( ) && row < rows ) {
33         string offset;
34         for( int i =0; i < ( 1 << (rows - row) ) - 1 ; ++i )
35             offset += ' ';
36
37
38         for( auto it =list.begin( ); it != list.end( ); ++it ) {
39             if( *it )
40                 cout << offset << (*it)->info() << " " << offset;

```

```

41         else
42             cout << offset << ". " << offset;
43     }
44     cout << endl;
45     row++;
46     list = tree.row( row );
47 }
48 }
49
50 int printUsage( const char* prog ) {
51
52     std::cout << "Reads an input file and searches it for a set of strings\n\n"
53         << "Usage: " << prog << " [type] [haystack] [needles]\n"
54         << "\t[type]\t\tTree type to use. One of 'splay', 'avl', 'treap', 'bst'\n"
55         << "\t[haystack]\tInput file, delimited by newlines\n"
56         << "\t[needles]\tFile containing sets of strings to search for, delimited by"
57         << "\t[random]\tOptimal customization of Treap\n"
58         << std::endl;
59     return 0;
60 }
61
62 bool extractNeedles( std::vector<string> &list, std::ifstream &file ) {
63     string needle;
64     while( !file.eof( ) ) {
65         std::getline( file, needle );
66         if( needle.size( ) )
67             list.push_back( needle );
68     }
69     return true;
70 }
71
72 bool fillTree( BinarySearchTree<string>* tree, std::ifstream &file ) {
73     string word;
74     while( !file.eof( ) ) {
75         std::getline( file, word );
76         if( word.size( ) )
77             tree->pushBack( word );
78     }
79     return true;
80 }
81
82 void findAll( std::vector<string> &list, BinarySearchTree<string>* tree ) {
83     int steps = 0, found = 0, notfound = 0;
84     for( auto needle : list ) {
85         if( tree->find( 0, needle ) ) {
86             found++;
87             steps += tree->lastSearchStepCount( );
88             if( found < 51 )
89                 std::cout << "Found " << needle << '\n'
90                 << " in " << tree->lastSearchStepCount( ) << " steps." << std::endl;
91         }
92         else if( ++notfound < 51 )
93             std::cout << "Didn't find " << needle << '\n' << std::endl;
94     }

```

```

95     if( found > 50 )
96         std::cout << found - 50 << " more results not shown here." << std::endl;
97     if( found )
98         cout << "Total search depth:           " << steps << endl
99             << "Number of matches:           " << found << endl
100            << "Number of misses:           " << notfound << endl
101            << "Average search depth (hits): " << steps/found << endl;
102 }
103
104 int main ( int argc, char **argv ) {
105
106     enum MODE { NONE =0, BST, AVL, SPLAY, TREAP };
107     int mode =NONE;
108
109     if( argc < 4 )
110         return printUsage( argv[0] );
111
112     if( std::string( argv[1] ) == "bst" )
113         mode =BST;
114     else if( std::string( argv[1] ) == "avl" )
115         mode =AVL;
116     else if( std::string( argv[1] ) == "treap" )
117         mode =TREAP;
118     if( std::string( argv[1] ) == "splay" )
119         mode =SPLAY;
120
121     if( !mode )
122         return printUsage( argv[0] );
123
124     std::ifstream fhaystack( argv[2] );
125     if( !fhaystack.good( ) ) {
126         std::cerr << "Could not open " << argv[2] << std::endl;
127         return -1;
128     }
129
130     std::ifstream fneedles( argv[3] );
131     if( !fneedles.good( ) ) {
132         std::cerr << "Could not open " << argv[3] << std::endl;
133         return -1;
134     }
135
136     if( argc == 5 ) {
137         if( argv[4] && mode != TREAP ) {
138             std::cerr << "This variable should only be set for Treaps." << std::endl;
139             return -1;
140         }
141         else if( atoi(argv[4]) <= 0 ) {
142             std::cerr << "This variable should only be an integer "
143                 << " greater than 0." << std::endl;
144             return -1;
145         }
146     }
147
148     std::vector<string> needles;

```

```

149     if( !extractNeedles( needles, fneedles ) ) {
150         cerr << "Could not read a set of strings to search for." << endl;
151         return -1;
152     }
153
154     BinarySearchTree<string> *tree;
155     switch( mode ) {
156         case BST:
157             tree = new BinarySearchTree<string>();
158             break;
159         case AVL:
160             tree = new AVLTree<string>();
161             break;
162         case SPLAY:
163             tree = new SplayTree<string>();
164             break;
165         case TREAP:
166             tree = new Treap<string>(atoi(argv[4]));
167             break;
168     }
169
170
171     // Define a start point to time measurement
172     auto start = std::chrono::system_clock::now();
173
174
175     if( !fillTree( tree, fhaystack ) ) {
176         cerr << "Could not read the haystack." << endl;
177         return -1;
178     }
179
180     // Determine the duration of the code block
181     auto duration =std::chrono::duration_cast<std::chrono::milliseconds>
182         (std::chrono::system_clock::now() - start);
183
184     cout << "Filled the binary search tree in " << duration.count() << "ms" << endl;
185
186     start = std::chrono::system_clock::now();
187     findAll( needles, tree );
188     duration =std::chrono::duration_cast<std::chrono::milliseconds>
189         (std::chrono::system_clock::now() - start);
190
191     cout << "Searched the haystack in " << duration.count() << "ms" << endl;
192
193     // Test pre-order
194     //for( auto word : *tree ) {
195     //    cout << word << '\n';
196     //}
197
198     fhaystack.close( );
199     fneedles.close( );
200     delete tree;
201
202     return 0;

```

203 }

## 6.3 Tree.h

```
1  /**
2   * Tree:
3   *
4   * @author Micky Faas (s1407937)
5   * @author Lisette de Schipper (s1396250)
6   * @file tree.h
7   * @date 26-10-2014
8   */
9
10 #ifndef TREE_H
11 #define TREE_H
12 #include "TreeNodeIterator.h"
13 #include <assert.h>
14 #include <list>
15 #include <map>
16
17 using namespace std;
18
19 template <class INFO_T> class SplayTree;
20
21 template <class INFO_T> class Tree
22 {
23     public:
24         enum ReplaceBehavoir {
25             DELETE_EXISTING,
26             ABORT_ON_EXISTING,
27             MOVE_EXISTING
28         };
29
30         typedef TreeNode<INFO_T> node_t;
31         typedef TreeNodeIterator<INFO_T> iterator;
32         typedef TreeNodeIterator_in<INFO_T> iterator_in;
33         typedef TreeNodeIterator_pre<INFO_T> iterator_pre;
34         typedef TreeNodeIterator_post<INFO_T> iterator_post;
35         typedef list<node_t*> nodelist;
36
37         /**
38          * @function Tree( )
39          * @abstract Constructor of an empty tree
40          */
41         Tree( )
42             : m_root( 0 ) {
43         }
44
45         /**
46          * @function Tree( )
47          * @abstract Copy-constructor of a tree. The new tree contains the nodes
48          *          from the tree given in the parameter (deep copy)
49          * @param tree, a tree
50          */
```

```

51     Tree( const Tree<INFO_T>& tree )
52         : m_root( 0 ) {
53         *this =tree;
54     }
55
56     /**
57     * @function   ~Tree( )
58     * @abstract   Destructor of a tree. Timber.
59     **/
60     ~Tree( ) {
61         clear( );
62     }
63
64     /**
65     * @function   begin_pre( )
66     * @abstract   begin point for pre-order iteration
67     * @return     iterator_pre containing the beginning of the tree in
68     *             pre-order
69     **/
70     iterator_pre begin_pre( ) {
71         // Pre-order traversal starts at the root
72         return iterator_pre( m_root );
73     }
74
75     /**
76     * @function   begin( )
77     * @abstract   begin point for a pre-order iteration
78     * @return     containing the beginning of the pre-Order iteration
79     **/
80     iterator_pre begin( ) {
81         return begin_pre( );
82     }
83
84     /**
85     * @function   end( )
86     * @abstract   end point for a pre-order iteration
87     * @return     the end of the pre-order iteration
88     **/
89     iterator_pre end( ) {
90         return iterator_pre( (node_t*)0 );
91     }
92
93     /**
94     * @function   end_pre( )
95     * @abstract   end point for pre-order iteration
96     * @return     iterator_pre containing the end of the tree in pre-order
97     **/
98     iterator_pre end_pre( ) {
99         return iterator_pre( (node_t*)0 );
100    }
101
102    /**
103    * @function   begin_in( )
104    * @abstract   begin point for in-order iteration

```

```

105      * @return      iterator_in containing the beginning of the tree in
106      *              in-order
107      **/
108      iterator_in begin_in( ) {
109          if( !m_root )
110              return end_in( );
111          node_t *n =m_root;
112          while( n->leftChild( ) )
113              n =n->leftChild( );
114          return iterator_in( n );
115      }
116
117      /**
118      * @function      end_in( )
119      * @abstract      end point for in-order iteration
120      * @return      iterator_in containing the end of the tree in in-order
121      **/
122      iterator_in end_in( ) {
123          return iterator_in( (node_t*)0 );
124      }
125
126      /**
127      * @function      begin_post( )
128      * @abstract      begin point for post-order iteration
129      * @return      iterator_post containing the beginning of the tree in
130      *              post-order
131      **/
132      iterator_post begin_post( ) {
133          if( !m_root )
134              return end_post( );
135          node_t *n =m_root;
136          while( n->leftChild( ) )
137              n =n->leftChild( );
138          return iterator_post( n );
139      }
140
141      /**
142      * @function      end_post( )
143      * @abstract      end point for post-order iteration
144      * @return      iterator_post containing the end of the tree in post-order
145      **/
146      iterator_post end_post( ) {
147          return iterator_post( (node_t*)0 );
148      }
149
150      /**
151      * @function      pushBack( )
152      * @abstract      a new TreeNode containing 'info' is added to the end
153      *                the node is added to the node that :
154      *                - is in the row as close to the root as possible
155      *                - has no children or only a left-child
156      *                - seen from the right hand side of the row
157      *                this is the 'natural' left-to-right filling order
158      *                compatible with array-based heaps and full b-trees

```



```

159      * @param      info, the contents of the new node
160      * @post       A node has been added.
161      **/
162      virtual node_t *pushBack( const INFO_T& info ) {
163          node_t *n =new node_t( info, 0 );
164          if( !m_root ) { // Empty tree, simplest case
165              m_root =n;
166          }
167          else { // Leaf node, there are two different scenarios
168              int max =getRowCountRecursive( m_root, 0 );
169              node_t *parent;
170              for( int i =1; i <= max; ++i ) {
171
172                  parent =getFirstEmptySlot( i );
173                  if( parent ) {
174                      if( !parent->leftChild( ) )
175                          parent->setLeftChild( n );
176                      else if( !parent->rightChild( ) )
177                          parent->setRightChild( n );
178                      n->setParent( parent );
179                      break;
180                  }
181              }
182          }
183          return n;
184      }
185
186      /**
187      * @function      insert( )
188      * @abstract      inserts node or subtree under a parent or creates an empty
189      *                  root node
190      * @param          info, contents of the new node
191      * @param          parent, parent node of the new node. When zero, the root is
192      *                  assumed
193      * @param          alignRight, insert() checks on which side of the parent
194      *                  node the new node can be inserted. By default, it checks
195      *                  the left side first.
196      *                  To change this behavior, set preferRight =true.
197      * @param          replaceBehavior, action if parent already has two children.
198      *                  One of:
199      *                  ABORT_ON_EXISTING - abort and return zero
200      *                  MOVE_EXISTING - make the parent's child a child of the new
201      *                  node, satisfies preferRight
202      *                  DELETE_EXISTING - remove one of the children of parent
203      *                  completely also satisfies preferRight
204      * @return          pointer to the inserted TreeNode, if insertion was
205      *                  successfull
206      * @pre             If the tree is empty, a root node will be created with info
207      *                  as it contents
208      * @pre             The instance pointed to by parent should be part of the
209      *                  called instance of Tree
210      * @post            Return zero if no node was created. Ownership is assumed on
211      *                  the new node.
212      *                  When DELETE_EXISTING is specified, the entire subtree on

```

```

213      *           preferred side may be deleted first.
214      **/
215      virtual node_t* insert( const INFO_T& info,
216                             node_t* parent =0,
217                             bool preferRight =false,
218                             int replaceBehavior =ABORT_ON_EXISTING ) {
219          if( !parent )
220              parent =m_root;
221
222          if( !parent )
223              return pushBack( info );
224
225          node_t *node =0;
226
227          if( !parent->leftChild( )
228              && ( !preferRight || ( preferRight &&
229                  parent->rightChild( ) ) ) ) {
230              node =new node_t( info, parent );
231              parent->setLeftChild( node );
232              node->setParent( parent );
233
234          } else if( !parent->rightChild( ) ) {
235              node =new node_t( info, parent );
236              parent->setRightChild( node );
237              node->setParent( parent );
238
239          } else if( replaceBehavior == MOVE_EXISTING ) {
240              node =new node_t( info, parent );
241              if( preferRight ) {
242                  node->setRightChild( parent->rightChild( ) );
243                  node->rightChild( )->setParent( node );
244                  parent->setRightChild( node );
245              } else {
246                  node->setLeftChild( parent->leftChild( ) );
247                  node->leftChild( )->setParent( node );
248                  parent->setLeftChild( node );
249              }
250
251          } else if( replaceBehavior == DELETE_EXISTING ) {
252              node =new node_t( info, parent );
253              if( preferRight ) {
254                  deleteRecursive( parent->rightChild( ) );
255                  parent->setRightChild( node );
256              } else {
257                  deleteRecursive( parent->leftChild( ) );
258                  parent->setLeftChild( node );
259              }
260
261          }
262          return node;
263      }
264
265      /**
266      * @function  replace( )

```

```

267      * @abstract    replaces an existing node with a new node
268      * @param      info, contents of the new node
269      * @param      node, node to be replaced. When zero, the root is assumed
270      * @param      alignRight, only for MOVE_EXISTING. If true, node will be
271      *              the right child of the new node. Otherwise, it will be the
272      *              left.
273      * @param      replaceBehavior, one of:
274      *              ABORT_ON_EXISTING - undefined for replace()
275      *              MOVE_EXISTING - make node a child of the new node,
276      *                          satisfies preferRight
277      *              DELETE_EXISTING - remove node completely
278      * @return      pointer to the inserted TreeNode, replace() is always
279      *              successful
280      * @pre          If the tree is empty, a root node will be created with info
281      *              as it contents
282      * @pre          The instance pointed to by node should be part of the
283      *              called instance of Tree
284      * @post         Ownership is assumed on the new node. When DELETE_EXISTING
285      *              is specified, the entire subtree pointed to by node is
286      *              deleted first.
287      **/
288      virtual node_t* replace( const INFO_T& info,
289                             node_t* node =0,
290                             bool alignRight =false ,
291                             int replaceBehavior =DELETE_EXISTING ) {
292          assert( replaceBehavior != ABORT_ON_EXISTING );
293
294          node_t *newnode =new node_t( info );
295          if( !node )
296              node =m_root;
297          if( !node )
298              return pushBack( info );
299
300          if( node->parent( ) ) {
301              newnode->setParent( node->parent( ) );
302              if( node->parent( )->leftChild( ) == node )
303                  node->parent( )->setLeftChild( newnode );
304              else
305                  node->parent( )->setRightChild( newnode );
306          } else
307              m_root =newnode;
308
309          if( replaceBehavior == DELETE_EXISTING ) {
310
311              deleteRecursive( node );
312          }
313          else if( replaceBehavior == MOVE_EXISTING ) {
314              if( alignRight )
315                  newnode->setRightChild( node );
316              else
317                  newnode->setLeftChild( node );
318              node->setParent( newnode );
319          }
320          return node;

```

```

321     }
322
323 /**
324  * @function   remove( )
325  * @abstract   removes and deletes node or subtree
326  * @param      n, node or subtree to be removed and deleted
327  * @post       after remove(), n points to an invalid address
328  */
329 virtual void remove( node_t *n ) {
330     if( !n )
331         return;
332     if( n->parent( ) ) {
333         if( n->parent( )->leftChild( ) == n )
334             n->parent( )->setLeftChild( 0 );
335         else if( n->parent( )->rightChild( ) == n )
336             n->parent( )->setRightChild( 0 );
337     }
338     deleteRecursive( n );
339 }
340
341 /**
342  * @function   clear( )
343  * @abstract   clears entire tree
344  * @pre        tree may be empty
345  * @post       all nodes and data are deallocated
346  */
347 void clear( ) {
348     deleteRecursive( m_root );
349     m_root =0;
350 }
351
352 /**
353  * @function   empty( )
354  * @abstract   test if tree is empty
355  * @return     true when empty
356  */
357 bool isEmpty( ) const {
358     return !m_root;
359 }
360
361 /**
362  * @function   root( )
363  * @abstract   returns address of the root of the tree
364  * @return     the address of the root of the tree is returned
365  * @pre        there needs to be a tree
366  */
367 node_t* root( ){
368     return m_root;
369 }
370
371 /**
372  * @function   row( )
373  * @abstract   returns an entire row/level in the tree
374  * @param      level, the desired row. Zero gives just the root.

```

```

375     * @return    a list containing all node pointers in that row
376     * @pre       level must be positive or zero
377     * @post
378     **/
379     nodelist row( int level ) {
380         nodelist rlist;
381         getRowRecursive( m_root, rlist, level );
382         return rlist;
383     }
384
385 /**
386  * @function    find( )
387  * @abstract    find the first occurrence of info and returns its node ptr
388  * @param       haystack, the root of the (sub)tree we want to look in
389  *              null if we want to start at the root of the tree
390  * @param       needle, the needle in our haystack
391  * @return      a pointer to the first occurrence of needle
392  * @post        there may be multiple occurrences of needle, we only return
393  *              one. A null-pointer is returned if no needle is found
394  **/
395     virtual node_t* find( node_t* haystack, const INFO_T& needle ) {
396         if( haystack == 0 ) {
397             if( m_root )
398                 haystack =m_root;
399             else
400                 return 0;
401         }
402         return findRecursive( haystack, needle );
403     }
404
405 /**
406  * @function    contains( )
407  * @abstract    determines if a certain content (needle) is found
408  * @param       haystack, the root of the (sub)tree we want to look in
409  *              null if we want to start at the root of the tree
410  * @param       needle, the needle in our haystack
411  * @return      true if needle is found
412  **/
413     bool contains( node_t* haystack, const INFO_T& needle ) {
414         return find( haystack, needle );
415     }
416
417 /**
418  * @function    toDot( )
419  * @abstract    writes tree in Dot-format to a stream
420  * @param       out, ostream to write to
421  * @pre         out must be a valid stream
422  * @post        out (file or cout) with the tree in dot-notation
423  **/
424     void toDot( ostream& out, const string & graphName ) {
425         if( isEmpty( ) )
426             return;
427         map<node_t *, int> addresses;
428         typename map< node_t *, int >::iterator adrIt;

```

```

429     int i =1;
430     int p;
431     iterator_pre it;
432     iterator_pre tempit;
433     addresses[m_root] =0;
434     out << "digraph " << graphName << '{' << endl << "' ' << 0 << "' ";
435     for( it =begin_pre( ); it != end_pre( ); ++it ) {
436         adrIt =addresses.find( &(*it) );
437         if( adrIt == addresses.end( ) ) {
438             addresses[&(*it)] =i;
439             p =i;
440             i ++;
441         }
442         if( (&(*it))->parent( ) != &(*tempit) )
443             out << ';' << endl << "' '
444                 << addresses.find( (&(*it))->parent( ))->second << "' ";
445         if( (&(*it)) != m_root )
446             out << " -> \"\" << p << "' ";
447         tempit =it;
448     }
449     out << ';' << endl;
450     for ( adrIt =addresses.begin( ); adrIt != addresses.end( ); ++adrIt )
451         out << adrIt->second << " [label=\"\"
452             << adrIt->first->info( ) << "\"\"];
453     out << '}'';
454 }
455
456 /**
457  * @function   copyFromNode( )
458  * @abstract   copies the the node source and its children to the node
459  *             dest
460  * @param      source, the node and its children that need to be copied
461  * @param      dest, the node who is going to get the copied children
462  * @param      left, this is true if it's a left child.
463  * @pre        there needs to be a tree and we can't copy to a root.
464  * @post       the subtree that starts at source is now also a child of
465  *             dest
466  */
467 void copyFromNode( node_t *source, node_t *dest, bool left ) {
468     if (!source)
469         return;
470     node_t *acorn =new node_t( dest );
471     if(left) {
472         if( dest->leftChild( ))
473             return;
474         dest->setLeftChild( acorn );
475     }
476     else {
477         if( dest->rightChild( ))
478             return;
479         dest->setRightChild( acorn );
480     }
481     cloneRecursive( source, acorn );
482 }

```

```

483
484     Tree<INFO_T>& operator=( const Tree<INFO_T>& tree ) {
485         clear( );
486         if( tree.m_root ) {
487             m_root =new node_t( (node_t*)0 );
488             cloneRecursive( tree.m_root, m_root );
489         }
490         return *this;
491     }
492
493 protected:
494     /**
495      * @function   cloneRecursive( )
496      * @abstract   cloning a subtree to a node
497      * @param      source, the node we want to start the cloning process from
498      * @param      dest, the node we want to clone to
499      * @post       the subtree starting at source is cloned to the node dest
500      */
501     void cloneRecursive( node_t *source, node_t* dest ) {
502         dest->info() =source->info();
503         if( source->leftChild( ) ) {
504             node_t *left =new node_t( dest );
505             dest->setLeftChild( left );
506             cloneRecursive( source->leftChild( ), left );
507         }
508         if( source->rightChild( ) ) {
509             node_t *right =new node_t( dest );
510             dest->setRightChild( right );
511             cloneRecursive( source->rightChild( ), right );
512         }
513     }
514
515     /**
516      * @function   deleteRecursive( )
517      * @abstract   delete all nodes of a given tree
518      * @param      root, starting point, is deleted last
519      * @post       the subtree has been deleted
520      */
521     void deleteRecursive( node_t *root ) {
522         if( !root )
523             return;
524         deleteRecursive( root->leftChild( ) );
525         deleteRecursive( root->rightChild( ) );
526         delete root;
527     }
528
529     /**
530      * @function   getRowCountRecursive( )
531      * @abstract   calculate the maximum depth/row count in a subtree
532      * @param      root, starting point
533      * @param      level, starting level
534      * @return     maximum depth/rows in the subtree
535      */
536     int getRowCountRecursive( node_t* root, int level ) {

```

```

537         if( !root )
538             return level;
539         return max(
540             getRowCountRecursive( root->leftChild( ), level+1 ),
541             getRowCountRecursive( root->rightChild( ), level+1 ) );
542     }
543
544 /**
545  * @function   getRowRecursive( )
546  * @abstract   compile a full list of one row in the tree
547  * @param      root, starting point
548  * @param      rlist, reference to the list so far
549  * @param      level, how many level still to go
550  * @post       a list of a row in the tree has been made.
551  */
552 void getRowRecursive( node_t* root, nodelist &rlist, int level ) {
553     // Base-case
554     if( !level ) {
555         rlist.push_back( root );
556     } else if( root ){
557         level--;
558         if( level && !root->leftChild( ) )
559             for( int i =0; i < (level<<1); ++i )
560                 rlist.push_back( 0 );
561         else
562             getRowRecursive( root->leftChild( ), rlist, level );
563
564         if( level && !root->rightChild( ) )
565             for( int i =0; i < (level<<1); ++i )
566                 rlist.push_back( 0 );
567         else
568             getRowRecursive( root->rightChild( ), rlist, level );
569     }
570 }
571
572 /**
573  * @function   findRecursive( )
574  * @abstract   first the first occurrence of needle and return its node
575  *             ptr
576  * @param      haystack, root of the search tree
577  * @param      needle, copy of the data to find
578  * @return     the node that contains the needle
579  */
580 node_t *findRecursive( node_t* haystack, const INFO_T &needle ) {
581     if( haystack->info( ) == needle )
582         return haystack;
583
584     node_t *n =0;
585     if( haystack->leftChild( ) )
586         n =findRecursive( haystack->leftChild( ), needle );
587     if( !n && haystack->rightChild( ) )
588         n =findRecursive( haystack->rightChild( ), needle );
589     return n;
590 }

```



```

591
592         friend class TreeNodeIterator_pre<INFO_T>;
593         friend class TreeNodeIterator_in<INFO_T>;
594         friend class SplayTree<INFO_T>;
595         TreeNode<INFO_T> *m_root;
596
597     private:
598         /**
599         * @function   getFirstEmptySlot( )
600         * @abstract   when a row has a continuous empty space on the right,
601         *               find the left-most parent in the above row that has
602         *               at least one empty slot.
603         * @param       level, how many level still to go
604         * @return      the first empty slot where we can put a new node
605         * @pre         level should be > 1
606         **/
607         node_t *getFirstEmptySlot( int level ) {
608             node_t *p =0;
609             nodelist rlist =row( level-1 ); // we need the parents of this level
610             /** changed auto to int **/
611             for( auto it =rlist.rbegin( ); it !=rlist.rend( ); ++it ) {
612                 if( !(*it)->hasChildren( ) )
613                     p =(*it);
614                 else if( !(*it)->rightChild( ) ) {
615                     p =(*it);
616                     break;
617                 } else
618                     break;
619             }
620             return p;
621         }
622     };
623
624 #endif

```

## 6.4 TreeNode.h

```

1  /**
2   * Treenode:
3   *
4   * @author   Micky Faas (s1407937)
5   * @author   Lisette de Schipper (s1396250)
6   * @file     Treenode.h
7   * @date     26-10-2014
8   **/
9
10 #ifndef TREENODE.H
11 #define TREENODE.H
12
13 using namespace std;
14
15 template <class INFO_T> class Tree;
16 class ExpressionTree;
17

```

```

18 template <class INFO_T> class TreeNode
19 {
20     public:
21         /**
22          * @function   TreeNode( )
23          * @abstract   Constructor, creates a node
24          * @param      info, the contents of a node
25          * @param      parent, the parent of the node
26          * @post       A node has been created.
27          */
28         TreeNode( const INFO_T& info, TreeNode<INFO_T>* parent =0 )
29             : m_lchild( 0 ), m_rchild( 0 ) {
30             m_info =info;
31             m_parent =parent;
32         }
33
34         /**
35          * @function   TreeNode( )
36          * @abstract   Constructor, creates a node
37          * @param      parent, the parent of the node
38          * @post       A node has been created.
39          */
40         TreeNode( TreeNode<INFO_T>* parent =0 )
41             : m_lchild( 0 ), m_rchild( 0 ) {
42             m_parent =parent;
43         }
44
45         /**
46          * @function   =
47          * @abstract   Sets a nodes content to N
48          * @param      n, the contents you want the node to have
49          * @post       The node now has those contents.
50          */
51         void operator =( INFO_T n ) { m_info =n; }
52
53         /**
54          * @function   INFO_T( ), info( )
55          * @abstract   Returns the content of a node
56          * @return      m_info, the contents of the node
57          */
58         operator INFO_T( ) const { return m_info; }
59         const INFO_T &info( ) const { return m_info; }
60         INFO_T &info( ) { return m_info; }
61         /**
62          * @function   atRow( )
63          * @abstract   returns the level or row-number of this node
64          * @return      row, an int of row the node is at
65          */
66         int atRow( ) const {
67             const TreeNode<INFO_T> *n =this;
68             int row =0;
69             while( n->parent( ) ) {
70                 n =n->parent( );
71                 row++;

```

```

72     }
73     return row;
74 }
75
76 /**
77  * @function parent( ), leftChild( ), rightChild( )
78  * @abstract returns the address of the parent, left child and right
79  *           child respectively
80  * @return   the address of the requested family member of the node
81  */
82 TreeNode<INFO_T> *parent( ) const { return m_parent; }
83 TreeNode<INFO_T> *leftChild( ) const { return m_lchild; }
84 TreeNode<INFO_T> *rightChild( ) const { return m_rchild; }
85
86 /**
87  * @function swapWith( )
88  * @abstract Swaps this node with another node in the tree
89  * @param    n, the node to swap this one with
90  * @pre      both this node and n must be in the same parent tree
91  * @post     n will have the parent and children of this node
92  *           and vice verse. Both nodes retain their data.
93  */
94 void swapWith( TreeNode<INFO_T>* n ) {
95     bool this_wasLeftChild =false, n_wasLeftChild =false;
96     if( parent( ) && parent( )->leftChild( ) == this )
97         this_wasLeftChild =true;
98     if( n->parent( ) && n->parent( )->leftChild( ) == n )
99         n_wasLeftChild =true;
100
101     // Swap the family info
102     TreeNode<INFO_T>* newParent =
103         ( n->parent( ) == this ) ? n : n->parent( );
104     TreeNode<INFO_T>* newLeft =
105         ( n->leftChild( ) == this ) ? n : n->leftChild( );
106     TreeNode<INFO_T>* newRight =
107         ( n->rightChild( ) == this ) ? n : n->rightChild( );
108
109     n->setParent( parent( ) == n ? this : parent( ) );
110     n->setLeftChild( leftChild( ) == n ? this : leftChild( ) );
111     n->setRightChild( rightChild( ) == n ? this : rightChild( ) );
112
113     setParent( newParent );
114     setLeftChild( newLeft );
115     setRightChild( newRight );
116
117     // Restore applicable pointers
118     if( n->leftChild( ) )
119         n->leftChild( )->setParent( n );
120     if( n->rightChild( ) )
121         n->rightChild( )->setParent( n );
122     if( leftChild( ) )
123         leftChild( )->setParent( this );
124     if( rightChild( ) )
125         rightChild( )->setParent( this );

```

```

126         if( n->parent( ) ) {
127             if( this_wasLeftChild )
128                 n->parent( )->setLeftChild( n );
129             else
130                 n->parent( )->setRightChild( n );
131         }
132         if( parent( ) ) {
133             if( n_wasLeftChild )
134                 parent( )->setLeftChild( this );
135             else
136                 parent( )->setRightChild( this );
137         }
138     }
139
140     /**
141     * @function   replace( )
142     * @abstract   Replaces the node with another node in the tree
143     * @param      n, the node we replace the node with, this one gets deleted
144     * @pre        both this node and n must be in the same parent tree
145     * @post       The node will be replaced and n will be deleted.
146     */
147     void replace( TreeNode<INFO_T>* n ) {
148         bool n_wasLeftChild =false;
149
150         if( n->parent( ) && n->parent( )->leftChild( ) == n )
151             n_wasLeftChild =true;
152
153         // Swap the family info
154         TreeNode<INFO_T>* newParent =
155             ( n->parent( ) == this ) ? n : n->parent( );
156         TreeNode<INFO_T>* newLeft =
157             ( n->leftChild( ) == this ) ? n : n->leftChild( );
158         TreeNode<INFO_T>* newRight =
159             ( n->rightChild( ) == this ) ? n : n->rightChild( );
160
161         setParent( newParent );
162         setLeftChild( newLeft );
163         setRightChild( newRight );
164         m_info = n->m_info;
165
166         // Restore applicable pointers
167         if( leftChild( ) )
168             leftChild( )->setParent( this );
169         if( rightChild( ) )
170             rightChild( )->setParent( this );
171
172         if( parent( ) ) {
173             if( n_wasLeftChild )
174                 parent( )->setLeftChild( this );
175             else
176                 parent( )->setRightChild( this );
177         }
178         delete n;
179     }

```

```

180
181
182     /**
183     * @function sibling( )
184     * @abstract returns the address of the sibling
185     * @return the address to the sibling or zero if there is no sibling
186     */
187     TreeNode<INFO_T>* sibling( ) {
188         if( parent( )->leftChild( ) == this )
189             return parent( )->rightChild( );
190         else if( parent( )->rightChild( ) == this )
191             return parent( )->leftChild( );
192         else
193             return 0;
194     }
195
196     /**
197     * @function hasChildren( ), hasParent( ), isFull( )
198     * @abstract Returns whether the node has children, has parents or is
199     *             full (has two children) respectively
200     * @param
201     * @return true or false, depending on what is requested from the node.
202     *         if hasChildren is called and the node has children, it will
203     *         return true, otherwise false.
204     *         If hasParent is called and the node has a parent, it will
205     *         return true, otherwise false.
206     *         If isFull is called and the node has two children, it will
207     *         return true, otherwise false.
208     */
209     bool hasChildren( ) const { return m_lchild || m_rchild; }
210     bool hasParent( ) const { return m_parent; }
211     bool isFull( ) const { return m_lchild && m_rchild; }
212
213     protected:
214         friend class Tree<INFO_T>;
215         friend class ExpressionTree;
216
217     /**
218     * @function setParent( ), setLeftChild( ), setRightChild( )
219     * @abstract sets the parent, left child and right child of the
220     *             particular node respectively
221     * @param p, the node we want to set a certain family member of
222     * @return void
223     * @post The node now has a parent, a left child or a right child
224     *        respectively.
225     */
226     void setParent( TreeNode<INFO_T> *p ) { m_parent =p; }
227     void setLeftChild( TreeNode<INFO_T> *p ) { m_lchild =p; }
228     void setRightChild( TreeNode<INFO_T> *p ) { m_rchild =p; }
229
230     private:
231         INFO_T m_info;
232         TreeNode<INFO_T> *m_parent;
233         TreeNode<INFO_T> *m_lchild;
234         TreeNode<INFO_T> *m_rchild;

```

```

234 };
235
236 /**
237 * @function <<
238 * @abstract the contents of the node are returned
239 * @param out, in what format we want to get the contents
240 * @param rhs, the node of which we want the contents
241 * @return the contents of the node.
242 */
243 template <class INFO_T> ostream &operator <<(ostream& out, const TreeNode<INFO_T> & r
244     out << rhs.info( );
245     return out;
246 }
247
248 #endif

```

## 6.5 TreeNodeIterator.h

```

1  /**
2   * TreeNodeIterator: Provides a set of iterators that follow the STL-standard
3   *
4   * @author Micky Faas (s1407937)
5   * @author Lisette de Schipper (s1396250)
6   * @file   TreeNodeIterator.h
7   * @date   26-10-2014
8   */
9
10 #include <iterator>
11 #include "TreeNode.h"
12
13 template <class INFO_T> class TreeNodeIterator
14     : public std::iterator<std::forward_iterator_tag,
15                             TreeNode<INFO_T>> {
16 public:
17     typedef TreeNode<INFO_T> node_t;
18
19     /**
20     * @function   TreeNodeIterator( )
21     * @abstract   (copy)constructor
22     * @pre        TreeNodeIterator is abstract and cannot be constructed
23     */
24     TreeNodeIterator( node_t* ptr =0 ) : p( ptr ) { }
25     TreeNodeIterator( const TreeNodeIterator& it ) : p( it.p ) { }
26
27     /**
28     * @function   (in)equality operator overload
29     * @abstract   Test (in)equality for two TreeNodeIterators
30     * @param      rhs, right-hand side of the comparison
31     * @return      true if both iterators point to the same node (==)
32     *              false if both iterators point to the same node (!=)
33     */
34     bool operator == (const TreeNodeIterator& rhs) { return p==rhs.p; }
35     bool operator != (const TreeNodeIterator& rhs) { return p!=rhs.p; }
36

```

```

37     /**
38      * @function   operator*( )
39      * @abstract   Cast operator to node_t reference
40      * @return     The value of the current node
41      * @pre        Must point to a valid node
42      */
43     node_t& operator*( ) { return *p; }
44
45     /**
46      * @function   operator++( )
47      * @abstract   pre- and post increment operators
48      * @return     TreeNodeIterator that has iterated one step
49      */
50     TreeNodeIterator &operator++( ) { next( ); return *this; }
51     TreeNodeIterator operator++( int )
52     { TreeNodeIterator tmp( *this ); operator++( ); return tmp; }
53 protected:
54
55     /**
56      * @function   next( ) //(pure virtual)
57      * @abstract   Implement this function to implement your own iterator
58      */
59     virtual bool next( ){ return false; }// =0;
60     node_t *p;
61 };
62
63 template <class INFO_T> class TreeNodeIterator_pre
64     : public TreeNodeIterator<INFO_T> {
65 public:
66     typedef TreeNode<INFO_T> node_t;
67
68     TreeNodeIterator_pre( node_t* ptr =0 )
69         : TreeNodeIterator<INFO_T>( ptr ) { }
70     TreeNodeIterator_pre( const TreeNodeIterator<INFO_T>& it )
71         : TreeNodeIterator<INFO_T>( it ) { }
72     TreeNodeIterator_pre( const TreeNodeIterator_pre& it )
73         : TreeNodeIterator<INFO_T>( it.p ) { }
74
75     TreeNodeIterator_pre &operator++( ) { next( ); return *this; }
76     TreeNodeIterator_pre operator++( int )
77     { TreeNodeIterator_pre tmp( *this ); operator++( ); return tmp; }
78
79 protected:
80     using TreeNodeIterator<INFO_T>::p;
81
82     /**
83      * @function   next( )
84      * @abstract   Takes one step in pre-order traversal
85      * @return     returns true if such a step exists
86      */
87     bool next( ) {
88         if( !p )
89             return false;
90         if( p->hasChildren( ) ) { // a possible child that can be the next

```

```

91         p = p->leftChild( ) ? p->leftChild( ) : p->rightChild( );
92         return true;
93     }
94     else if( p->hasParent( ) // we have a right brother
95             && p->parent( )->rightChild( )
96             && p->parent( )->rightChild( ) != p ) {
97         p = p->parent( )->rightChild( );
98         return true;
99     }
100    else if( p->hasParent( ) ) { // just a parent, thus we go up
101        TreeNode<INFO_T> *tmp = p->parent( );
102        while( tmp->parent( ) ) {
103            if( tmp->parent( )->rightChild( )
104                && tmp->parent( )->rightChild( ) != tmp ) {
105                p = tmp->parent( )->rightChild( );
106                return true;
107            }
108            tmp = tmp->parent( );
109        }
110    }
111    // Nothing left
112    p = 0;
113    return false;
114 }
115
116 };
117
118 template <class INFO_T> class TreeNodeIterator_in
119     : public TreeNodeIterator<INFO_T>{
120 public:
121     typedef TreeNode<INFO_T> node_t;
122
123     TreeNodeIterator_in( node_t* ptr = 0 )
124         : TreeNodeIterator<INFO_T>( ptr ) { }
125     TreeNodeIterator_in( const TreeNodeIterator<INFO_T>& it )
126         : TreeNodeIterator<INFO_T>( it ) { }
127     TreeNodeIterator_in( const TreeNodeIterator_in& it )
128         : TreeNodeIterator<INFO_T>( it.p ) { }
129
130     TreeNodeIterator_in &operator++( ) { next( ); return *this; }
131     TreeNodeIterator_in operator++( int )
132     { TreeNodeIterator_in tmp( *this ); operator++( ); return tmp; }
133
134 protected:
135     using TreeNodeIterator<INFO_T>::p;
136     /**
137     * @function next( )
138     * @abstract Takes one step in in-order traversal
139     * @return returns true if such a step exists
140     */
141     bool next( ) {
142         if( p->rightChild( ) ) {
143             p = p->rightChild( );
144             while( p->leftChild( ) )

```



```

145         p =p->leftChild( );
146         return true;
147     }
148     else if( p->parent( ) && p->parent( )->leftChild( ) == p ) {
149         p =p->parent( );
150         return true;
151     } else if( p->parent( ) && p->parent( )->rightChild( ) == p ) {
152         p =p->parent( );
153         while( p->parent( ) && p == p->parent( )->rightChild( ) ) {
154             p =p->parent( );
155         }
156         if( p )
157             p =p->parent( );
158         if( p )
159             return true;
160         else
161             return false;
162     }
163     // Er is niks meer
164     p =0;
165     return false;
166 }
167 };
168
169 template <class INFO_T> class TreeNodeIterator_post
170     : public TreeNodeIterator<INFO_T>{
171 public:
172     typedef TreeNode<INFO_T> node_t;
173
174     TreeNodeIterator_post( node_t* ptr =0 )
175         : TreeNodeIterator<INFO_T>( ptr ) { }
176     TreeNodeIterator_post( const TreeNodeIterator<INFO_T>& it )
177         : TreeNodeIterator<INFO_T>( it ) { }
178     TreeNodeIterator_post( const TreeNodeIterator_post& it )
179         : TreeNodeIterator<INFO_T>( it.p ) { }
180
181     TreeNodeIterator_post &operator++( ) { next( ); return *this; }
182     TreeNodeIterator_post operator++( int )
183     { TreeNodeIterator_post tmp( *this ); operator++( ); return tmp; }
184
185 protected:
186     using TreeNodeIterator<INFO_T>::p;
187     /**
188     * @function next( )
189     * @abstract Takes one step in post-order traversal
190     * @return returns true if such a step exists
191     */
192     bool next( ) {
193
194         if( p->hasParent( ) // We have a right brother
195             && p->parent( )->rightChild( )
196             && p->parent( )->rightChild( ) != p ) {
197             p =p->parent( )->rightChild( );
198             while( p->leftChild( ) )

```

```

199         p =p->leftChild( );
200         return true;
201     } else if( p->parent( ) ) {
202         p =p->parent( );
203         return true;
204     }
205     // Nothing left
206     p =0;
207     return false;
208 }
209 };

```

## 6.6 SelfOrganizingTree.h

```

1  /**
2   * SelfOrganizingTree - Abstract base type inheriting from Tree
3   *
4   * @author Micky Faas (s1407937)
5   * @author Lisette de Schipper (s1396250)
6   * @file SelfOrganizingTree.h
7   * @date 3-11-2014
8   */
9
10 #ifndef SELFORGANIZINGTREE_H
11 #define SELFORGANIZINGTREE_H
12
13 #include "BinarySearchTree.h"
14
15 template <class INFO_T> class SelfOrganizingTree : public BinarySearchTree<INFO_T> {
16 public:
17     typedef BSTNode<INFO_T> node_t;
18     typedef BinarySearchTree<INFO_T> S; // super class
19
20     /**
21      * @function SelfOrganizingTree( ) : S( )
22      * @abstract Constructor
23      */
24     SelfOrganizingTree( ) : S( ) { }
25
26     /**
27      * @function rotateLeft( ) and rotateRight( )
28      * @abstract Performs a rotation with the given node as root of the
29      * rotating subtree, either left of right.
30      * The tree's root pointer will be updated if neccesary.
31      * @param node, the node to rotate
32      * @pre The node must be a node in this tree
33      * @post The node may be be the new root of the tree
34      * No nodes will be invalidated and no new memory is
35      * allocated. Iterators may become invalid.
36      */
37     virtual node_t *rotateLeft( node_t * node ){
38         if( this->root( ) == node )
39             return static_cast<node_t *>( S::m_root = node->rotateLeft( ) );
40         else

```

```

41         return node->rotateLeft( );
42     }
43
44     virtual node_t *rotateRight( node_t * node ){
45         if( this->root( ) == node )
46             return static_cast<node_t *>( S::m_root = node->rotateRight( ) );
47         else
48             return node->rotateRight( );
49     }
50
51     private:
52
53 };
54
55
56 #endif

```

## 6.7 BinarySearchTree.h

```

1  /**
2   * BinarySearchTree - BST that inherits from Tree
3   *
4   * @author Micky Faas (s1407937)
5   * @author Lisette de Schipper (s1396250)
6   * @file BinarySearchTree.h
7   * @date 3-11-2014
8   */
9
10 #ifndef BINARYSEARCHTREE_H
11 #define BINARYSEARCHTREE_H
12
13 #include "Tree.h"
14 #include "BSTNode.h"
15
16 template <class INFO_T> class BinarySearchTree : public Tree<INFO_T> {
17     public:
18         typedef BSTNode<INFO_T> node_t;
19         typedef Tree<INFO_T> S; // super class
20
21         BinarySearchTree( ) : S( ) { }
22         BinarySearchTree( const BinarySearchTree& cpy ) : S( cpy ) { }
23
24         virtual ~BinarySearchTree( ) { }
25
26     /**
27      * @function pushBack( )
28      * @abstract reimplemented virtual function from Tree<>
29      *          this function is semantically identical to insert()
30      * @param info, the contents of the new node
31      */
32     virtual node_t *pushBack( const INFO_T& info ) {
33         return insert( info );
34     }
35

```

```

36      /**
37      * @function   insert( )
38      * @abstract   reimplemented virtual function from Tree<>
39      *             the exact location of the new node is determined
40      *             by the rules of the binary search tree.
41      * @param      info, the contents of the new node
42      * @param      parent, ignored
43      * @param      preferRight, ignored
44      * @param      replaceBehavior, ignored
45      * @return     returns a pointer to the inserted node
46      */
47      virtual node_t* insert( const INFO_T& info,
48                             TreeNode<INFO_T>* parent =0, // Ignored
49                             bool preferRight =false,      // Ignored
50                             int replaceBehavior =S::ABORT_ON_EXISTING ) { // Ignored
51          node_t *n =new node_t( );
52          return insertInto( info, n );
53      }
54
55      /**
56      * @function   replace( )
57      * @abstract   reimplemented virtual function from Tree<>
58      *             replaces a given node or the root
59      *             the location of the replaced node may be different
60      *             due to the consistency of the binary search tree
61      * @param      info, the contents of the new node
62      * @param      node, node to be replaced
63      * @param      alignRight, ignored
64      * @param      replaceBehavior, ignored
65      * @return     returns a pointer to the new node
66      * @pre        node should be in this tree
67      * @post       replace() will delete and/or remove node.
68      *             if node is 0, it will take the root instead
69      */
70      virtual node_t* replace( const INFO_T& info,
71                              TreeNode<INFO_T>* node =0,
72                              bool alignRight =false,
73                              int replaceBehavior =S::DELETE_EXISTING ) {
74          node_t *newnode;
75          if( !node )
76              node =S::m_root;
77          if( !node )
78              return pushBack( info );
79
80          bool swap =false;
81          // We can either just swap the new node with the old and remove
82          // the old, or we can remove the old and add the new node via
83          // pushBack(). This depends on the value of info
84          if( !node->hasChildren( ) ) {
85              swap =true;
86          }
87          else if( !(node->leftChild( )
88                  && node->leftChild( )->info( ) > info )
89                  && !(node->rightChild( )

```

```

90         && node->rightChild( )->info( ) < info ) ) {
91             swap =true;
92         }
93         if( swap ) {
94             newnode =new node_t( info );
95             if( node == S::m_root )
96                 S::m_root =newnode;
97             node->swapWith( newnode );
98             delete node;
99         } else {
100             remove( node );
101             newnode =pushBack( info );
102         }
103     }
104     return newnode;
105 }
106
107 /**
108  * @function remove( )
109  * @abstract reimplemented virtual function from Tree<>
110  *           removes a given node or the root and restores the
111  *           BST properties
112  * @param    node, node to be removed
113  * @pre      node should be in this tree
114  * @post     memory for node will be deallocated
115  */
116 virtual void remove( TreeNode<INFO_T> *node ) {
117     node_t *n =static_cast<node_t*>( node );
118
119     while( n->isFull( ) ) {
120         // the difficult case
121         // we could take either left of right here
122         TreeNode<INFO_T> *temp;
123         temp =n->leftChild( );
124         while( temp->rightChild( ) ) {
125             temp =temp->rightChild( );
126         }
127         if( n == S::m_root )
128             S::m_root =temp;
129         n->swapWith( temp );
130     }
131
132     // Assume the above is fixed
133     while( n->hasChildren( ) ) {
134         if( n->leftChild( ) ) {
135             if( n == S::m_root )
136                 S::m_root =n->leftChild( );
137             n->swapWith( n->leftChild( ) );
138         }
139         else {
140             if( n == S::m_root )
141                 S::m_root =n->rightChild( );
142             n->swapWith( n->rightChild( ) );
143         }
144     }

```

```

144         }
145     }
146
147     if( n->parent( ) && n->parent( )->leftChild( ) == n )
148         static_cast<node_t*>( n->parent( ) )->setLeftChild( 0 );
149     else if( n->parent( ) && n->parent( )->rightChild( ) == n )
150         static_cast<node_t*>( n->parent( ) )->setRightChild( 0 );
151     delete n;
152 }
153
154 /**
155  * @function    find( )
156  * @abstract    reimplemented virtual function from Tree<>
157  *              performs a binary search in a given (sub)tree
158  * @param       haystack, the subtree to search. Give 0 for the entire tree
159  * @param       needle, key/info-value to find
160  * @return      returns a pointer to node, if found
161  * @pre         haystack should be in this tree
162  * @post        may return 0
163  */
164 virtual TreeNode<INFO_T>* find( TreeNode<INFO_T>* haystack,
165                                const INFO_T& needle ) {
166     m_searchStepCounter = 0;
167
168     if( !haystack )
169         haystack = S::m_root;
170     while( haystack && haystack->info( ) != needle ) {
171         m_searchStepCounter++;
172         if( haystack->info( ) > needle )
173             haystack = haystack->leftChild( );
174         else
175             haystack = haystack->rightChild( );
176     }
177     if( !haystack )
178         m_searchStepCounter = -1;
179     return haystack;
180 }
181
182 /**
183  * @function    lastSearchStepCount( )
184  * @abstract    gives the amount of steps needed to complete the most
185  *              recent call to find( )
186  * @return      positive amount of steps on a defined search result,
187  *              -1 on no search result
188  */
189 virtual int lastSearchStepCount( ) const {
190     return m_searchStepCounter;
191 }
192
193 /**
194  * @function    min( )
195  * @abstract    Returns the node with the least value in a binary search
196  *              tree. This is achieved through recursion.
197  * @param       node - the node from which we start looking

```

```

198      * @return      Eventually, at the end of the recursion, we return the
199      *              address of the node with the smallest value.
200      * @post         The node with the smallest value is returned.
201      **/
202      node_t* min( node_t* node ) const {
203          return node->leftChild( ) ?
204              min(static_cast<node_t*>( node->leftChild( ) ) ) : node;
205      }
206
207      /**
208      * @function      min( )
209      * @abstract      We call the function mentioned above and then
210      *              return the node with the least value in a binary search
211      *              tree.
212      * @return        We return the address of the node with the smallest value.
213      * @post         The node with the smallest value is returned.
214      **/
215      node_t* min( ) const {
216          return min( static_cast<node_t*>( this->root( ) ) );
217      }
218
219      /**
220      * @function      max( )
221      * @abstract      Returns the node with the highest value in a binary
222      *              search tree. This is achieved through recursion.
223      * @param         node - the node from which we start looking
224      * @return        Eventually, at the end of the recursion, we return the
225      *              address of the node with the highest value.
226      * @post         The node with the highest value is returned.
227      **/
228      node_t* max( node_t* node ) const {
229          return node->rightChild( ) ?
230              max(static_cast<node_t*>( node->rightChild( ) ) ) : node;
231      }
232
233      /**
234      * @function      max( )
235      * @abstract      We call the function mentioned above and then
236      *              return the node with the highest value in a binary
237      *              search tree.
238      * @return        We return the address of the node with the highest value.
239      * @post         The node with the highest value is returned.
240      **/
241      node_t* max( ) const {
242          return max( static_cast<node_t*>( this->root( ) ) );
243      }
244
245      protected:
246      /**
247      * @function      insertInto( )
248      * @abstract      Inserts new node into the tree following BST rules
249      *              Assumes that the memory for the node is already allocated
250      *              This function exists mainly because of derived classes
251      *              want to insert nodes of a derived type.

```

```

252     * @param      info, the contents of the new node
253     * @param      n, node pointer, should be already allocated
254     * @return      returns a pointer to the inserted node
255     **/
256     virtual node_t* insertInto( const INFO_T& info,
257                                node_t* n ) { // Preallocated
258         n->info() =info;
259
260         if( !S::m_root )
261             S::m_root =n;
262         else {
263             node_t *parent =0;
264             node_t *sub =static_cast<node_t*>( S::m_root );
265             do {
266                 if( *n < *sub ) {
267                     parent =sub;
268                     sub =static_cast<node_t*>( parent->leftChild( ) );
269                 }
270                 else {
271                     parent =sub;
272                     sub =static_cast<node_t*>( parent->rightChild( ) );
273                 }
274             } while( sub );
275             if( *n < *parent )
276                 parent->setLeftChild( n );
277             else
278                 parent->setRightChild( n );
279             n->setParent( parent );
280         }
281         return n;
282     }
283
284     int m_searchStepCounter;
285 };
286
287 #endif

```

## 6.8 BSTNode.h

```

1  /**
2   * BSTNode - Node atom for BinarySearchTree
3   *
4   * @author      Micky Faas (s1407937)
5   * @author      Lisette de Schipper (s1396250)
6   * @file         BSTNode.h
7   * @date         3-11-2014
8   **/
9
10 #ifndef BSTNODE_H
11 #define BSTNODE_H
12
13 #include "TreeNode.h"
14
15 template <class INFO_T> class BinarySearchTree;

```



```

16
17 template <class INFO_T> class BSTNode : public TreeNode<INFO_T>
18 {
19     public:
20         typedef TreeNode<INFO_T> S; // super class
21
22         /**
23          * @function   BSTNode( )
24          * @abstract   Constructor, creates a node
25          * @param      info, the contents of a node
26          * @param      parent, the parent of the node
27          * @post       A node has been created.
28          */
29         BSTNode( const INFO_T& info, BSTNode<INFO_T>* parent =0 )
30             : S( info, parent ) { }
31
32         /**
33          * @function   BSTNode( )
34          * @abstract   Constructor, creates a node
35          * @param      parent, the parent of the node
36          * @post       A node has been created.
37          */
38         BSTNode( BSTNode<INFO_T>* parent =0 )
39             : S( (S)parent ) { }
40
41         // Idea: rotate this node left and return the node that comes in its place
42         BSTNode *rotateLeft( ) {
43
44             if( !this->rightChild( ) ) // Cannot rotate
45                 return this;
46
47             bool isLeftChild =this->parent( ) && this == this->parent( )->leftChild(
48
49             // new root of tree
50             BSTNode *newTop =static_cast<BSTNode *>(this->rightChild( ));
51             // new rightchild of the node that is rotated
52             BSTNode *newRight =static_cast<BSTNode *>(newTop->leftChild( ));
53             // the parent under which all of the magic is happening
54             BSTNode *topParent =static_cast<BSTNode *>(this->parent( ));
55
56             // We become left-child of our right-child
57             // newTop takes our place with our parent
58             newTop->setParent( topParent );
59             if( isLeftChild && topParent )
60                 topParent->setLeftChild( newTop );
61             else if( topParent )
62                 topParent->setRightChild( newTop );
63
64             newTop->setLeftChild( this );
65             this->setParent( newTop );
66
67             // We take the left-child of newTop as our right-child
68             this->setRightChild( newRight );
69             if( newRight )

```

```

70         newRight->setParent( this );
71
72     return newTop;
73 }
74
75 // Idea: rotate this node right and return the node that comes in its place
76 BSTNode *rotateRight( ) {
77     if( !this->leftChild( ) ) // Cannot rotate
78         return this;
79
80     bool isRightChild =this->parent( ) && this == this->parent( )->rightChild
81
82     // new root of tree
83     BSTNode *newTop =static_cast<BSTNode *>(this->leftChild( ));
84     // new leftchild of the node that is rotated
85     BSTNode *newLeft =static_cast<BSTNode *>(newTop->rightChild( ));
86     // the parent under which all of the magic is happening
87     BSTNode *topParent =static_cast<BSTNode *>(this->parent( ));
88
89     // We become left-child of our right-child
90     // newTop takes our place with our parent
91     newTop->setParent( topParent );
92     if( isRightChild && topParent )
93         topParent->setRightChild( newTop );
94     else if( topParent )
95         topParent->setLeftChild( newTop );
96
97     newTop->setRightChild( this );
98     this->setParent( newTop );
99
100    // We take the left-child of newTop as our right-child
101    this->setLeftChild( newLeft );
102    if( newLeft )
103        newLeft->setParent( this );
104
105    return newTop;
106 }
107
108 bool operator <( const BSTNode<INFO_T> &rhs ) {
109     return S::info() < rhs.info();
110 }
111
112 bool operator <=( const BSTNode<INFO_T> &rhs ) {
113     return S::info() <= rhs.info();
114 }
115
116 bool operator >( const BSTNode<INFO_T> &rhs ) {
117     return S::info() > rhs.info();
118 }
119
120 bool operator >=( const BSTNode<INFO_T> &rhs ) {
121     return S::info() >= rhs.info();
122 }
123 protected:

```

```

124         friend class BinarySearchTree<INFO_T>;
125     };
126
127 #endif

```

## 6.9 AVLTree.h

```

1  /**
2   * AVLTree - AVL-SelfOrganizingTree that inherits from SelfOrganizingTree
3   *
4   * @author Micky Faas (s1407937)
5   * @author Lisette de Schipper (s1396250)
6   * @file AVLTree.h
7   * @date 9-12-2014
8   */
9
10 #ifndef AVLTREE_H
11 #define AVLTREE_H
12
13 #include "SelfOrganizingTree.h"
14 #include "AVLNode.h"
15
16 template <class INFO_T> class AVLTree : public SelfOrganizingTree<INFO_T> {
17     public:
18         typedef AVLNode<INFO_T> node_t;
19         typedef SelfOrganizingTree<INFO_T> S; // super class
20
21         /**
22          * @function AVLTree( )
23          * @abstract constructor
24          * @post An AVLTree is created
25          */
26         AVLTree( ) : S( ) { }
27
28         /**
29          * @function AVLTree( )
30          * @abstract constructor
31          * @param cpy
32          * @post An AVLTree is created
33          */
34         AVLTree( const AVLTree& cpy ) : S( cpy ) { }
35
36         /**
37          * @function insert( )
38          * @abstract A node with label 'info' is inserted into the tree and
39          * put in the right place. A label may not appear twice in
40          * a tree.
41          * @param info - the label of the node
42          * @return the node we inserted
43          * @post The tree now contains a node with 'info'
44          */
45         node_t* insert( const INFO_T& info,
46                         TreeNode<INFO_T>* parent =0, // Ignored
47                         bool preferRight =false, // Ignored

```

```

48         int replaceBehavior = 0 ) { // Ignored
49         if( S::find( this->root( ), info ) )
50             return 0;
51         node_t *node = new node_t( );
52         S::insertInto( info, node );
53         rebalance( node );
54         return node;
55     }
56
57     /**
58     * @function      remove( )
59     * @abstract      A node is removed in such a way that the properties of
60     *                an AVL tree remain intact.
61     * @param         node - the node we're going to remove
62     * @post          The node has been removed, but the remaining tree still
63     *                contains all of its other nodes and still has all the
64     *                AVL properties.
65     */
66     void remove( node_t* node ) {
67         // if it's a leaf
68         if( !node->leftChild( ) && !node->rightChild( ) )
69             S::remove( node );
70         // internal node with kids
71         else {
72             if( node->rightChild( ) ) {
73                 node = static_cast<node_t*>( S::replace(
74                     S::min( static_cast<node_t*>(
75                         node->rightChild( ) )->info( ), node ) );
76                 removeMin( static_cast<node_t*>( node->rightChild( ) ) );
77                 node->setRightChild( node->rightChild( ) );
78             }
79             else
80                 // just delete the node and replace it with its leftChild
81                 node->replace( node->leftChild( ) );
82         }
83     }
84
85     private:
86
87     /**
88     * @function      removeMin( )
89     * @abstract      Recursively we go through the tree to find the node with
90     *                the smallest value in the subtree with root node. Then we
91     *                restore the balance factors of all its parents.
92     * @param         node - the root of the subtree we're looking in
93     * @return        At the end of the recursion we return the parent of the
94     *                node with the smallest value. Then we go up the tree and
95     *                rebalance every parent from this upwards.
96     * @post          The node with the smallest value is deleted and every
97     *                node still has the correct balance factor.
98     */
99     node_t* removeMin( node_t* node ) {
100         node_t* temp;
101         if( node->leftChild( ) )

```

```

102         temp =removeMin( static_cast<node_t*>( node->leftChild( ) ) );
103     else {
104         temp =static_cast<node_t*>( node->parent( ) );
105         S::remove( node );
106     }
107     rebalance( temp );
108     return temp;
109 }
110
111 /**
112  * @function      removeMax( )
113  * @abstract      Recursively we go through the tree to find the node with
114  *                the highest value in the subtree with root node. Then we
115  *                restore the balance factors of all its parents.
116  * @param         node - the root of the subtree we're looking in
117  * @return        At the end of the recursion we return the parent of the
118  *                node with the highest value. Then we go up the tree and
119  *                rebalance every parent from this upwards.
120  * @post          The node with the highest value is deleted and every
121  *                node still has the correct balance factor.
122  */
123 node_t* removeMax( node_t* node ) {
124     node_t* temp;
125     if( node->rightChild( ) )
126         temp =removeMin( static_cast<node_t*>( node->rightChild( ) ) );
127     else {
128         temp =static_cast<node_t*>( node->parent( ) );
129         S::remove( node );
130     }
131     rebalance( temp );
132     return temp;
133 }
134
135 /**
136  * @function      rotateLeft( )
137  * @abstract      We rotate a node left and make sure all the internal
138  *                heights of the nodes are up to date.
139  * @param         node - the node we're going to rotate left
140  * @return        we return the node that is now at the top of this
141  *                particular subtree.
142  * @post          The node is rotated to the left and the heights are up
143  *                to date.
144  */
145 node_t* rotateLeft( node_t* node ) {
146     node_t *temp =static_cast<node_t*>( S::rotateLeft( node ) );
147     temp->updateHeight( );
148     if( temp->leftChild( ) )
149         static_cast<node_t*>( temp->leftChild( ) )->updateHeight( );
150     return temp;
151 }
152
153 /**
154  * @function      rotateRight( )
155  * @abstract      We rotate a node right and make sure all the internal

```

```

156      *           heights of the nodes are up to date.
157      * @param      node - the node we're going to rotate right
158      * @return      we return the node that is now at the top of this
159      *               particular subtree.
160      * @post         The node is rotated to the right and the heights are up
161      *               to date.
162      **/
163      node_t* rotateRight( node_t* node ) {
164          node_t* temp =static_cast<node_t*>( S::rotateRight( node ) );
165          temp->updateHeight( );
166          if( temp->rightChild( ) )
167              static_cast<node_t*>( temp->rightChild( ) )->updateHeight( );
168          return temp;
169      }
170
171      /**
172      * @function      rebalance( )
173      * @abstract      The tree is rebalanced. We do the necessary rotations
174      *               from the bottom up to make sure the AVL properties are
175      *               still intact.
176      * @param          node - the node we're going to rebalance
177      * @post           The tree is now perfectly balanced.
178      **/
179      void rebalance( node_t* node ) {
180          node->updateHeight( );
181
182          node_t* temp =node;
183          while( temp->parent( ) ) {
184              temp =static_cast<node_t*>( temp->parent( ) );
185              temp->updateHeight( );
186              // right subtree too deep
187              if( temp->balanceFactor( ) == 2 ) {
188                  if( temp->rightChild( ) ) {
189                      if( static_cast<node_t*>( temp->rightChild( ) )
190                          ->balanceFactor( ) < 0 )
191                          this->rotateRight(
192                              static_cast<node_t*>( temp->rightChild( ) ) );
193                  }
194                  this->rotateLeft( temp );
195              }
196              // left subtree too deep
197              else if( temp->balanceFactor( ) == -2 ) {
198                  if( temp->leftChild( ) ) {
199                      if( static_cast<node_t*>( temp->leftChild( ) )->
200                          balanceFactor( ) > 0 )
201                          this->rotateLeft(
202                              static_cast<node_t*>( temp->leftChild( ) ) );
203                  }
204                  this->rotateRight( temp );
205              }
206          }
207      }
208  };
209

```

210 **#endif**

## 6.10 AVLNode.h

```
1  /**
2   * AVLNode - Node atom type for AVLTree
3   *
4   * @author Micky Faas (s1407937)
5   * @author Lisette de Schipper (s1396250)
6   * @file AVLNode.h
7   * @date 9-11-2014
8   */
9
10 #ifndef AVLNODE_H
11 #define AVLNODE_H
12
13 #include "BSTNode.h"
14
15 template <class INFO_T> class AVLTree;
16
17 template <class INFO_T> class AVLNode : public BSTNode<INFO_T>
18 {
19     public:
20         typedef BSTNode<INFO_T> S; // super class
21
22         /**
23          * @function AVLNode( )
24          * @abstract Constructor, creates a node
25          * @param info, the contents of a node
26          * @param parent, the parent of the node
27          * @post A node has been created.
28          */
29         AVLNode( const INFO_T& info, AVLNode<INFO_T>* parent =0 )
30             : S( info, parent ) {
31         }
32
33         /**
34          * @function AVLNode( )
35          * @abstract Constructor, creates a node
36          * @param parent, the parent of the node
37          * @post A node has been created.
38          */
39         AVLNode( AVLNode<INFO_T>* parent =0 )
40             : S( (S)parent ) {
41         }
42
43         /**
44          * @function balanceFactor( )
45          * @abstract we return the height of the rightchild subtracted with
46          * the height of the left child. Because of the properties
47          * of an AVLtree, this should never be less than -1 or more
48          * than 1.
49          * @return we return the difference between the height of the
50          * rightchild and the leftchild.
```

```

51      * @post          The difference between the two child nodes is returned.
52      **/
53      int balanceFactor( ){
54          return static_cast<AVLNode *>( this->rightChild( ) )->getHeight( ) -
55                 static_cast<AVLNode *>( this->leftChild( ) )->getHeight( );
56      }
57
58      /**
59      * @function        updateHeight( )
60      * @abstract        we update the height of the node.
61      * @pre             The children of the node need to have the correct height.
62      * @post            The node now has the right height.
63      **/
64      void updateHeight( ) {
65          int lHeight =static_cast<AVLNode *>( this->leftChild( ) )
66                      ->getHeight( );
67          int rHeight =static_cast<AVLNode *>( this->rightChild( ) )
68                      ->getHeight( );
69
70          this->height =( 1 + ( ( lHeight > rHeight ) ? lHeight : rHeight ) );
71      }
72
73      /**
74      * @function        getHeight( )
75      * @abstract        we want to know the height of the node.
76      * @return          we return the height of the node.
77      * @post            The current height of the node is returned.
78      **/
79      int getHeight( ) {
80          return (this ? this->height : 0);
81      }
82
83      bool operator <( const AVLNode<INFO_T> &rhs ) {
84          return S::info() < rhs.info();
85      }
86
87      bool operator <=( const AVLNode<INFO_T> &rhs ) {
88          return S::info() <= rhs.info();
89      }
90
91      bool operator >( const AVLNode<INFO_T> &rhs ) {
92          return S::info() > rhs.info();
93      }
94
95      bool operator >=( const AVLNode<INFO_T> &rhs ) {
96          return S::info() >= rhs.info();
97      }
98
99      protected:
100         friend class AVLTree<INFO_T>;
101
102      private:
103         int height;
104    };

```



```

105
106
107 #endif

```

## 6.11 SplayTree.h

```

1  /**
2   * SplayTree - Splay-tree implementation
3   *
4   * @author Micky Faas (s1407937)
5   * @author Lisette de Schipper (s1396250)
6   * @file SplayTree.h
7   * @date 3-11-2014
8   */
9
10 #ifndef SPLAYTREE_H
11 #define SPLAYTREE_H
12
13 #include "SelfOrganizingTree.h"
14
15 template <class INFO_T> class SplayTree : public SelfOrganizingTree<INFO_T> {
16 public:
17     typedef BSTNode<INFO_T> node_t;
18     typedef SelfOrganizingTree<INFO_T> S; // super class
19
20     SplayTree( ) : SelfOrganizingTree<INFO_T>( ) { }
21
22     SplayTree( const SplayTree& copy )
23         : SelfOrganizingTree<INFO_T>( copy ) { }
24
25     /**
26     * @function insert( )
27     * @abstract reimplemented virtual function from BinarySearchTree<>
28     * the new node will always be the root
29     * @param info, the contents of the new node
30     * @param parent, ignored
31     * @param preferRight, ignored
32     * @param replaceBehavior, ignored
33     * @return returns a pointer to the inserted node (root)
34     */
35     virtual node_t* insert( const INFO_T& info,
36                             TreeNode<INFO_T>* parent =0, // Ignored
37                             bool preferRight =false, // Ignored
38                             int replaceBehavior =0 ) { // Ignored
39         return splay( S::insert( info, parent, preferRight ) );
40     }
41
42     /**
43     * @function replace( )
44     * @abstract reimplemented virtual function from BinarySearchTree<>
45     * replaces a given node or the root
46     * the resulting node will be propagated to location of the root
47     * @param info, the contents of the new node
48     * @param node, node to be replaced

```

```

49      * @param      alignRight, ignored
50      * @param      replaceBehavior, ignored
51      * @return      returns a pointer to the new node (=root)
52      * @pre         node should be in this tree
53      * @post        replace() will delete and/or remove node.
54      *              if node is 0, it will take the root instead
55      **/
56      virtual node_t* replace( const INFO_T& info,
57                              TreeNode<INFO_T>* node =0,
58                              bool alignRight =false,
59                              int replaceBehavior =0 ) {
60          return splay( S::replace( info, node, alignRight ) );
61      }
62
63      /**
64      * @function      remove( )
65      * @abstract      reimplemented virtual function from BinarySearchTree<>
66      *                removes a given node or the root and restores the
67      *                BST properties. The node-to-be-removed will be splayed
68      *                before removal.
69      * @param         node, node to be removed
70      * @pre           node should be in this tree
71      * @post          memory for node will be deallocated
72      **/
73      virtual void remove( TreeNode<INFO_T> *node ) {
74          S::remove( splay( static_cast<node_t*>(node) ) );
75      }
76
77      /**
78      * @function      find( )
79      * @abstract      reimplemented virtual function from Tree<>
80      *                performs a binary search in a given (sub)tree
81      *                splays the node (if found) afterwards
82      * @param         haystack, the subtree to search. Give 0 for the entire tree
83      * @param         needle, key/info-value to find
84      * @return        returns a pointer to node, if found
85      * @pre           haystack should be in this tree
86      * @post          may return 0, the structure of the tree may change
87      **/
88      virtual TreeNode<INFO_T>* find( TreeNode<INFO_T>* haystack,
89                                      const INFO_T& needle ) {
90          return splay( static_cast<node_t*>( S::find( haystack, needle ) ) );
91      }
92
93      /**
94      * @function      splay( )
95      * @abstract      Performs the splay operation on a given node.
96      *                'Splay' means a certain amount of rotations in order
97      *                to make the given node be the root of the tree while
98      *                maintaining the binary search tree properties.
99      * @param         node, the node to splay
100      * @pre           The node must be a node in this tree
101      * @post          The node will be the new root of the tree
102      *               No nodes will be invalidated and no new memory is

```

```

103      *          allocated. Iterators may become invalid.
104      **/
105      node_t* splay( node_t* node ) {
106
107          enum MODE {
108              LEFT =0x1, RIGHT =0x2,
109              PLEFT =0x4, PRIGHT =0x8 };
110
111          // Can't splay the root (or null)
112          if( !node || S::m_root == node )
113              return node;
114
115          node_t *p =static_cast<node_t*>( node->parent( ) );
116          int mode;
117
118          while( p != S::m_root ) {
119              if( p->leftChild( ) == node )
120                  mode =RIGHT;
121              else
122                  mode =LEFT;
123
124              assert( p->parent( ) != nullptr );
125
126              // Node's grandparent
127              node_t* g =static_cast<node_t*>( p->parent( ) );
128
129              if( g->leftChild( ) == p )
130                  mode |= PRIGHT;
131              else
132                  mode |= PLEFT;
133
134              // True if either mode is LEFT|PLEFT or RIGHT|PRIGHT
135              if( (mode >> 2) == (mode & 0x3 ) ) {
136                  // the 'zig-zig' step
137                  // first rotate g-p then p-node
138
139                  if( mode & PLEFT )
140                      this->rotateLeft( g );
141                  else
142                      this->rotateRight( g );
143
144                  if( mode & LEFT )
145                      this->rotateLeft( p );
146                  else
147                      this->rotateRight( p );
148              }
149              else {
150                  // the 'zig-zag' step
151                  // first rotate p-node then g-p
152
153                  if( mode & LEFT )
154                      this->rotateLeft( p );
155                  else
156                      this->rotateRight( p );

```

```

157
158         if( mode & PLEFT )
159             this->rotateLeft( g );
160         else
161             this->rotateRight( g );
162     }
163
164     // perhaps we're done already...
165     if( node == this->root( ) )
166         return node;
167     else
168         p =static_cast<node_t*>( node->parent( ) );
169     }
170
171     // The 'zig-step': parent of node is the root
172
173     if( p->leftChild( ) == node )
174         this->rotateRight( p );
175     else
176         this->rotateLeft( p );
177
178     return node;
179 }
180 };
181
182 #endif

```

## 6.12 Treap.h

```

1  /**
2   * Treap - Treap that inherits from SelfOrganizingTree
3   *
4   * @author Micky Faas (s1407937)
5   * @author Lisette de Schipper (s1396250)
6   * @file Treap.h
7   * @date 9-12-2014
8   */
9
10 #ifndef TREAP_H
11 #define TREAP_H
12
13 #include "SelfOrganizingTree.h"
14 #include "TreapNode.h"
15
16 template <class INFO_T> class Treap : public SelfOrganizingTree<INFO_T> {
17     public:
18         typedef TreapNode<INFO_T> node_t;
19         typedef SelfOrganizingTree<INFO_T> S; // super class
20
21     /**
22      * @function Treap( )
23      * @abstract constructor
24      * @post A Treap is created
25      */

```

```

26     Treap( int randomRange =100 ) : S( ) {
27         random =randomRange;
28         srand( time( NULL ) );
29     }
30
31 /**
32  * @function      Treap( )
33  * @abstract      constructor
34  * @param         cpy
35  * @post          A Treap is created
36  */
37     Treap( const Treap& cpy, int randomRange =100 ) : S( cpy ) {
38         random =randomRange;
39         srand( time( NULL ) );
40     }
41
42 /**
43  * @function      insert( )
44  * @abstract      A node with label 'info' is inserted into the tree and
45  *               put in the right place. A label may not appear twice in
46  *               a tree.
47  * @param         info - the label of the node
48  * @return        the node we inserted
49  * @post          The tree now contains a node with 'info'
50  */
51     node_t* insert( const INFO_T& info,
52                     TreeNode<INFO_T>* parent =0, // Ignored
53                     bool preferRight =false,    // Ignored
54                     int replaceBehavior =0 ) { // Ignored
55         // Prevent duplicates
56
57         if( S::find( this->root( ), info ) )
58             return 0;
59         node_t *node =new node_t( );
60         S::insertInto( info, node );
61         node->priority =rand( ) % random + 1;
62         rebalance( node );
63
64         return node;
65     }
66
67 /**
68  * @function      remove( )
69  * @abstract      the node provided with the parameter is deleted from the
70  *               tree by rotating it down until it becomes a leaf or has
71  *               only one child. In the first case it's just deleted,
72  *               in the second it's replaced by its subtree.
73  * @param         node - the node to be deleted
74  * @post          The node is deleted from the tree which still retains
75  *               the Treap properties.
76  */
77     void remove( node_t* node ) {
78         node_t *temp = node;
79         // rotating it down until the condition no longer applies.

```

```

80         while( temp->leftChild( ) && temp->rightChild( ) )
81         {
82             if( static_cast<node_t*>( temp->rightChild( ) )->priority >
83                 static_cast<node_t*>( temp->leftChild( ) )->priority )
84                 this->rotateLeft( temp );
85             else
86                 this->rotateRight( temp );
87         }
88         // if it's a leaf
89         if( !temp->leftChild( ) && !temp->rightChild( ) )
90             S::remove( temp );
91         // if it only has a right child
92         else if( !temp->leftChild( ) )
93             temp->replace( static_cast<node_t*>( temp->rightChild( ) ) );
94         // if it only has a left child
95         else if( !temp->rightChild( ) )
96             temp->replace( static_cast<node_t*>( temp->leftChild( ) ) );
97     }
98
99     private:
100         int random;
101
102         /**
103          * @function      rebalance( )
104          * @abstract      The tree is rebalanced. We do the necessary rotations
105          *                  from the bottom up to make sure the Treap properties are
106          *                  still intact.
107          * @param          info - the label of the node
108          * @return         the node we inserted
109          * @post           The tree is now perfectly balanced.
110          */
111         void rebalance( node_t* node ) {
112             if( !node )
113                 return;
114             node_t* temp = node;
115             int myPriority = node->priority;
116             while( temp->parent( ) &&
117                 myPriority >
118                 static_cast<node_t*>( temp->parent( ) )->priority ) {
119                 temp = static_cast<node_t*>( temp->parent( ) );
120                 if( temp->leftChild( ) == node )
121                     this->rotateRight( temp );
122                 else
123                     this->rotateLeft( temp );
124             }
125         }
126
127     };
128
129 #endif

```

## 6.13 TreapNode.h

```

1  /**

```

```

2  * TreapNode - Node atom type for Treap
3  *
4  * @author Micky Faas (s1407937)
5  * @author Lisette de Schipper (s1396250)
6  * @file TreapNode.h
7  * @date 9-11-2014
8  **/
9
10 #ifndef TREAPNODE_H
11 #define TREAPNODE_H
12
13 #include "BSTNode.h"
14
15 template <class INFO_T> class Treap;
16
17 template <class INFO_T> class TreapNode : public BSTNode<INFO_T>
18 {
19     public:
20         typedef BSTNode<INFO_T> S; // super class
21
22         /**
23          * @function TreapNode( )
24          * @abstract Constructor, creates a node
25          * @param info, the contents of a node
26          * @param parent, the parent of the node
27          * @post A node has been created.
28          **/
29         TreapNode( const INFO_T& info, TreapNode<INFO_T>* parent =0 )
30             : S( info, parent ), priority( 0 ) {
31         }
32
33         /**
34          * @function TreapNode( )
35          * @abstract Constructor, creates a node
36          * @param parent, the parent of the node
37          * @post A node has been created.
38          **/
39         TreapNode( TreapNode<INFO_T>* parent =0 )
40             : S( (S)parent ), priority( 0 ) {
41         }
42
43         /**
44          * @function replace( )
45          * @abstract Replaces the node with another node in the tree
46          * @param n, the node we replace the node with, this one gets deleted
47          * @pre both this node and n must be in the same parent tree
48          * @post The node will be replaced and n will be deleted.
49          **/
50         void replace( TreapNode<INFO_T>* n ) {
51             priority = n->priority;
52             this->S::replace( n );
53         }
54
55         bool operator <( const TreapNode<INFO_T> &rhs ) {

```

```

56         return S::info() < rhs.info();
57     }
58
59     bool operator <=( const TreapNode<INFO_T> &rhs ) {
60         return S::info() <= rhs.info();
61     }
62
63     bool operator >( const TreapNode<INFO_T> &rhs ) {
64         return S::info() > rhs.info();
65     }
66
67     bool operator >=( const TreapNode<INFO_T> &rhs ) {
68         return S::info() >= rhs.info();
69     }
70
71     int priority;
72
73     protected:
74         friend class Treap<INFO_T>;
75 };
76
77
78 #endif

```