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($logn$).

2.1 Implementatie binaire zoekboom

De binairy 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:

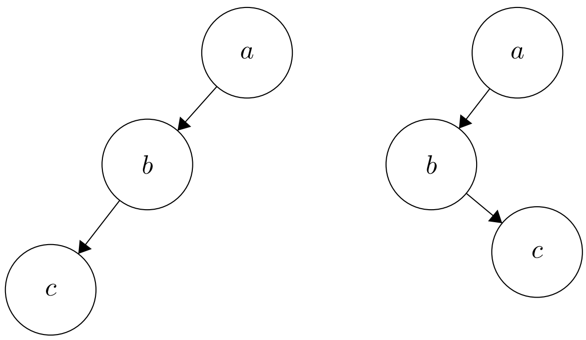
- Linker-kindknopen hebben altijd een kleinere waarde dan hun ouder, rechter-kindknopen altijd een grotere.
- Bij een MIN-boom is dit omgekeerd. Onze implementatie is enkel een MAX-boom.
- Toevoegen kan zonder verwisselen worden uitgevoerd (in tegenstelling tot bijv. een heap).
- Voor verwijderen of 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($logn$) 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($logn$)) en vervolgens moet de boom op basis van de toevoeging geherstructureerd worden. Dit laatste is in het ergste geval O($logn$), 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($logn$). 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 herorganiseerd na elke operatie, ook na operaties die alleen lezen, zoals `find()`. Deze herorganisatiestap 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 uitvoerd blijft bovendien de binairy-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 definieren:

- 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($logn$) 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 alsvolgt:

```
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 geheugenvoetafdruk 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 voldoede om verschillen te kunnen zien. We hebben een aantal testcondities gebruikt:

- Voor het inladen een wel of niet alfabetisch gesoorteerd 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 gevariëerd.

3.5 Hypothesen

- De binairy 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

```
/**
 * main.cc:
 *
 * @author  Micky Faas (s1407937)
 * @author  Lisette de Schipper (s1396250)
 * @file    main.cc
 * @date    26-10-2014
 **/

#include <iostream>
#include "BinarySearchTree.h"
#include "Tree.h"
#include "AVLTree.h"
#include "SplayTree.h"
#include "Treap.h"
#include <string>

using namespace std;

// Makkelijk voor debuggen, moet nog beter
template<class T> void printTree( Tree<T> tree, int rows ) {
    typename Tree<T>::nodelist list =tree.row( 0 );
    int row =0;
    while( !list.empty( ) && row < rows ) {
        string offset;
        for( int i =0; i < ( 1 << (rows - row) ) - 1 ; ++i )
```

```
            offset += ' ';


        for( auto it =list.begin( ); it != list.end( ); ++it ) {
            if( *it )
                cout << offset << (*it)->info() << " " << offset;
            else
                cout << offset << ". " << offset;
        }
        cout << endl;
        row++;
        list =tree.row( row );
    }
}

int main ( int argc, char **argv ) {

    /* BST hieronder */

    cout << "BST:" << endl;
    BinarySearchTree<int> bst;

   /* auto root =bst.pushBack( 10 );
    bst.pushBack( 5 );
    bst.pushBack( 15 );

    bst.pushBack( 25 );
    bst.pushBack( 1 );
    bst.pushBack( -1 );
    bst.pushBack( 11 );
    bst.pushBack( 12 );*/

    Tree<int>* bstP =&bst; // Dit werkt gewoon :-)

    auto root =bstP->pushBack( 10 );
    bstP->pushBack( 5 );
    bstP->pushBack( 15 );

    bstP->pushBack( 25 );
    bstP->pushBack( 1 );
    bstP->pushBack( -1 );
    bstP->pushBack( 11 );
    bstP->pushBack( 12 );

    //printTree<int>( bst, 5 );


    //bst.remove( bst.find( 0, 15 ) );
    //bst.replace( -2, bst.find( 0, 5 ) );


    printTree<int>( bst, 5 );

    bst.remove( root );
```

```
    printTree<int>( bst, 5 );

    /* Splay Trees hieronder */

    cout << "Splay Boom:" << endl;
    SplayTree<int> splay;

    splay.pushBack( 10 );
    auto a =splay.pushBack( 5 );
    splay.pushBack( 15 );

    splay.pushBack( 25 );
    auto b =splay.pushBack( 1 );
    splay.pushBack( -1 );
    auto c =splay.pushBack( 11 );
    splay.pushBack( 12 );

    //printTree<int>( splay, 5 );

    //a->swapWith( b );
    //splay.remove( splay.find( 0, 15 ) );
    //splay.replace( -2, splay.find( 0, 5 ) );


    printTree<int>( splay, 5 );

    //splay.remove( root );

    splay.splay( c );

    printTree<int>( splay, 5 );

    // Test AVLTree //

    AVLTree<char> test;
    test.insert( 'a' );
    auto d =test.insert( 'b' );
    test.insert( 'c' );
    test.insert( 'd' );
    test.insert( 'e' );
    test.insert( 'f' );
    test.insert( 'g' );
    cout << "AVL Boompje:" << endl;
    printTree<char>( test, 5 );
    cout << d->info( ) << " verwijderen: " << endl;
    test.remove( d );
    printTree<char>( test, 5 );

    // Test Treap //

    cout << "Treap" << endl;

```

```
    Treap<int> testTreap(5);
    testTreap.insert(2);
    testTreap.insert(3);
    auto e =testTreap.insert(4);
    testTreap.insert(5);
    printTree<int>( testTreap, 5 );
    testTreap.remove(e);
    printTree<int>( testTreap, 5 );

    return 0;
}
```

6.2 hooiberg.cc

```
/**
 * hooiberg.cc:
 *
 * @author  Micky Faas (s1407937)
 * @author  Lisette de Schipper (s1396250)
 * @file    helehogebomen.cc
 * @date    10-12-2014
 **/

#include "BinarySearchTree.h"
#include "Tree.h"
#include "AVLTree.h"
#include "SplayTree.h"
#include "Treap.h"

#include <iostream>
#include <string>
#include <fstream>
#include <vector>
#include <chrono>

// Only works on *nix operating systems
// Needed for precision timing
#include <sys/time.h>

using namespace std;

// Makkelijk voor debuggen, moet nog beter
template<class T> void printTree( Tree<T> tree, int rows ) {
    typename Tree<T>::nodelist list =tree.row( 0 );
    int row =0;
    while( !list.empty( ) && row < rows ) {
        string offset;
        for( int i =0; i < ( 1 << (rows - row) ) - 1 ; ++i )
            offset += ' ';


        for( auto it =list.begin( ); it != list.end( ); ++it ) {
            if( *it )
                cout << offset << (*it)->info() << " " << offset;
```

```cpp
            else
                cout << offset << ". " << offset;
        }
        cout << endl;
        row++;
        list =tree.row( row );
    }
}

int printUsage( const char* prog ) {

    std::cout << "Reads an input file and searches it for a set of strings\n\n"
         << "Usage: " << prog << " [type] [haystack] [needles]\n"
         << "\t[type]\t\tTree type to use. One of ‘splay’, ‘avl’, ‘treap’, ‘bst’\n"
         << "\t[haystack]\tInput file, delimited by newlines\n"
         << "\t[needles]\tFile containing sets of strings to search for, delimited by
         << "\t[random]\tOptimal customization of Treap\n"
         << std::endl;
    return 0;
}

bool extractNeedles( std::vector<string> &list, std::ifstream &file ) {
    string needle;
    while( !file.eof( ) ) {
        std::getline( file, needle );
        if( needle.size( ) )
            list.push_back( needle );
    }
    return true;
}

bool fillTree( BinarySearchTree<string>* tree, std::ifstream &file ) {
    string word;
    while( !file.eof( ) ) {
        std::getline( file, word );
        if( word.size( ) )
            tree->pushBack( word );
    }
    return true;
}

void findAll( std::vector<string> &list, BinarySearchTree<string>* tree ) {
    int steps =0, found =0, notfound =0;
    for( auto needle : list ) {
        if( tree->find( 0, needle ) ) {
            found++;
            steps +=tree->lastSearchStepCount( );
            if( found < 51 )
                std::cout << "Found ‘" << needle << '\''
                << " in " << tree->lastSearchStepCount( ) << " steps." << std::endl;
        }
        else if( ++notfound < 51 )
            std::cout << "Didn't find ‘" << needle << '\'' << std::endl;
    }
```

```cpp
    if( found > 50 )
        std::cout << found - 50 << " more results not shown here." << std::endl;
    if( found )
        cout << "Total search depth:           " << steps << endl
             << "Number of matches:            " << found << endl
             << "Number of misses:             " << notfound << endl
             << "Average search depth (hits): " << steps/found << endl;
}

int main ( int argc, char **argv ) {

    enum MODE { NONE =0, BST, AVL, SPLAY, TREAP };
    int mode =NONE;

    if( argc < 4 )
        return printUsage( argv[0] );

    if( std::string( argv[1] ) == "bst" )
        mode =BST;
    else if( std::string( argv[1] ) == "avl" )
        mode =AVL;
    else if( std::string( argv[1] ) == "treap" )
        mode =TREAP;
    if( std::string( argv[1] ) == "splay" )
        mode =SPLAY;

    if( !mode )
        return printUsage( argv[0] );

    std::ifstream fhaystack( argv[2] );
    if( !fhaystack.good( ) ) {
        std::cerr << "Could not open " << argv[2] << std::endl;
        return -1;
    }

    std::ifstream fneedles( argv[3] );
    if( !fneedles.good( ) ) {
        std::cerr << "Could not open " << argv[3] << std::endl;
        return -1;
    }

    if( argv[4] && mode != TREAP ) {
        std::cerr << "This variable should only be set for Treaps." << std::endl;
        return -1;
    }
    else if( atoi(argv[4]) <= 0 ) {
        std::cerr << "This variable should only be an integer "
                  << " greater than 0." << std::endl;
        return -1;
    }

    std::vector<string> needles;
    if( !extractNeedles( needles, fneedles ) ) {
        cerr << "Could not read a set of strings to search for." << endl;
```

```
        return -1;
    }

    BinarySearchTree<string> *tree;
    switch( mode ) {
        case BST:
            tree = new BinarySearchTree<string>();
            break;
        case AVL:
            tree = new AVLTree<string>();
            break;
        case SPLAY:
            tree = new SplayTree<string>();
            break;
        case TREAP:
            tree = new Treap<string>(atoi(argv[4]));
            break;
    }


    // Define a start point to time measurement
    auto start = std::chrono::system_clock::now();


    if( !fillTree( tree, fhaystack ) ) {
        cerr << "Could not read the haystack." << endl;
        return -1;
    }

    // Determine the duration of the code block
    auto duration =std::chrono::duration_cast<std::chrono::milliseconds>
                            (std::chrono::system_clock::now() - start);

    cout << "Filled the binary search tree in " << duration.count() << "ms" << endl;

    start = std::chrono::system_clock::now();
    findAll( needles, tree );
    duration =std::chrono::duration_cast<std::chrono::milliseconds>
                            (std::chrono::system_clock::now() - start);

    cout << "Searched the haystack in " << duration.count() << "ms" << endl;

    // Test pre-order
    //for( auto word : *tree ) {
    //    cout << word << '\n';
    //}

    fhaystack.close( );
    fneedles.close( );
    delete tree;

    return 0;
}
```

6.3 Tree.h

```
/**
 * Tree:
 *
 * @author  Micky Faas (s1407937)
 * @author  Lisette de Schipper (s1396250)
 * @file    tree.h
 * @date    26-10-2014
 **/

#ifndef TREE_H
#define TREE_H
#include "TreeNodeIterator.h"
#include <assert.h>
#include <list>
#include <map>

using namespace std;

template <class INFO_T> class SplayTree;

template <class INFO_T> class Tree
{
    public:
        enum ReplaceBehavoir {
            DELETE_EXISTING,
            ABORT_ON_EXISTING,
            MOVE_EXISTING
        };

        typedef TreeNode<INFO_T> node_t;
        typedef TreeNodeIterator<INFO_T> iterator;
        typedef TreeNodeIterator_in<INFO_T> iterator_in;
        typedef TreeNodeIterator_pre<INFO_T> iterator_pre;
        typedef TreeNodeIterator_post<INFO_T> iterator_post;
        typedef list<node_t*> nodelist;

       /**
        * @function  Tree( )
        * @abstract  Constructor of an empty tree
        **/
        Tree( )
            : m_root( 0 ) {
        }

       /**
        * @function  Tree( )
        * @abstract  Copy-constructor of a tree. The new tree contains the nodes
        *            from the tree given in the parameter (deep copy)
        * @param     tree, a tree
        **/
        Tree( const Tree<INFO_T>& tree )
            : m_root( 0 ) {
```

```
            *this =tree;
        }

        /**
        * @function  ~Tree( )
        * @abstract  Destructor of a tree. Timber.
        **/
        ~Tree( ) {
          clear( );
        }

       /**
        * @function  begin_pre( )
        * @abstract  begin point for pre-order iteration
        * @return    interator_pre containing the beginning of the tree in
        *            pre-order
        **/
        iterator_pre begin_pre( ) {
            // Pre-order traversal starts at the root
            return iterator_pre( m_root );
          }

       /**
        * @function  begin( )
        * @abstract  begin point for a pre-order iteration
        * @return    containing the beginning of the pre-Order iteration
        **/
        iterator_pre begin( ) {
            return begin_pre( );
        }

       /**
        * @function  end( )
        * @abstract  end point for a pre-order iteration
        * @return    the end of the pre-order iteration
        **/
        iterator_pre end( ) {
            return iterator_pre( (node_t*)0 );
        }

       /**
        * @function  end_pre( )
        * @abstract  end point for pre-order iteration
        * @return    interator_pre containing the end of the tree in pre-order
        **/
        iterator_pre end_pre( ) {
            return iterator_pre( (node_t*)0 );
        }

       /**
        * @function  begin_in( )
        * @abstract  begin point for in-order iteration
        * @return    interator_in containing the beginning of the tree in
        *            in-order
```

```
        **/
        iterator_in begin_in( ) {
            if( !m_root )
                return end_in( );
            node_t *n =m_root;
            while( n->leftChild( ) )
                n =n->leftChild( );
            return iterator_in( n );
          }

        /**
        * @function  end_in( )
        * @abstract  end point for in-order iteration
        * @return    interator_in containing the end of the tree in in-order
        **/
        iterator_in end_in( ) {
            return iterator_in( (node_t*)0 );
        }

        /**
        * @function  begin_post( )
        * @abstract  begin point for post-order iteration
        * @return    interator_post containing the beginning of the tree in
        *            post-order
        **/
        iterator_post begin_post( ) {
            if( !m_root )
                return end_post( );
            node_t *n =m_root;
            while( n->leftChild( ) )
                n =n->leftChild( );
            return iterator_post( n );
        }

        /**
        * @function  end_post( )
        * @abstract  end point for post-order iteration
        * @return    interator_post containing the end of the tree in post-order
        **/
        iterator_post end_post( ) {
            return iterator_post( (node_t*)0 );
        }

        /**
        * @function  pushBack( )
        * @abstract  a new TreeNode containing `info` is added to the end
        *            the node is added to the node that :
        *               - is in the row as close to the root as possible
        *               - has no children or only a left-child
        *               - seen from the right hand side of the row
        *            this is the `natural` left-to-right filling order
        *            compatible with array-based heaps and full b-trees
        * @param     info, the contents of the new node
        * @post      A node has been added.
```

```
        **/
        virtual node_t *pushBack( const INFO_T& info ) {
            node_t *n = new node_t( info, 0 );
            if( !m_root ) { // Empty tree, simplest case
                m_root = n;
            }
            else { // Leaf node, there are two different scenarios
                int max = getRowCountRecursive( m_root, 0 );
                node_t *parent;
                for( int i = 1; i <= max; ++i ) {

                    parent = getFirstEmptySlot( i );
                    if( parent ) {
                        if( !parent->leftChild( ) )
                            parent->setLeftChild( n );
                        else if( !parent->rightChild( ) )
                            parent->setRightChild( n );
                        n->setParent( parent );
                        break;
                    }
                }
            }
            return n;
        }

       /**
        * @function  insert( )
        * @abstract  inserts node or subtree under a parent or creates an empty
        *            root node
        * @param     info, contents of the new node
        * @param     parent, parent node of the new node. When zero, the root is
        *            assumed
        * @param     alignRight, insert() checks on which side of the parent
        *            node the new node can be inserted. By default, it checks
        *            the left side first.
        *            To change this behavior, set preferRight = true.
        * @param     replaceBehavior, action if parent already has two children.
        *            One of:
        *            ABORT_ON_EXISTING - abort and return zero
        *            MOVE_EXISTING - make the parent's child a child of the new
        *                            node, satisfies preferRight
        *            DELETE_EXISTING - remove one of the children of parent
        *                              completely also satisfies preferRight
        * @return    pointer to the inserted TreeNode, if insertion was
        *            successfull
        * @pre       If the tree is empty, a root node will be created with info
        *            as it contents
        * @pre       The instance pointed to by parent should be part of the
        *            called instance of Tree
        * @post      Return zero if no node was created. Ownership is assumed on
        *            the new node.
        *            When DELETE_EXISTING is specified, the entire subtree on
        *            preferred side may be deleted first.
        **/
```

```
        virtual node_t* insert( const INFO_T& info,
                        node_t* parent =0,
                        bool preferRight =false,
                        int replaceBehavior =ABORT_ON_EXISTING ) {
            if( !parent )
                parent =m_root;

            if( !parent )
                return pushBack( info );

            node_t *node =0;

            if( !parent->leftChild( )
                  && ( !preferRight || ( preferRight &&
                        parent->rightChild( ) ) ) ) ) {
                node =new node_t( info, parent );
                parent->setLeftChild( node );
                node->setParent( parent );

            } else if( !parent->rightChild( ) ) {
                node =new node_t( info, parent );
                parent->setRightChild( node );
                node->setParent( parent );

            } else if( replaceBehavior == MOVE_EXISTING ) {
                node =new node_t( info, parent );
                if( preferRight ) {
                    node->setRightChild( parent->rightChild( ) );
                    node->rightChild( )->setParent( node );
                    parent->setRightChild( node );
                } else {
                    node->setLeftChild( parent->leftChild( ) );
                    node->leftChild( )->setParent( node );
                    parent->setLeftChild( node );
                }

            } else if( replaceBehavior == DELETE_EXISTING ) {
                node =new node_t( info, parent );
                if( preferRight ) {
                    deleteRecursive( parent->rightChild( ) );
                    parent->setRightChild( node );
                } else {
                    deleteRecursive( parent->leftChild( ) );
                    parent->setLeftChild( node );
                }

            }
            return node;
        }

       /**
        * @function  replace( )
        * @abstract  replaces an existing node with a new node
        * @param     info, contents of the new node
```

```
        * @param     node, node to be replaced. When zero, the root is assumed
        * @param     alignRight, only for MOVE_EXISTING. If true, node will be
        *            the right child of the new node. Otherwise, it will be the
        *            left.
        * @param     replaceBehavior, one of:
        *            ABORT_ON_EXISTING - undefined for replace()
        *            MOVE_EXISTING - make node a child of the new node,
        *                            satisfies preferRight
        *            DELETE_EXISTING - remove node completely
        * @return    pointer to the inserted TreeNode, replace() is always
        *            successful
        * @pre       If the tree is empty, a root node will be created with info
        *            as it contents
        * @pre       The instance pointed to by node should be part of the
        *            called instance of Tree
        * @post      Ownership is assumed on the new node. When DELETE_EXISTING
        *            is specified, the entire subtree pointed to by node is
        *            deleted first.
        **/
        virtual node_t* replace( const INFO_T& info,
                                 node_t* node =0,
                                 bool alignRight =false,
                                 int replaceBehavior =DELETE_EXISTING ) {
            assert( replaceBehavior != ABORT_ON_EXISTING );

            node_t *newnode =new node_t( info );
            if( !node )
                node =m_root;
            if( !node )
                return pushBack( info );

            if( node->parent( ) ) {
                newnode->setParent( node->parent( ) );
                if( node->parent( )->leftChild( ) == node )
                    node->parent( )->setLeftChild( newnode );
                else
                    node->parent( )->setRightChild( newnode );
            } else
                m_root =newnode;

            if( replaceBehavior == DELETE_EXISTING ) {

                deleteRecursive( node );
            }
            else if( replaceBehavior == MOVE_EXISTING ) {
                if( alignRight )
                    newnode->setRightChild( node );
                else
                    newnode->setLeftChild( node );
                node->setParent( newnode );
            }
            return node;
        }

```

```
        /**
         * @function  remove( )
         * @abstract  removes and deletes node or subtree
         * @param     n, node or subtree to be removed and deleted
         * @post      after remove(), n points to an invalid address
         **/
         virtual void remove( node_t *n ) {
             if( !n )
                 return;
             if( n->parent( ) ) {
                 if( n->parent( )->leftChild( ) == n )
                     n->parent( )->setLeftChild( 0 );
                 else if( n->parent( )->rightChild( ) == n )
                     n->parent( )->setRightChild( 0 );
             }
             deleteRecursive( n );
         }

        /**
         * @function  clear( )
         * @abstract  clears entire tree
         * @pre       tree may be empty
         * @post      all nodes and data are deallocated
         **/
         void clear( ) {
             deleteRecursive( m_root );
             m_root =0;
         }

        /**
         * @function  empty( )
         * @abstract  test if tree is empty
         * @return    true when empty
         **/
         bool isEmpty( ) const {
             return !m_root;
         }

       /**
         * @function  root( )
         * @abstract  returns address of the root of the tree
         * @return    the adress of the root of the tree is returned
         * @pre       there needs to be a tree
         **/
         node_t* root( ){
             return m_root;
         }

        /**
         * @function  row( )
         * @abstract  returns an entire row/level in the tree
         * @param     level, the desired row. Zero gives just the root.
         * @return    a list containing all node pointers in that row
         * @pre       level must be positive or zero
```

```
         * @post
         **/
        nodelist row( int level ) {
            nodelist rlist;
            getRowRecursive( m_root, rlist, level );
            return rlist;
        }

        /**
         * @function  find( )
         * @abstract  find the first occurrence of info and returns its node ptr
         * @param     haystack, the root of the (sub)tree we want to look in
         *            null if we want to start at the root of the tree
         * @param     needle, the needle in our haystack
         * @return    a pointer to the first occurrence of needle
         * @post      there may be multiple occurrences of needle, we only return
         *            one. A null-pointer is returned if no needle is found
         **/
        virtual node_t* find( node_t* haystack, const INFO_T& needle ) {
            if( haystack == 0 ) {
                    if( m_root )
                        haystack =m_root;
                    else
                        return 0;
            }
            return findRecursive( haystack, needle );
        }

        /**
         * @function  contains( )
         * @abstract  determines if a certain content (needle) is found
         * @param     haystack, the root of the (sub)tree we want to look in
         *            null if we want to start at the root of the tree
         * @param     needle, the needle in our haystack
         * @return    true if needle is found
         **/
        bool contains( node_t* haystack, const INFO_T& needle ) {
            return find( haystack, needle );
        }

        /**
         * @function  toDot( )
         * @abstract  writes tree in Dot-format to a stream
         * @param     out, ostream to write to
         * @pre       out must be a valid stream
         * @post      out (file or cout) with the tree in dot-notation
         **/
        void toDot( ostream& out, const string & graphName ) {
            if( isEmpty( ) )
                return;
            map<node_t *, int> adresses;
            typename map< node_t *, int >::iterator adrIt;
            int i =1;
            int p;
```

```
            iterator_pre it;
            iterator_pre tempit;
            adresses[m_root] =0;
            out << "digraph " << graphName << '{' << endl << '"' << 0 << '"';
            for( it =begin_pre( ); it != end_pre( ); ++it ) {
                adrIt =adresses.find( &(*it) );
                if( adrIt == adresses.end( ) ) {
                    adresses[&(*it)] =i;
                    p =i;
                    i ++;
                }
                if( (&(*it))->parent( ) != &(*tempit) )
                  out << ';' << endl << '"'
                      << adresses.find( (&(*it))->parent( ))->second << '"';
                if( (&(*it)) != m_root )
                    out << " -> \"" << p << '"';
                tempit =it;
            }
            out << ';' << endl;
            for ( adrIt =adresses.begin( ); adrIt != adresses.end( ); ++adrIt )
                out << adrIt->second << " [label=\""
                    << adrIt->first->info( ) << "\"]";
            out << '}';
        }

        /**
         * @function  copyFromNode( )
         * @abstract  copies the the node source and its children to the node
         *            dest
         * @param     source, the node and its children that need to be copied
         * @param     dest, the node who is going to get the copied children
         * @param     left, this is true if it's a left child.
         * @pre       there needs to be a tree and we can't copy to a root.
         * @post      the subtree that starts at source is now also a child of
         *            dest
         **/
        void copyFromNode( node_t *source, node_t *dest, bool left ) {
            if (!source)
                return;
            node_t *acorn =new node_t( dest );
            if(left) {
                if( dest->leftChild( ))
                    return;
                dest->setLeftChild( acorn );
            }
            else {
                if( dest->rightChild( ))
                    return;
                dest->setRightChild( acorn );
            }
            cloneRecursive( source, acorn );
        }

        Tree<INFO_T>& operator=( const Tree<INFO_T>& tree ) {
```

```
            clear( );
            if( tree.m_root ) {
                m_root = new node_t( (node_t*)0 );
                cloneRecursive( tree.m_root, m_root );
            }
            return *this;
        }

    protected:
       /**
        * @function  cloneRecursive( )
        * @abstract  cloning a subtree to a node
        * @param     source, the node we want to start the cloning process from
        * @param     dest, the node we want to clone to
        * @post      the subtree starting at source is cloned to the node dest
        **/
        void cloneRecursive( node_t *source, node_t* dest ) {
            dest->info() = source->info();
            if( source->leftChild( ) ) {
                node_t *left = new node_t( dest );
                dest->setLeftChild( left );
                cloneRecursive( source->leftChild( ), left );
            }
            if( source->rightChild( ) ) {
                node_t *right = new node_t( dest );
                dest->setRightChild( right );
                cloneRecursive( source->rightChild( ), right );
            }
        }

       /**
        * @function  deleteRecursive( )
        * @abstract  delete all nodes of a given tree
        * @param     root, starting point, is deleted last
        * @post      the subtree has been deleted
        **/
        void deleteRecursive( node_t *root ) {
            if( !root )
                return;
            deleteRecursive( root->leftChild( ) );
            deleteRecursive( root->rightChild( ) );
            delete root;
        }

       /**
        * @function  getRowCountRecursive( )
        * @abstract  calculate the maximum depth/row count in a subtree
        * @param     root, starting point
        * @param     level, starting level
        * @return    maximum depth/rows in the subtree
        **/
        int getRowCountRecursive( node_t* root, int level ) {
            if( !root )
                return level;
```

```
            return max(
                    getRowCountRecursive( root->leftChild( ), level+1 ),
                    getRowCountRecursive( root->rightChild( ), level+1 ) );
        }

       /**
        * @function  getRowRecursive( )
        * @abstract  compile a full list of one row in the tree
        * @param     root, starting point
        * @param     rlist, reference to the list so far
        * @param     level, how many level still to go
        * @post      a list of a row in the tree has been made.
        **/
        void getRowRecursive( node_t* root, nodelist &rlist, int level ) {
            // Base-case
            if( !level ) {
                rlist.push_back( root );
            } else if( root ){
                level--;
                if( level && !root->leftChild( ) )
                    for( int i =0; i < (level<<1); ++i )
                        rlist.push_back( 0 );
                else
                   getRowRecursive( root->leftChild( ), rlist, level );

                if( level && !root->rightChild( ) )
                    for( int i =0; i < (level<<1); ++i )
                        rlist.push_back( 0 );
                else
                    getRowRecursive( root->rightChild( ), rlist, level );
            }
        }

        /**
        * @function  findRecursive( )
        * @abstract  first the first occurrence of needle and return its node
        *            ptr
        * @param     haystack, root of the search tree
        * @param     needle, copy of the data to find
        * @return    the node that contains the needle
        **/
        node_t *findRecursive( node_t* haystack, const INFO_T &needle ) {
            if( haystack->info( ) == needle )
                return haystack;

            node_t *n =0;
            if( haystack->leftChild( ) )
                n =findRecursive( haystack->leftChild( ), needle );
            if( !n && haystack->rightChild( ) )
                n =findRecursive( haystack->rightChild( ), needle );
            return n;
        }

        friend class TreeNodeIterator_pre<INFO_T>;
```

```
        friend class TreeNodeIterator_in<INFO_T>;
        friend class SplayTree<INFO_T>;
        TreeNode<INFO_T> *m_root;

    private:
        /**
        * @function  getFirstEmptySlot( )
        * @abstract  when a row has a continuous empty space on the right,
        *            find the left-most parent in the above row that has
        *            at least one empty slot.
        * @param     level, how many level still to go
        * @return    the first empty slot where we can put a new node
        * @pre       level should be > 1
        **/
        node_t *getFirstEmptySlot( int level ) {
            node_t *p =0;
            nodelist rlist =row( level-1 ); // we need the parents of this level
            /** changed auto to int **/
            for( auto it =rlist.rbegin( ); it !=rlist.rend( ); ++it ) {
                if( !(*it)->hasChildren( ) )
                    p =(*it);
                else if( !(*it)->rightChild( ) ) {
                    p =(*it);
                    break;
                } else
                    break;
            }
            return p;
        }
};

#endif
```

6.4 TreeNode.h

```
/**
 * Treenode:
 *
 * @author  Micky Faas (s1407937)
 * @author  Lisette de Schipper (s1396250)
 * @file    Treenode.h
 * @date    26-10-2014
 **/

#ifndef TREENODE_H
#define TREENODE_H

using namespace std;

template <class INFO_T> class Tree;
class ExpressionTree;

template <class INFO_T> class TreeNode
{
```

```
    public:
      /**
       * @function  TreeNode( )
       * @abstract  Constructor, creates a node
       * @param     info, the contents of a node
       * @param     parent, the parent of the node
       * @post      A node has been created.
       **/
       TreeNode( const INFO_T& info, TreeNode<INFO_T>* parent =0 )
           : m_lchild( 0 ), m_rchild( 0 ) {
           m_info =info;
           m_parent =parent;
       }

      /**
       * @function  TreeNode( )
       * @abstract  Constructor, creates a node
       * @param     parent, the parent of the node
       * @post      A node has been created.
       **/
       TreeNode( TreeNode<INFO_T>* parent =0 )
           : m_lchild( 0 ), m_rchild( 0 ) {
           m_parent =parent;
       }

      /**
       * @function  =
       * @abstract  Sets a nodes content to N
       * @param     n, the contents you want the node to have
       * @post      The node now has those contents.
       **/
       void operator =( INFO_T n ) { m_info =n; }

      /**
       * @function  INFO_T( ), info( )
       * @abstract  Returns the content of a node
       * @return    m_info, the contents of the node
       **/
       operator INFO_T( ) const { return m_info; }
       const INFO_T &info( ) const { return m_info; }
       INFO_T &info( ) { return m_info; }
       /**
       * @function  atRow( )
       * @abstract  returns the level or row-number of this node
       * @return    row, an int of row the node is at
       **/
       int atRow( ) const {
           const TreeNode<INFO_T> *n =this;
           int row =0;
           while( n->parent( ) ) {
               n =n->parent( );
               row++;
           }
           return row;
```

```
        }

        /**
        * @function  parent( ), leftChild( ), rightChild( )
        * @abstract  returns the adress of the parent, left child and right
        *            child respectively
        * @return    the adress of the requested family member of the node
        **/
        TreeNode<INFO_T> *parent( ) const { return m_parent; }
        TreeNode<INFO_T> *leftChild( ) const { return m_lchild; }
        TreeNode<INFO_T> *rightChild( ) const { return m_rchild; }

        /**
        * @function  swapWith( )
        * @abstract  Swaps this node with another node in the tree
        * @param     n, the node to swap this one with
        * @pre       both this node and n must be in the same parent tree
        * @post      n will have the parent and children of this node
        *            and vice verse. Both nodes retain their data.
        **/
        void swapWith( TreeNode<INFO_T>* n ) {
            bool this_wasLeftChild =false, n_wasLeftChild =false;
            if( parent( ) && parent( )->leftChild( ) == this )
                this_wasLeftChild =true;
            if( n->parent( ) && n->parent( )->leftChild( ) == n )
                n_wasLeftChild =true;

            // Swap the family info
            TreeNode<INFO_T>* newParent =
                ( n->parent( ) == this ) ? n : n->parent( );
            TreeNode<INFO_T>* newLeft =
                ( n->leftChild( ) == this ) ? n :n->leftChild( );
            TreeNode<INFO_T>* newRight =
                 ( n->rightChild( ) == this ) ? n :n->rightChild( );

            n->setParent( parent( ) == n ? this : parent( ) );
            n->setLeftChild( leftChild( ) == n ? this : leftChild( ) );
            n->setRightChild( rightChild( ) == n ? this : rightChild( ) );

            setParent( newParent );
            setLeftChild( newLeft );
            setRightChild( newRight );

            // Restore applicable pointers
            if( n->leftChild( ) )
                n->leftChild( )->setParent( n );
            if( n->rightChild( ) )
                n->rightChild( )->setParent( n );
            if( leftChild( ) )
                leftChild( )->setParent( this );
            if( rightChild( ) )
                rightChild( )->setParent( this );
            if( n->parent( ) ) {
                if( this_wasLeftChild )
```

```
                n->parent( )->setLeftChild( n );
            else
                n->parent( )->setRightChild( n );
        }
        if( parent( ) ) {
            if( n_wasLeftChild )
                parent( )->setLeftChild( this );
            else
                parent( )->setRightChild( this );
        }
    }

    /**
    * @function  replace( )
    * @abstract  Replaces the node with another node in the tree
    * @param     n, the node we replace the node with, this one gets deleted
    * @pre       both this node and n must be in the same parent tree
    * @post      The node will be replaced and n will be deleted.
    **/
    void replace( TreeNode<INFO_T>* n ) {
        bool n_wasLeftChild =false;

        if( n->parent( ) && n->parent( )->leftChild( ) == n )
            n_wasLeftChild =true;

        // Swap the family info
        TreeNode<INFO_T>* newParent =
            ( n->parent( ) == this ) ? n : n->parent( );
        TreeNode<INFO_T>* newLeft =
            ( n->leftChild( ) == this ) ? n :n->leftChild( );
        TreeNode<INFO_T>* newRight =
             ( n->rightChild( ) == this ) ? n :n->rightChild( );

        setParent( newParent );
        setLeftChild( newLeft );
        setRightChild( newRight );
        m_info = n->m_info;

        // Restore applicable pointers
        if( leftChild( ) )
            leftChild( )->setParent( this );
        if( rightChild( ) )
            rightChild( )->setParent( this );

        if( parent( ) ) {
            if( n_wasLeftChild )
                parent( )->setLeftChild( this );
            else
                parent( )->setRightChild( this );
        }
        delete n;
    }

    /**
```

```
     * @function  sibling( )
     * @abstract  returns the address of the sibling
     * @return    the address to the sibling or zero if there is no sibling
     **/
    TreeNode<INFO_T>* sibling( ) {
        if( parent( )->leftChild( ) == this )
            return parent( )->rightChild( );
        else if( parent( )->rightChild( ) == this )
            return parent( )->leftChild( );
        else
            return 0;
    }

   /**
    * @function  hasChildren( ), hasParent( ), isFull( )
    * @abstract  Returns whether the node has children, has parents or is
    *            full (has two children) respectively
    * @param
    * @return    true or false, depending on what is requested from the node.
    *            if hasChildren is called and the node has children, it will
    *            return true, otherwise false.
    *            If hasParent is called and the node has a parent, it will
    *            return true, otherwise false.
    *            If isFull is called and the node has two children, it will
    *            return true, otherwise false.
    **/
    bool hasChildren( ) const { return m_lchild || m_rchild; }
    bool hasParent( ) const { return m_parent; }
    bool isFull( ) const { return m_lchild && m_rchild; }

  protected:
    friend class Tree<INFO_T>;
    friend class ExpressionTree;

   /**
    * @function  setParent( ), setLeftChild( ), setRightChild( )
    * @abstract  sets the parent, left child and right child of the
    *            particular node respectively
    * @param     p, the node we want to set a certain family member of
    * @return    void
    * @post      The node now has a parent, a left child or a right child
    *            respectively.
    **/
    void setParent( TreeNode<INFO_T> *p ) { m_parent =p; }
    void setLeftChild( TreeNode<INFO_T> *p ) { m_lchild =p; }
    void setRightChild( TreeNode<INFO_T> *p ) { m_rchild =p; }

  private:
    INFO_T m_info;
    TreeNode<INFO_T> *m_parent;
    TreeNode<INFO_T> *m_lchild;
    TreeNode<INFO_T> *m_rchild;
};
```

```
/**
* @function  <<
* @abstract  the contents of the node are returned
* @param     out, in what format we want to get the contents
* @param     rhs, the node of which we want the contents
* @return    the contents of the node.
**/
template <class INFO_T> ostream &operator <<(ostream& out, const TreeNode<INFO_T> & r
    out << rhs.info( );
    return out;
}

#endif
```

6.5 TreeNodeIterator.h

```
/**
 * TreeNodeIterator: Provides a set of iterators that follow the STL-standard
 *
 * @author  Micky Faas (s1407937)
 * @author  Lisette de Schipper (s1396250)
 * @file    TreeNodeIterator.h
 * @date    26-10-2014
 **/

#include <iterator>
#include "TreeNode.h"

template <class INFO_T> class TreeNodeIterator
                        : public std::iterator<std::forward_iterator_tag,
                                               TreeNode<INFO_T>> {
    public:
        typedef TreeNode<INFO_T> node_t;

       /**
        * @function  TreeNodeIterator( )
        * @abstract  (copy)constructor
        * @pre       TreeNodeIterator is abstract and cannot be constructed
        **/
        TreeNodeIterator( node_t* ptr =0 ) : p( ptr ) { }
        TreeNodeIterator( const TreeNodeIterator& it ) : p( it.p ) { }

       /**
        * @function  (in)equality operator overload
        * @abstract  Test (in)equality for two TreeNodeIterators
        * @param     rhs, right-hand side of the comparison
        * @return    true if both iterators point to the same node (==)
        *            false if both iterators point to the same node (!=)
        **/
        bool operator == (const TreeNodeIterator& rhs) { return p==rhs.p; }
        bool operator != (const TreeNodeIterator& rhs) { return p!=rhs.p; }

       /**
        * @function  operator*( )
```

```
         * @abstract  Cast operator to node_t reference
         * @return    The value of the current node
         * @pre       Must point to a valid node
         **/
         node_t& operator*( ) { return *p; }

        /**
         * @function  operator++( )
         * @abstract  pre- and post increment operators
         * @return    TreeNodeIterator that has iterated one step
         **/
         TreeNodeIterator &operator++( ) { next( ); return *this; }
         TreeNodeIterator operator++( int )
             { TreeNodeIterator tmp( *this ); operator++( ); return tmp; }
     protected:

        /**
         * @function  next( ) //(pure virtual)
         * @abstract  Implement this function to implement your own iterator
         */
         virtual bool next( ){ return false; }// =0;
         node_t *p;
};

template <class INFO_T> class TreeNodeIterator_pre
                        : public TreeNodeIterator<INFO_T> {
     public:
         typedef TreeNode<INFO_T> node_t;

         TreeNodeIterator_pre( node_t* ptr =0 )
             : TreeNodeIterator<INFO_T>( ptr ) { }
         TreeNodeIterator_pre( const TreeNodeIterator<INFO_T>& it )
             : TreeNodeIterator<INFO_T>( it ) { }
         TreeNodeIterator_pre( const TreeNodeIterator_pre& it )
             : TreeNodeIterator<INFO_T>( it.p ) { }

         TreeNodeIterator_pre &operator++( ) { next( ); return *this; }
         TreeNodeIterator_pre operator++( int )
             { TreeNodeIterator_pre tmp( *this ); operator++( ); return tmp; }

     protected:
         using TreeNodeIterator<INFO_T >::p;

        /**
         * @function  next( )
         * @abstract  Takes one step in pre-order traversal
         * @return    returns true if such a step exists
         */
         bool next( ) {
             if( !p )
                 return false;
             if( p->hasChildren( ) ) { // a possible child that can be the next
                 p =p->leftChild( ) ? p->leftChild( ) : p->rightChild( );
                 return true;
```

```
            }
            else if( p->hasParent( ) // we have a right brother
                    && p->parent( )->rightChild( )
                    && p->parent( )->rightChild( ) != p ) {
                p =p->parent( )->rightChild( );
                return true;
            }
            else if( p->hasParent( ) ) { // just a parent, thus we go up
                TreeNode<INFO_T> *tmp =p->parent( );
                while( tmp->parent( ) ) {
                    if( tmp->parent( )->rightChild( )
                            && tmp->parent( )->rightChild( ) != tmp ) {
                        p =tmp->parent( )->rightChild( );
                        return true;
                    }
                    tmp =tmp->parent( );
                }
            }
            // Nothing left
            p =0;
            return false;
        }

};

template <class INFO_T> class TreeNodeIterator_in
                        : public TreeNodeIterator<INFO_T>{
    public:
        typedef TreeNode<INFO_T> node_t;

        TreeNodeIterator_in( node_t* ptr =0 )
            : TreeNodeIterator<INFO_T>( ptr ) { }
        TreeNodeIterator_in( const TreeNodeIterator<INFO_T>& it )
            : TreeNodeIterator<INFO_T>( it ) { }
        TreeNodeIterator_in( const TreeNodeIterator_in& it )
            : TreeNodeIterator<INFO_T>( it.p ) { }

        TreeNodeIterator_in &operator++( ) { next( ); return *this; }
        TreeNodeIterator_in operator++( int )
            { TreeNodeIterator_in tmp( *this ); operator++( ); return tmp; }

    protected:
        using TreeNodeIterator<INFO_T >::p;
       /**
        * @function  next( )
        * @abstract  Takes one step in in-order traversal
        * @return    returns true if such a step exists
        */
        bool next( ) {
            if( p->rightChild( ) ) {
                p =p->rightChild( );
                while( p->leftChild( ) )
                    p =p->leftChild( );
                return true;
```

```
            }
            else if( p->parent( ) && p->parent( )->leftChild( ) == p ) {
                p =p->parent( );
                return true;
            } else if( p->parent( ) && p->parent( )->rightChild( ) == p ) {
                p =p->parent( );
                while( p->parent( ) && p == p->parent( )->rightChild( ) ) {
                    p =p->parent( );
                }
                if( p )
                    p =p->parent( );
                if( p )
                    return true;
                else
                    return false;
            }
            // Er is niks meer
            p =0;
            return false;
        }
};

template <class INFO_T> class TreeNodeIterator_post
                        : public TreeNodeIterator<INFO_T>{
    public:
        typedef TreeNode<INFO_T> node_t;

        TreeNodeIterator_post( node_t* ptr =0 )
            : TreeNodeIterator<INFO_T>( ptr ) { }
        TreeNodeIterator_post( const TreeNodeIterator<INFO_T>& it )
            : TreeNodeIterator<INFO_T>( it ) { }
        TreeNodeIterator_post( const TreeNodeIterator_post& it )
            : TreeNodeIterator<INFO_T>( it.p ) { }

        TreeNodeIterator_post &operator++( ) { next( ); return *this; }
        TreeNodeIterator_post operator++( int )
            { TreeNodeIterator_post tmp( *this ); operator++( ); return tmp; }

    protected:
        using TreeNodeIterator<INFO_T >::p;
        /**
         * @function  next( )
         * @abstract  Takes one step in post-order traversal
         * @return    returns true if such a step exists
         */
        bool next( ) {

            if( p->hasParent( ) // We have a right brother
                    && p->parent( )->rightChild( )
                    && p->parent( )->rightChild( ) != p ) {
                p =p->parent( )->rightChild( );
                while( p->leftChild( ) )
                    p =p->leftChild( );
                return true;
```

```
            } else if( p->parent( ) ) {
                p =p->parent( );
                return true;
            }
            // Nothing left
            p =0;
            return false;
        }
};
```

6.6 SelfOrganizingTree.h

```
/**
 * SelfOrganizingTree - Abstract base type inheriting from Tree
 *
 * @author  Micky Faas (s1407937)
 * @author  Lisette de Schipper (s1396250)
 * @file    SelfOrganizingTree.h
 * @date    3-11-2014
 **/

#ifndef SELFORGANIZINGTREE_H
#define SELFORGANIZINGTREE_H

#include "BinarySearchTree.h"

template <class INFO_T> class SelfOrganizingTree : public BinarySearchTree<INFO_T> {
    public:
        typedef BSTNode<INFO_T> node_t;
        typedef BinarySearchTree<INFO_T> S; // super class

      /**
        * @function  SelfOrganizingTree( ) : S( )
        * @abstract  Constructor
        **/
        SelfOrganizingTree( ) : S( ) { }

       /**
        * @function  rotateLeft( ) and rotateRight( )
        * @abstract  Performs a rotation with the given node as root of the
        *            rotating subtree, either left of right.
        *            The tree's root pointer will be updated if neccesary.
        * @param     node, the node to rotate
        * @pre       The node must be a node in this tree
        * @post      The node may be be the new root of the tree
        *            No nodes will be invalided and no new memory is
        *            allocated. Iterators may become invalid.
        **/
        virtual node_t *rotateLeft( node_t * node ){
            if( this->root( ) == node )
                return static_cast<node_t *>( S::m_root = node->rotateLeft( ) );
            else
                return node->rotateLeft( );
        }
```

```
        virtual node_t *rotateRight( node_t * node ){
            if( this->root( ) == node )
                return static_cast<node_t *>( S::m_root = node->rotateRight( ) );
            else
                return node->rotateRight( );
        }

    private:

};


#endif
```

6.7 BinarySearchTree.h

```
/**
 * BinarySearchTree - BST that inherits from Tree
 *
 * @author  Micky Faas (s1407937)
 * @author  Lisette de Schipper (s1396250)
 * @file    BinarySearchTree.h
 * @date    3-11-2014
 **/

#ifndef BINARYSEARCHTREE_H
#define BINARYSEARCHTREE_H

#include "Tree.h"
#include "BSTNode.h"

template <class INFO_T> class BinarySearchTree : public Tree<INFO_T> {
    public:
        typedef BSTNode<INFO_T> node_t;
        typedef Tree<INFO_T> S; // super class

        BinarySearchTree( ) : S( ) { }
        BinarySearchTree( const BinarySearchTree& cpy ) : S( cpy ) { }

        virtual ~BinarySearchTree( ) { }

       /**
        * @function  pushBack( )
        * @abstract  reimplemented virtual function from Tree<>
        *            this function is semantically identical to insert()
        * @param     info, the contents of the new node
        **/
        virtual node_t *pushBack( const INFO_T& info ) {
            return insert( info );
        }

       /**
        * @function  insert( )
```

```
     * @abstract  reimplemented virtual function from Tree<>
     *            the exact location of the new node is determined
     *            by the rules of the binary search tree.
     * @param     info, the contents of the new node
     * @param     parent, ignored
     * @param     preferRight, ignored
     * @param     replaceBehavior, ignored
     * @return    returns a pointer to the inserted node
     **/
    virtual node_t* insert( const INFO_T& info,
                      TreeNode<INFO_T>* parent =0, // Ignored
                      bool preferRight =false,     // Ignored
                      int replaceBehavior =S::ABORT_ON_EXISTING ) { // Ignored
        node_t *n =new node_t( );
        return insertInto( info, n );
    }

    /**
     * @function  replace( )
     * @abstract  reimplemented virtual function from Tree<>
     *            replaces a given node or the root
     *            the location of the replaced node may be different
     *            due to the consistency of the binary search tree
     * @param     info, the contents of the new node
     * @param     node, node to be replaced
     * @param     alignRight, ignored
     * @param     replaceBehavior, ignored
     * @return    returns a pointer to the new node
     * @pre       node should be in this tree
     * @post      replace() will delete and/or remove node.
     *            if node is 0, it will take the root instead
     **/
    virtual node_t* replace( const INFO_T& info,
                       TreeNode<INFO_T>* node =0,
                       bool alignRight =false,
                       int replaceBehavior =S::DELETE_EXISTING ) {
        node_t *newnode;
        if( !node )
            node =S::m_root;
        if( !node )
            return pushBack( info );

        bool swap =false;
        // We can either just swap the new node with the old and remove
        // the old, or we can remove the old and add the new node via
        // pushBack(). This depends on the value of info
        if( !node->hasChildren( ) ) {
            swap =true;
        }
        else if( !(node->leftChild( )
                && node->leftChild( )->info( ) > info )
                && !(node->rightChild( )
                && node->rightChild( )->info( ) < info ) ) {
            swap =true;
```

```
            }
            if( swap ) {
                newnode = new node_t( info );
                if( node == S::m_root )
                    S::m_root = newnode;
                node->swapWith( newnode );
                delete node;
            } else {
                remove( node );
                newnode = pushBack( info );

            }
            return newnode;
        }

       /**
        * @function  remove( )
        * @abstract  reimplemented virtual function from Tree<>
        *            removes a given node or the root and restores the
        *            BST properties
        * @param     node, node to be removed
        * @pre       node should be in this tree
        * @post      memory for node will be deallocated
        **/
        virtual void remove( TreeNode<INFO_T> *node ) {
            node_t *n = static_cast<node_t*>( node );

            while( n->isFull( ) ) {
                // the difficult case
                // we could take either left of right here
                TreeNode<INFO_T> *temp;
                temp = n->leftChild( );
                while( temp->rightChild( ) ) {
                    temp = temp->rightChild( );
                }
                if( n == S::m_root )
                    S::m_root = temp;
                n->swapWith( temp );
            }


            // Assume the above is fixed
            while( n->hasChildren( ) ) {
                if( n->leftChild( ) ) {
                    if( n == S::m_root )
                        S::m_root = n->leftChild( );
                    n->swapWith( n->leftChild( ) );
                }
                else {
                    if( n == S::m_root )
                        S::m_root = n->rightChild( );
                    n->swapWith( n->rightChild( ) );
                }
            }
```

```
            if( n->parent( ) && n->parent( )->leftChild( ) == n )
                static_cast<node_t*>( n->parent( ) )->setLeftChild( 0 );
            else if( n->parent( ) && n->parent( )->rightChild( ) == n )
                static_cast<node_t*>( n->parent( ) )->setRightChild( 0 );
            delete n;
        }

       /**
        * @function  find( )
        * @abstract  reimplemented virtual function from Tree<>
        *            performs a binary search in a given (sub)tree
        * @param     haystack, the subtree to search. Give 0 for the entire tree
        * @param     needle, key/info-value to find
        * @return    returns a pointer to node, if found
        * @pre       haystack should be in this tree
        * @post      may return 0
        **/
        virtual TreeNode<INFO_T>* find( TreeNode<INFO_T>* haystack,
                                        const INFO_T& needle ) {
            m_searchStepCounter =0;

            if( !haystack )
                haystack =S::m_root;
            while( haystack && haystack->info( ) != needle ) {
                m_searchStepCounter++;
                if( haystack->info( ) > needle )
                    haystack =haystack->leftChild( );
                else
                    haystack =haystack->rightChild( );
            }
            if( !haystack )
                m_searchStepCounter =-1;
            return haystack;
        }

       /**
        * @function  lastSearchStepCount( )
        * @abstract  gives the amount of steps needed to complete the most
        *            recent call to find( )
        * @return    positive amount of steps on a defined search result,
        *            -1 on no search result
        */
        virtual int lastSearchStepCount( ) const {
            return m_searchStepCounter;
        }

       /**
        * @function     min( )
        * @abstract     Returns the node with the least value in a binary search
        *               tree. This is achieved through recursion.
        * @param        node - the node from which we start looking
        * @return       Eventually, at the end of the recursion, we return the
        *               adress of the node with the smallest value.
```

```
     * @post          The node with the smallest value is returned.
     **/
     node_t* min( node_t* node ) const {
         return node->leftChild( ) ?
                 min(static_cast<node_t*>( node->leftChild( ) ) ) : node;
     }

    /**
     * @function     min( )
     * @abstract     We call the function mentioned above and then
     *               return the node with the least value in a binary search
     *               tree.
     * @return       We return the adress of the node with the smallest value.
     * @post         The node with the smallest value is returned.
     **/
     node_t* min( ) const {
         return min( static_cast<node_t*>( this->root( ) ) );
     }

    /**
     * @function     max( )
     * @abstract     Returns the node with the highest value in a binary
     *               search tree. This is achieved through recursion.
     * @param        node - the node from which we start looking
     * @return       Eventually, at the end of the recursion, we return the
     *               adress of the node with the highest value.
     * @post         The node with the highest value is returned.
     **/
     node_t* max( node_t* node ) const  {
         return node->rightChild( ) ?
                 max(static_cast<node_t*>( node->rightChild( ) ) ) : node;
     }

    /**
     * @function     max( )
     * @abstract     We call the function mentioned above and then
     *               return the node with the highest value in a binary
     *               search tree.
     * @return       We return the adress of the node with the highest value.
     * @post         The node with the highest value is returned.
     **/
     node_t* max( ) const {
         return max( static_cast<node_t*>( this->root( ) ) );
     }

 protected:
     /**
     * @function  insertInto( )
     * @abstract  Inserts new node into the tree following BST rules
     *            Assumes that the memory for the node is already allocated
     *            This function exists mainly because of derived classes
     *            want to insert nodes of a derived type.
     * @param     info, the contents of the new node
     * @param     n, node pointer, should be already allocated
```

```
        * @return    returns a pointer to the inserted node
        **/
        virtual node_t* insertInto( const INFO_T& info,
                            node_t* n ) { // Preallocated
            n->info() =info;

            if( !S::m_root )
                S::m_root =n;
            else {
                node_t *parent =0;
                node_t *sub =static_cast<node_t*>( S::m_root );
                do {
                    if( *n < *sub ) {
                        parent =sub;
                        sub =static_cast<node_t*>( parent->leftChild( ) );
                    }
                    else {
                        parent =sub;
                        sub =static_cast<node_t*>( parent->rightChild( ) );
                    }
                } while( sub );
                if( *n < *parent )
                    parent->setLeftChild( n );
                else
                    parent->setRightChild( n );
                n->setParent( parent );
            }
            return n;
        }

        int m_searchStepCounter;
};

#endif
```

6.8 BSTNode.h

```
/**
 * BSTNode - Node atom for BinarySearchTree
 *
 * @author  Micky Faas (s1407937)
 * @author  Lisette de Schipper (s1396250)
 * @file    BSTNode.h
 * @date    3-11-2014
 **/

#ifndef BSTNODE_H
#define BSTNODE_H

#include "TreeNode.h"

template <class INFO_T> class BinarySearchTree;

template <class INFO_T> class BSTNode : public TreeNode<INFO_T>
```

```
{
    public:
        typedef TreeNode<INFO_T> S; // super class

        /**
        * @function  BSTNode( )
        * @abstract  Constructor, creates a node
        * @param     info, the contents of a node
        * @param     parent, the parent of the node
        * @post      A node has been created.
        **/
        BSTNode( const INFO_T& info, BSTNode<INFO_T>* parent =0 )
            : S( info, parent ) { }

       /**
        * @function  BSTNode( )
        * @abstract  Constructor, creates a node
        * @param     parent, the parent of the node
        * @post      A node has been created.
        **/
        BSTNode( BSTNode<INFO_T>* parent =0 )
            : S( (S)parent ) { }

        // Idea: rotate this node left and return the node that comes in its place
        BSTNode *rotateLeft( ) {

            if( !this->rightChild( ) ) // Cannot rotate
                return this;

            bool isLeftChild =this->parent( ) && this == this->parent( )->leftChild(

            // new root of tree
            BSTNode *newTop =static_cast<BSTNode *>(this->rightChild( ));
            // new rightchild of the node that is rotated
            BSTNode *newRight =static_cast<BSTNode *>(newTop->leftChild( ));
            // the parent under which all of the magic is happening
            BSTNode *topParent =static_cast<BSTNode *>(this->parent( ));

            // We become left-child of our right-child
            // newTop takes our place with our parent
            newTop->setParent( topParent );
            if( isLeftChild && topParent )
                topParent->setLeftChild( newTop );
            else if( topParent )
                topParent->setRightChild( newTop );

            newTop->setLeftChild( this );
            this->setParent( newTop );

            // We take the left-child of newTop as our right-child
            this->setRightChild( newRight );
            if( newRight )
                newRight->setParent( this );

```

```
            return newTop;
        }

        // Idea: rotate this node right and return the node that comes in its place
        BSTNode *rotateRight( ) {
            if( !this->leftChild( ) ) // Cannot rotate
                return this;

            bool isRightChild =this->parent( ) && this == this->parent( )->rightChild

            // new root of tree
            BSTNode *newTop =static_cast<BSTNode *>(this->leftChild( ));
            // new leftchild of the node that is rotated
            BSTNode *newLeft =static_cast<BSTNode *>(newTop->rightChild( ));
            // the parent under which all of the magic is happening
            BSTNode *topParent =static_cast<BSTNode *>(this->parent( ));

            // We become left-child of our right-child
            // newTop takes our place with our parent
            newTop->setParent( topParent );
            if( isRightChild && topParent )
                topParent->setRightChild( newTop );
            else if( topParent )
                topParent->setLeftChild( newTop );

            newTop->setRightChild( this );
            this->setParent( newTop );

            // We take the left-child of newTop as our right-child
            this->setLeftChild( newLeft );
            if( newLeft )
                newLeft->setParent( this );

            return newTop;
        }

        bool operator <( const BSTNode<INFO_T> &rhs ) {
            return S::info() < rhs.info();
        }

        bool operator <=( const BSTNode<INFO_T> &rhs ) {
            return S::info() <= rhs.info();
        }

        bool operator >( const BSTNode<INFO_T> &rhs ) {
            return S::info() > rhs.info();
        }

        bool operator >=( const BSTNode<INFO_T> &rhs ) {
            return S::info() >= rhs.info();
        }
    protected:
        friend class BinarySearchTree<INFO_T>;
};
```

```

#endif
```

6.9 AVLTree.h

```
/**
 * AVLTree - AVL-SelfOrganizingTree that inherits from SelfOrganizingTree
 *
 * @author  Micky Faas (s1407937)
 * @author  Lisette de Schipper (s1396250)
 * @file    AVLTree.h
 * @date    9-12-2014
 **/

#ifndef AVLTREE_H
#define AVLTREE_H

#include "SelfOrganizingTree.h"
#include "AVLNode.h"

template <class INFO_T> class AVLTree : public SelfOrganizingTree<INFO_T> {
    public:
        typedef AVLNode<INFO_T> node_t;
        typedef SelfOrganizingTree<INFO_T> S; // super class

       /**
        * @function     AVLTree( )
        * @abstract     constructor
        * @post         An AVLTree is created
        **/
        AVLTree( ) : S( ) { }

       /**
        * @function     AVLTree( )
        * @abstract     constructor
        * @param        cpy
        * @post         An AVLTree is created
        **/
        AVLTree( const AVLTree& cpy ) : S( cpy ) { }

       /**
        * @function     insert( )
        * @abstract     A node with label 'info' is inserted into the tree and
        *               put in the right place. A label may not appear twice in
        *               a tree.
        * @param        info - the label of the node
        * @return       the node we inserted
        * @post         The tree now contains a node with 'info'
        **/
        node_t* insert( const INFO_T& info,
                        TreeNode<INFO_T>* parent =0, // Ignored
                        bool preferRight =false,     // Ignored
                        int replaceBehavior =0 ) { // Ignored
            if( S::find( this->root( ), info ) )
```

```
                return 0;
            node_t *node = new node_t( );
            S::insertInto( info, node );
            rebalance( node );
            return node;
        }

        /**
         * @function     remove( )
         * @abstract     A node is removed in such a way that the properties of
         *               an AVL tree remain intact.
         * @param        node - the node we're going to remove
         * @post         The node has breen removed, but the remaining tree still
         *               contains all of its other nodes and still has all the
         *               AVL properties.
         **/
        void remove( node_t* node ) {
            // if it's a leaf
            if( !node->leftChild( ) && !node->rightChild( ) )
                S::remove( node );
            // internal node with kids
            else {
                if( node->rightChild( ) ) {
                    node = static_cast<node_t*>( S::replace(
                           S::min(  static_cast<node_t*>(
                           node->rightChild( ) ) )->info( ), node ) );
                    removeMin( static_cast<node_t*>( node->rightChild( ) ) );
                    node->setRightChild( node->rightChild( ));
                }
                else
                    // just delete the node and replace it with its leftChild
                    node->replace( node->leftChild( ) );
            }
        }

    private:

        /**
         * @function     removeMin( )
         * @abstract     Recursively we go through the tree to find the node with
         *               the smallest value in the subtree with root node. Then we
         *               restore the balance factors of all its parents.
         * @param        node - the root of the subtree we're looking in
         * @return       At the end of the recursion we return the parent of the
         *               node with the smallest value. Then we go up the tree and
         *               rebalance every parent from this upwards.
         * @post         The node with the smallest value is deleted and every
         *               node still has the correct balance factor.
         **/
        node_t* removeMin( node_t* node ) {
            node_t* temp;
            if( node->leftChild( ) )
                temp = removeMin( static_cast<node_t*>( node->leftChild( ) ) );
            else {
```

```
                temp =static_cast<node_t*>( node->parent( ) );
                S::remove( node );
            }
            rebalance( temp );
            return temp;
        }

       /**
        * @function     removeMax( )
        * @abstract     Recursively we go through the tree to find the node with
        *               the highest value in the subtree with root node. Then we
        *               restore the balance factors of all its parents.
        * @param        node - the root of the subtree we're looking in
        * @return       At the end of the recursion we return the parent of the
        *               node with the highest value. Then we go up the tree and
        *               rebalance every parent from this upwards.
        * @post         The node with the highest value is deleted and every
        *               node still has the correct balance factor.
        **/
        node_t* removeMax( node_t* node ) {
            node_t* temp;
            if( node->rightChild( ) )
                temp =removeMin( static_cast<node_t*>( node->rightChild( ) ) );
            else {
                temp =static_cast<node_t*>( node->parent( ) );
                S::remove( node );
            }
            rebalance( temp );
            return temp;
        }

       /**
        * @function     rotateLeft( )
        * @abstract     We rotate a node left and make sure all the internal
        *               heights of the nodes are up to date.
        * @param        node - the node we're going to rotate left
        * @return       we return the node that is now at the top of this
        *               particular subtree.
        * @post         The node is rotated to the left and the heights are up
        *               to date.
        **/
        node_t* rotateLeft( node_t* node ) {
            node_t *temp =static_cast<node_t*>( S::rotateLeft( node ) );
            temp->updateHeight( );
            if( temp->leftChild( ) )
                static_cast<node_t *>( temp->leftChild( ) )->updateHeight( );
            return temp;
        }

       /**
        * @function     rotateRight( )
        * @abstract     We rotate a node right and make sure all the internal
        *               heights of the nodes are up to date.
        * @param        node - the node we're going to rotate right
```

```
         * @return       we return the node that is now at the top of this
         *               particular subtree.
         * @post         The node is rotated to the right and the heights are up
         *               to date.
         **/
        node_t* rotateRight( node_t* node ) {
            node_t* temp =static_cast<node_t*>( S::rotateRight( node ) );
            temp->updateHeight( );
            if( temp->rightChild( ) )
                static_cast<node_t*>( temp->rightChild( ) )->updateHeight( );
            return temp;
        }

        /**
         * @function     rebalance( )
         * @abstract     The tree is rebalanced. We do the necessary rotations
         *               from the bottom up to make sure the AVL properties are
         *               still intact.
         * @param        node - the node we're going to rebalance
         * @post         The tree is now perfectly balanced.
         **/
        void rebalance( node_t* node ) {
            node->updateHeight( );

            node_t* temp =node;
            while( temp->parent( ) ) {
                temp =static_cast<node_t*>( temp->parent( ) );
                temp->updateHeight( );
                // right subtree too deep
                if( temp->balanceFactor( ) == 2 ) {
                    if( temp->rightChild( ) ) {
                        if( static_cast<node_t*>( temp->rightChild( ) )
                            ->balanceFactor( ) < 0 )
                            this->rotateRight(
                            static_cast<node_t*>( temp->rightChild( ) ) );
                    }
                    this->rotateLeft( temp );
                }
                // left subtree too deep
                else if( temp->balanceFactor( ) == -2 ) {
                    if( temp->leftChild( ) ) {
                        if( static_cast<node_t*>( temp->leftChild( ) )->
                            balanceFactor( ) > 0 )
                            this->rotateLeft(
                            static_cast<node_t*>( temp->leftChild( ) ) );
                    }
                    this->rotateRight( temp );
                }
            }
        }
};

#endif
```

6.10 AVLNode.h

```
/**
 * AVLNode - Node atom type for AVLTree
 *
 * @author  Micky Faas (s1407937)
 * @author  Lisette de Schipper (s1396250)
 * @file    AVLNode.h
 * @date    9-11-2014
 **/

#ifndef AVLNODE_H
#define AVLNODE_H

#include "BSTNode.h"

template <class INFO_T> class AVLTree;

template <class INFO_T> class AVLNode : public BSTNode<INFO_T>
{
    public:
        typedef BSTNode<INFO_T> S; // super class

        /**
        * @function     AVLNode( )
        * @abstract     Constructor, creates a node
        * @param        info, the contents of a node
        * @param        parent, the parent of the node
        * @post         A node has been created.
        **/
        AVLNode( const INFO_T& info, AVLNode<INFO_T>* parent =0 )
            : S( info, parent ) {
        }

       /**
        * @function     AVLNode( )
        * @abstract     Constructor, creates a node
        * @param        parent, the parent of the node
        * @post         A node has been created.
        **/
        AVLNode( AVLNode<INFO_T>* parent =0 )
            : S( (S)parent ) {
        }

       /**
        * @function     balanceFactor( )
        * @abstract     we return the height of the rightchild subtracted with
        *               the height of the left child. Because of the properties
        *               of an AVLtree, this should never be less than -1 or more
        *               than 1.
        * @return       we return the difference between the height of the
        *               rightchild and the leftchild.
        * @post         The difference between the two child nodes is returned.
        **/
```

```
        int balanceFactor( ){
            return static_cast<AVLNode *>( this->rightChild( ) )->getHeight( ) -
                   static_cast<AVLNode *>( this->leftChild( ) )->getHeight( );
        }

       /**
        * @function     updateHeight( )
        * @abstract     we update the height of the node.
        * @pre          The children of the node need to have the correct height.
        * @post         The node now has the right height.
        **/
        void updateHeight( ) {
            int lHeight =static_cast<AVLNode *>( this->leftChild( ) )
                         ->getHeight( );
            int rHeight =static_cast<AVLNode *>( this->rightChild( ) )
                         ->getHeight( );

            this->height =( 1 + ( ( lHeight > rHeight ) ? lHeight : rHeight ) );
        }

       /**
        * @function     getHeight( )
        * @abstract     we want to know the height of the node.
        * @return       we return the height of the node.
        * @post         The current height of the node is returned.
        **/
        int getHeight( ) {
            return (this ? this->height : 0);
        }

        bool operator <( const AVLNode<INFO_T> &rhs ) {
            return S::info() < rhs.info();
        }

        bool operator <=( const AVLNode<INFO_T> &rhs ) {
            return S::info() <= rhs.info();
        }

        bool operator >( const AVLNode<INFO_T> &rhs ) {
            return S::info() > rhs.info();
        }

        bool operator >=( const AVLNode<INFO_T> &rhs ) {
            return S::info() >= rhs.info();
        }

    protected:
        friend class AVLTree<INFO_T>;

    private:
        int height;
};
```

```
#endif
```

6.11 SplayTree.h

```
/**
 * SplayTree - Splay-tree implementation
 *
 * @author  Micky Faas (s1407937)
 * @author  Lisette de Schipper (s1396250)
 * @file    SplayTree.h
 * @date    3-11-2014
 **/

#ifndef SPLAYTREE_H
#define SPLAYTREE_H

#include "SelfOrganizingTree.h"

template <class INFO_T> class SplayTree : public SelfOrganizingTree<INFO_T> {
    public:
        typedef BSTNode<INFO_T> node_t;
        typedef SelfOrganizingTree<INFO_T> S; // super class

        SplayTree( ) : SelfOrganizingTree<INFO_T>( ) { }

        SplayTree( const SplayTree& copy )
            : SelfOrganizingTree<INFO_T>( copy ) { }

        /**
        * @function  insert( )
        * @abstract  reimplemented virtual function from BinarySearchTree<>
        *            the new node will always be the root
        * @param     info, the contents of the new node
        * @param     parent, ignored
        * @param     preferRight, ignored
        * @param     replaceBehavior, ignored
        * @return    returns a pointer to the inserted node (root)
        **/
        virtual node_t* insert( const INFO_T& info,
                                TreeNode<INFO_T>* parent =0, // Ignored
                                bool preferRight =false,     // Ignored
                                int replaceBehavior =0 ) { // Ignored
            return splay( S::insert( info, parent, preferRight ) );
        }

        /**
        * @function  replace( )
        * @abstract  reimplemented virtual function from BinarySearchTree<>
        *            replaces a given node or the root
        *            the resulting node will be propagated to location of the root
        * @param     info, the contents of the new node
        * @param     node, node to be replaced
        * @param     alignRight, ignored
        * @param     replaceBehavior, ignored
```

```
         * @return    returns a pointer to the new node (=root)
         * @pre       node should be in this tree
         * @post      replace() will delete and/or remove node.
         *            if node is 0, it will take the root instead
         **/
        virtual node_t* replace( const INFO_T& info,
                                TreeNode<INFO_T>* node =0,
                                bool alignRight =false,
                                int replaceBehavior =0 ) {
            return splay( S::replace( info, node, alignRight ) );
        }

        /**
         * @function  remove( )
         * @abstract  reimplemented virtual function from BinarySearchTree<>
         *            removes a given node or the root and restores the
         *            BST properties. The node-to-be-removed will be spayed
         *            before removal.
         * @param     node, node to be removed
         * @pre       node should be in this tree
         * @post      memory for node will be deallocated
         **/
        virtual void remove( TreeNode<INFO_T> *node ) {
            S::remove( splay( static_cast<node_t*>(node) ) );
        }

        /**
         * @function  find( )
         * @abstract  reimplemented virtual function from Tree<>
         *            performs a binary search in a given (sub)tree
         *            splays the node (if found) afterwards
         * @param     haystack, the subtree to search. Give 0 for the entire tree
         * @param     needle, key/info-value to find
         * @return    returns a pointer to node, if found
         * @pre       haystack should be in this tree
         * @post      may return 0, the structure of the tree may change
         **/
        virtual TreeNode<INFO_T>* find( TreeNode<INFO_T>* haystack,
                                        const INFO_T& needle ) {
            return splay( static_cast<node_t*>( S::find( haystack, needle ) ) );
        }

        /**
         * @function  splay( )
         * @abstract  Performs the splay operation on a given node.
         *            'Splay' means a certain amount of rotations in order
         *            to make the given node be the root of the tree while
         *            maintaining the binary search tree properties.
         * @param     node, the node to splay
         * @pre       The node must be a node in this tree
         * @post      The node will be the new root of the tree
         *            No nodes will be invalided and no new memory is
         *            allocated. Iterators may become invalid.
         **/
```

```
    node_t* splay( node_t* node ) {

        enum MODE {
            LEFT =0x1, RIGHT =0x2,
            PLEFT =0x4, PRIGHT =0x8 };

        // Can't splay the root (or null)
        if( !node || S::m_root == node )
            return node;

        node_t *p =static_cast<node_t*>( node->parent( ) );
        int mode;

        while( p != S::m_root ) {
            if( p->leftChild( ) == node )
                mode =RIGHT;
            else
                mode =LEFT;

            assert( p->parent( ) != nullptr );

            // Node's grandparent
            node_t* g =static_cast<node_t*>( p->parent( ) );

            if( g->leftChild( ) == p )
                mode |= PRIGHT;
            else
                mode |= PLEFT;

            // True if either mode is LEFT|PLEFT or RIGHT|PRIGHT
            if( (mode >> 2) == (mode & 0x3 ) ) {
                // the 'zig-zig' step
                // first rotate g-p then p-node

                if( mode & PLEFT )
                    this->rotateLeft( g );
                else
                    this->rotateRight( g );

                if( mode & LEFT )
                    this->rotateLeft( p );
                else
                    this->rotateRight( p );
            }
            else {
                // the 'zig-zag' step
                // first rotate p-node then g-p

                if( mode & LEFT )
                    this->rotateLeft( p );
                else
                    this->rotateRight( p );

                if( mode & PLEFT )
```

```
                        this->rotateLeft( g );
                    else
                        this->rotateRight( g );
                }

                // perhaps we're done already...
                if( node == this->root( ) )
                    return node;
                else
                    p = static_cast<node_t*>( node->parent( ) );
            }

            // The 'zig-step': parent of node is the root

            if( p->leftChild( ) == node )
                this->rotateRight( p );
            else
                this->rotateLeft( p );

            return node;
        }
};

#endif
```

6.12 Treap.h

```
/**
 * Treap - Treap that inherits from SelfOrganizingTree
 *
 * @author  Micky Faas (s1407937)
 * @author  Lisette de Schipper (s1396250)
 * @file    Treap.h
 * @date    9-12-2014
 **/

#ifndef TREAP_H
#define TREAP_H

#include "SelfOrganizingTree.h"
#include "TreapNode.h"

template <class INFO_T> class Treap : public SelfOrganizingTree<INFO_T> {
    public:
        typedef TreapNode<INFO_T> node_t;
        typedef SelfOrganizingTree<INFO_T> S; // super class

       /**
        * @function     Treap( )
        * @abstract     constructor
        * @post         A Treap is created
        **/
        Treap( int randomRange = 100 ) : S( ) {
            random = randomRange;
```

```
            srand( time( NULL ) );
        }

        /**
         * @function     Treap( )
         * @abstract     constructor
         * @param        cpy
         * @post         A Treap is created
         **/
        Treap( const Treap& cpy, int randomRange =100 ) : S( cpy ) {
            random =randomRange;
            srand( time( NULL ) );
        }

        /**
         * @function     insert( )
         * @abstract     A node with label 'info' is inserted into the tree and
         *               put in the right place. A label may not appear twice in
         *               a tree.
         * @param        info - the label of the node
         * @return       the node we inserted
         * @post         The tree now contains a node with 'info'
         **/
        node_t* insert( const INFO_T& info,
                        TreeNode<INFO_T>* parent =0, // Ignored
                        bool preferRight =false,     // Ignored
                        int replaceBehavior =0 ) { // Ignored
            // Prevent duplicates

            if( S::find( this->root( ), info ) )
                return 0;
            node_t *node =new node_t( );
            S::insertInto( info, node );
            node->priority =rand( ) % random + 1;
            rebalance( node );

            return node;
        }

        /**
         * @function     remove( )
         * @abstract     the node provided with the parameter is deleted from the
         *               tree by rotating it down until it becomes a leaf or has
         *               only one child. In the first  case it's just deleted,
         *               in the second it's replaced by its subtree.
         * @param        node - the node to be deleted
         * @post         The node is deleted from the tree which still retains
         *               the Treap properties.
         **/
        void remove( node_t* node ) {
            node_t *temp = node;
            // rotating it down until the condition no longer applies.
            while( temp->leftChild( ) && temp->rightChild( ) )
            {
```

```
                if( static_cast<node_t*>( temp->rightChild( ) )->priority >
                    static_cast<node_t*>( temp->leftChild( ) )->priority )
                    this->rotateLeft( temp );
                else
                    this->rotateRight( temp );
            }
            // if it's a leaf
            if( !temp->leftChild( ) && !temp->rightChild( ) )
                S::remove( temp );
            // if it only has a right child
            else if( !temp->leftChild( ) )
                temp->replace( static_cast<node_t*>( temp->rightChild( ) ) );
            // if it only has a left child
            else if( !node->rightChild( ) )
                temp->replace( static_cast<node_t*>( temp->leftChild( ) ) );
        }

    private:
        int random;

        /**
         * @function     rebalance( )
         * @abstract     The tree is rebalanced. We do the necessary rotations
         *               from the bottom up to make sure the Treap properties are
         *               still intact.
         * @param        info - the label of the node
         * @return       the node we inserted
         * @post         The tree is now perfectly balanced.
         **/
        void rebalance( node_t* node ) {
            if( !node )
                return;
            node_t* temp =node;
            int myPriority =node->priority;
            while( temp->parent( ) &&
                   myPriority >
                   static_cast<node_t*>( temp->parent( ) )->priority ) {
                temp =static_cast<node_t*>( temp->parent( ) );
                if( temp->leftChild( ) == node )
                    this->rotateRight( temp );
                else
                    this->rotateLeft( temp );
            }
        }

};

#endif
```

6.13 TreapNode.h

```
/**
 * TreapNode - Node atom type for Treap
 *
```

```
 * @author  Micky Faas (s1407937)
 * @author  Lisette de Schipper (s1396250)
 * @file    TreapNode.h
 * @date    9-11-2014
 **/

#ifndef TREAPNODE_H
#define TREAPNODE_H

#include "BSTNode.h"

template <class INFO_T> class Treap;

template <class INFO_T> class TreapNode : public BSTNode<INFO_T>
{
    public:
        typedef BSTNode<INFO_T> S; // super class

        /**
        * @function     TreapNode( )
        * @abstract     Constructor, creates a node
        * @param        info, the contents of a node
        * @param        parent, the parent of the node
        * @post         A node has been created.
        **/
        TreapNode( const INFO_T& info, TreapNode<INFO_T>* parent =0 )
            : S( info, parent ), priority( 0 ) {
        }

       /**
        * @function     TreapNode( )
        * @abstract     Constructor, creates a node
        * @param        parent, the parent of the node
        * @post         A node has been created.
        **/
        TreapNode( TreapNode<INFO_T>* parent =0 )
            : S( (S)parent ), priority( 0 ) {
        }

        /**
        * @function  replace( )
        * @abstract  Replaces the node with another node in the tree
        * @param     n, the node we replace the node with, this one gets deleted
        * @pre       both this node and n must be in the same parent tree
        * @post      The node will be replaced and n will be deleted.
        **/
        void replace( TreapNode<INFO_T>* n ) {
            priority = n->priority;
            this->S::replace( n );
        }

        bool operator <( const TreapNode<INFO_T> &rhs ) {
            return S::info() < rhs.info();
        }
```

```
        bool operator <=( const TreapNode<INFO_T> &rhs ) {
            return S::info() <= rhs.info();
        }

        bool operator >( const TreapNode<INFO_T> &rhs ) {
            return S::info() > rhs.info();
        }

        bool operator >=( const TreapNode<INFO_T> &rhs ) {
            return S::info() >= rhs.info();
        }

        int priority;

    protected:
        friend class Treap<INFO_T>;
};


#endif
```