Hele Hogebomen

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

Inleiding

Werkwijze

Experimenten

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.

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).

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?

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.

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.

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.
- Items over Treap

Resultaten

Appendix

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<int> test;
    test.insert(2);
    auto d =test.insert(4);
    test.insert(8);
    test.insert(7);
    test.insert(6);
    test.insert(1);
    test.insert(0);
    cout << "AVL Boompje:" << endl;
    printTree<int>( test, 5 );
    cout << d->info( ) << " verwijderen: " << endl;
    test.remove( d );
    printTree<int>( test, 5 );

    // Test Treap //

    cout << "Treap" << endl;

    Treap<int> testTreap;
    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;
}
```

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;
            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
         << 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;
    }
    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;
    }

    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>();
            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;
}
```

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
```

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
```

TreeNodeIterator.h

```cpp
/**
 * 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;
        }
};
```