

# Hele Hogeboomen

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 als volgt:

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

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

## 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 geheugen-voetafdruk 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 voldoende om verschillen te kunnen zien. We hebben een aantal testcondities gebruikt:

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

## Hypothesen

- De binary search tree zal vermoedelijk het snelst nieuwe data toevoegen. De splay tree heeft veel ingewikkelde rotatie bij een insert, dus deze zal het traagst zijn.
- Bij het gedicht zal de splay boom waarschijnlijk het snelst zijn omdat deze optimaliseert voor herhalingen.
- ...
- De bomen die een aparte node-klasse gebruiken (avl en treap) gebruiken het meeste geheugen.
- Items over Treap

## Resultaten

## Appendix

### main.cc

```
1  /**
2   * main.cc:
3   *
4   * @author Micky Faas (s1407937)
5   * @author Lisette de Schipper (s1396250)
6   * @file main.cc
7   * @date 26-10-2014
8   */
9
10 #include <iostream>
11 #include "BinarySearchTree.h"
12 #include "Tree.h"
13 #include "AVLTree.h"
14 #include "SplayTree.h"
15 #include "Treap.h"
16 #include <string>
17
18 using namespace std;
19
20 // Makkelijk voor debuggen, moet nog beter
21 template<class T> void printTree( Tree<T> tree, int rows ) {
22     typename Tree<T>::odelist list = tree.row( 0 );
23     int row = 0;
24     while( !list.empty( ) && row < rows ) {
25         string offset;
26         for( int i = 0; i < ( 1 << ( rows - row ) ) - 1 ; ++i )
27             offset += ' ';
28
29         for( auto it = list.begin( ); it != list.end( ); ++it ) {
30             if( *it )
```

```

32         cout << offset << (*it)->info() << " " << offset;
33     else
34         cout << offset << ". " << offset;
35     }
36     cout << endl;
37     row++;
38     list = tree.row( row );
39 }
40 }
41
42 int main ( int argc, char **argv ) {
43
44     /* BST hieronder */
45
46     cout << "BST:" << endl;
47     BinarySearchTree<int> bst;
48
49     /* auto root =bst.pushBack( 10 );
50     bst.pushBack( 5 );
51     bst.pushBack( 15 );
52
53     bst.pushBack( 25 );
54     bst.pushBack( 1 );
55     bst.pushBack( -1 );
56     bst.pushBack( 11 );
57     bst.pushBack( 12 );*/
58
59     Tree<int>* bstP =&bst; // Dit werkt gewoon :- )
60
61     auto root =bstP->pushBack( 10 );
62     bstP->pushBack( 5 );
63     bstP->pushBack( 15 );
64
65     bstP->pushBack( 25 );
66     bstP->pushBack( 1 );
67     bstP->pushBack( -1 );
68     bstP->pushBack( 11 );
69     bstP->pushBack( 12 );
70
71     //printTree<int>( bst, 5 );
72
73
74     //bst.remove( bst.find( 0, 15 ) );
75     //bst.replace( -2, bst.find( 0, 5 ) );
76
77
78     printTree<int>( bst, 5 );
79
80     bst.remove( root );
81
82
83     printTree<int>( bst, 5 );
84
85     /* Splay Trees hieronder */

```

```

86
87     cout << "Splay Boom:" << endl;
88     SplayTree<int> splay;
89
90     splay.pushBack( 10 );
91     auto a =splay.pushBack( 5 );
92     splay.pushBack( 15 );
93
94     splay.pushBack( 25 );
95     auto b =splay.pushBack( 1 );
96     splay.pushBack( -1 );
97     auto c =splay.pushBack( 11 );
98     splay.pushBack( 12 );
99
100    //printTree<int>( splay, 5 );
101
102    //a->swapWith( b );
103    //splay.remove( splay.find( 0, 15 ) );
104    //splay.replace( -2, splay.find( 0, 5 ) );
105
106
107    printTree<int>( splay, 5 );
108
109    //splay.remove( root );
110
111    splay.splay( c );
112
113    printTree<int>( splay, 5 );
114
115    // Test AVLTree //
116
117    AVLTree<int> test;
118    test.insert(2);
119    auto d =test.insert(4);
120    test.insert(8);
121    test.insert(7);
122    test.insert(6);
123    test.insert(1);
124    test.insert(0);
125    cout << "AVL Boompje:" << endl;
126    printTree<int>( test, 5 );
127    cout << d->info( ) << " verwijderen: " << endl;
128    test.remove( d );
129    printTree<int>( test, 5 );
130
131    // Test Treap //
132
133    cout << "Treap" << endl;
134
135    Treap<int> testTreap;
136    testTreap.insert(2);
137    testTreap.insert(3);
138    auto e =testTreap.insert(4);
139    testTreap.insert(5);

```

```

140     printTree<int>( testTreap, 5 );
141     testTreap.remove(e);
142     printTree<int>( testTreap, 5 );
143
144
145     return 0;
146 }

```

## hooiberg.cc

```

1  /**
2   * hooiberg.cc:
3   *
4   * @author Micky Faas (s1407937)
5   * @author Lisette de Schipper (s1396250)
6   * @file helehogebomen.cc
7   * @date 10-12-2014
8   */
9
10 #include "BinarySearchTree.h"
11 #include "Tree.h"
12 #include "AVLTree.h"
13 #include "SplayTree.h"
14 #include "Treap.h"
15
16 #include <iostream>
17 #include <string>
18 #include <fstream>
19 #include <vector>
20 #include <chrono>
21
22 // Only works on *nix operating systems
23 // Needed for precision timing
24 #include <sys/time.h>
25
26 using namespace std;
27
28 // Makkelijk voor debuggen, moet nog beter
29 template<class T> void printTree( Tree<T> tree, int rows ) {
30     typename Tree<T>::nodelist list =tree.row( 0 );
31     int row =0;
32     while( !list.empty( ) && row < rows ) {
33         string offset;
34         for( int i =0; i < ( 1 << (rows - row) ) - 1 ; ++i )
35             offset += ' ';
36
37
38         for( auto it =list.begin( ); it != list.end( ); ++it ) {
39             if( *it )
40                 cout << offset << (*it)->info() << " " << offset;
41             else
42                 cout << offset << ". " << offset;
43         }
44         cout << endl;

```

```

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

```

```

99         << "Number of misses: " << notfound << endl
100         << "Average search depth (hits): " << steps/found << endl;
101     }
102
103     int main ( int argc, char **argv ) {
104
105         enum MODE { NONE =0, BST, AVL, SPLAY, TREAP };
106         int mode =NONE;
107
108         if( argc < 4 )
109             return printUsage( argv[0] );
110
111         if( std::string( argv[1] ) == "bst" )
112             mode =BST;
113         else if( std::string( argv[1] ) == "avl" )
114             mode =AVL;
115         else if( std::string( argv[1] ) == "treap" )
116             mode =TREAP;
117         if( std::string( argv[1] ) == "splay" )
118             mode =SPLAY;
119
120         if( !mode )
121             return printUsage( argv[0] );
122
123         std::ifstream fhaystack( argv[2] );
124         if( !fhaystack.good( ) ) {
125             std::cerr << "Could not open " << argv[2] << std::endl;
126             return -1;
127         }
128
129         std::ifstream fneedles( argv[3] );
130         if( !fneedles.good( ) ) {
131             std::cerr << "Could not open " << argv[3] << std::endl;
132             return -1;
133         }
134
135         std::vector<string> needles;
136         if( !extractNeedles( needles, fneedles ) ) {
137             cerr << "Could not read a set of strings to search for." << endl;
138             return -1;
139         }
140
141         BinarySearchTree<string> *tree;
142         switch( mode ) {
143             case BST:
144                 tree = new BinarySearchTree<string>();
145                 break;
146             case AVL:
147                 tree = new AVLTree<string>();
148                 break;
149             case SPLAY:
150                 tree = new SplayTree<string>();
151                 break;
152             case TREAP:

```



```

153         tree = new Treap<string>();
154         break;
155     }
156
157
158     // Define a start point to time measurement
159     auto start = std::chrono::system_clock::now();
160
161
162     if( !fillTree( tree, fhaystack ) ) {
163         cerr << "Could not read the haystack." << endl;
164         return -1;
165     }
166
167     // Determine the duration of the code block
168     auto duration =std::chrono::duration_cast<std::chrono::milliseconds>
169         (std::chrono::system_clock::now() - start);
170
171     cout << "Filled the binary search tree in " << duration.count() << "ms" << endl;
172
173     start = std::chrono::system_clock::now();
174     findAll( needles, tree );
175     duration =std::chrono::duration_cast<std::chrono::milliseconds>
176         (std::chrono::system_clock::now() - start);
177
178     cout << "Searched the haystack in " << duration.count() << "ms" << endl;
179
180     // Test pre-order
181     //for( auto word : *tree ) {
182     //    cout << word << '\n';
183     //}
184
185     fhaystack.close( );
186     fneedles.close( );
187     delete tree;
188
189     return 0;
190 }

```

## Tree.h

```

1  /**
2   * Tree:
3   *
4   * @author Micky Faas (s1407937)
5   * @author Lisette de Schipper (s1396250)
6   * @file tree.h
7   * @date 26-10-2014
8   */
9
10 #ifndef TREE_H
11 #define TREE_H
12 #include "TreeNodeIterator.h"
13 #include <assert.h>

```

```

14 #include <list>
15 #include <map>
16
17 using namespace std;
18
19 template <class INFO_T> class SplayTree;
20
21 template <class INFO_T> class Tree
22 {
23     public:
24         enum ReplaceBehavoir {
25             DELETE_EXISTING,
26             ABORT_ON_EXISTING,
27             MOVE_EXISTING
28         };
29
30         typedef TreeNode<INFO_T> node_t;
31         typedef TreeNodeIterator<INFO_T> iterator;
32         typedef TreeNodeIterator_in<INFO_T> iterator_in;
33         typedef TreeNodeIterator_pre<INFO_T> iterator_pre;
34         typedef TreeNodeIterator_post<INFO_T> iterator_post;
35         typedef list<node_t*> nodelist;
36
37         /**
38          * @function   Tree( )
39          * @abstract   Constructor of an empty tree
40          */
41         Tree( )
42             : m_root( 0 ) {
43         }
44
45         /**
46          * @function   Tree( )
47          * @abstract   Copy-constructor of a tree. The new tree contains the nodes
48          *             from the tree given in the parameter (deep copy)
49          * @param      tree, a tree
50          */
51         Tree( const Tree<INFO_T>& tree )
52             : m_root( 0 ) {
53             *this =tree;
54         }
55
56         /**
57          * @function   ~Tree( )
58          * @abstract   Destructor of a tree. Timber.
59          */
60         ~Tree( ) {
61             clear( );
62         }
63
64         /**
65          * @function   begin_pre( )
66          * @abstract   begin point for pre-order iteration
67          * @return      iterator_pre containing the beginning of the tree in

```

```

68         *           pre-order
69     **/
70     iterator_pre begin_pre( ) {
71         // Pre-order traversal starts at the root
72         return iterator_pre( m_root );
73     }
74
75     /**
76     * @function    begin( )
77     * @abstract    begin point for a pre-order iteration
78     * @return      containing the beginning of the pre-Order iteration
79     **/
80     iterator_pre begin( ) {
81         return begin_pre( );
82     }
83
84     /**
85     * @function    end( )
86     * @abstract    end point for a pre-order iteration
87     * @return      the end of the pre-order iteration
88     **/
89     iterator_pre end( ) {
90         return iterator_pre( (node_t*)0 );
91     }
92
93     /**
94     * @function    end_pre( )
95     * @abstract    end point for pre-order iteration
96     * @return      iterator_pre containing the end of the tree in pre-order
97     **/
98     iterator_pre end_pre( ) {
99         return iterator_pre( (node_t*)0 );
100    }
101
102    /**
103    * @function    begin_in( )
104    * @abstract    begin point for in-order iteration
105    * @return      iterator_in containing the beginning of the tree in
106    *              in-order
107    **/
108    iterator_in begin_in( ) {
109        if( !m_root )
110            return end_in( );
111        node_t *n =m_root;
112        while( n->leftChild( ) )
113            n =n->leftChild( );
114        return iterator_in( n );
115    }
116
117    /**
118    * @function    end_in( )
119    * @abstract    end point for in-order iteration
120    * @return      iterator_in containing the end of the tree in in-order
121    **/

```

```

122     iterator_in end_in( ) {
123         return iterator_in( (node_t*)0 );
124     }
125
126 /**
127  * @function   begin_post( )
128  * @abstract   begin point for post-order iteration
129  * @return     iterator_post containing the beginning of the tree in
130  *             post-order
131  */
132     iterator_post begin_post( ) {
133         if( !m_root )
134             return end_post( );
135         node_t *n =m_root;
136         while( n->leftChild( ) )
137             n =n->leftChild( );
138         return iterator_post( n );
139     }
140
141 /**
142  * @function   end_post( )
143  * @abstract   end point for post-order iteration
144  * @return     iterator_post containing the end of the tree in post-order
145  */
146     iterator_post end_post( ) {
147         return iterator_post( (node_t*)0 );
148     }
149
150 /**
151  * @function   pushBack( )
152  * @abstract   a new TreeNode containing 'info' is added to the end
153  *             the node is added to the node that :
154  *             - is in the row as close to the root as possible
155  *             - has no children or only a left-child
156  *             - seen from the right hand side of the row
157  *             this is the 'natural' left-to-right filling order
158  *             compatible with array-based heaps and full b-trees
159  * @param      info, the contents of the new node
160  * @post       A node has been added.
161  */
162     virtual node_t *pushBack( const INFO_T& info ) {
163         node_t *n =new node_t( info, 0 );
164         if( !m_root ) { // Empty tree, simplest case
165             m_root =n;
166         }
167         else { // Leaf node, there are two different scenarios
168             int max =getRowCountRecursive( m_root, 0 );
169             node_t *parent;
170             for( int i =1; i <= max; ++i ) {
171
172                 parent =getFirstEmptySlot( i );
173                 if( parent ) {
174                     if( !parent->leftChild( ) )
175                         parent->setLeftChild( n );

```

```

176         else if( !parent->rightChild( ) )
177             parent->setRightChild( n );
178         n->setParent( parent );
179         break;
180     }
181 }
182 }
183     return n;
184 }
185
186 /**
187  * @function insert( )
188  * @abstract inserts node or subtree under a parent or creates an empty
189  *             root node
190  * @param info, contents of the new node
191  * @param parent, parent node of the new node. When zero, the root is
192  *             assumed
193  * @param alignRight, insert() checks on which side of the parent
194  *             node the new node can be inserted. By default, it checks
195  *             the left side first.
196  *             To change this behavior, set preferRight =true.
197  * @param replaceBehavior, action if parent already has two children.
198  *             One of:
199  *             ABORT_ON_EXISTING - abort and return zero
200  *             MOVE_EXISTING - make the parent's child a child of the new
201  *                             node, satisfies preferRight
202  *             DELETE_EXISTING - remove one of the children of parent
203  *                             completely also satisfies preferRight
204  * @return pointer to the inserted TreeNode, if insertion was
205  *             successfull
206  * @pre If the tree is empty, a root node will be created with info
207  *       as it contents
208  * @pre The instance pointed to by parent should be part of the
209  *       called instance of Tree
210  * @post Return zero if no node was created. Ownership is assumed on
211  *       the new node.
212  *       When DELETE_EXISTING is specified, the entire subtree on
213  *       preferred side may be deleted first.
214  */
215 virtual node_t* insert( const INFO_T& info,
216                        node_t* parent =0,
217                        bool preferRight =false,
218                        int replaceBehavior =ABORT_ON_EXISTING ) {
219     if( !parent )
220         parent =m_root;
221
222     if( !parent )
223         return pushBack( info );
224
225     node_t *node =0;
226
227     if( !parent->leftChild( )
228         && ( !preferRight || ( preferRight &&
229             parent->rightChild( ) ) ) ) {

```

```

230         node =new node_t( info, parent );
231         parent->setLeftChild( node );
232         node->setParent( parent );
233
234     } else if( !parent->rightChild( ) ) {
235         node =new node_t( info, parent );
236         parent->setRightChild( node );
237         node->setParent( parent );
238
239     } else if( replaceBehavior == MOVE_EXISTING ) {
240         node =new node_t( info, parent );
241         if( preferRight ) {
242             node->setRightChild( parent->rightChild( ) );
243             node->rightChild( )->setParent( node );
244             parent->setRightChild( node );
245         } else {
246             node->setLeftChild( parent->leftChild( ) );
247             node->leftChild( )->setParent( node );
248             parent->setLeftChild( node );
249         }
250
251     } else if( replaceBehavior == DELETE_EXISTING ) {
252         node =new node_t( info, parent );
253         if( preferRight ) {
254             deleteRecursive( parent->rightChild( ) );
255             parent->setRightChild( node );
256         } else {
257             deleteRecursive( parent->leftChild( ) );
258             parent->setLeftChild( node );
259         }
260
261     }
262     return node;
263 }
264
265 /**
266  * @function replace( )
267  * @abstract replaces an existing node with a new node
268  * @param info, contents of the new node
269  * @param node, node to be replaced. When zero, the root is assumed
270  * @param alignRight, only for MOVE_EXISTING. If true, node will be
271  * the right child of the new node. Otherwise, it will be the
272  * left.
273  * @param replaceBehavior, one of:
274  * ABORT_ON_EXISTING - undefined for replace()
275  * MOVE_EXISTING - make node a child of the new node,
276  * satisfies preferRight
277  * DELETE_EXISTING - remove node completely
278  * @return pointer to the inserted TreeNode, replace() is always
279  * successful
280  * @pre If the tree is empty, a root node will be created with info
281  * as it contents
282  * @pre The instance pointed to by node should be part of the
283  * called instance of Tree

```

```

284      * @post      Ownership is assumed on the new node. When DELETE_EXISTING
285      *            is specified, the entire subtree pointed to by node is
286      *            deleted first.
287      **/
288      virtual node_t* replace( const INFO_T& info,
289                              node_t* node =0,
290                              bool alignRight =false ,
291                              int replaceBehavior =DELETE_EXISTING ) {
292          assert( replaceBehavior != ABORT_ON_EXISTING );
293
294          node_t *newnode =new node_t( info );
295          if( !node )
296              node =m_root;
297          if( !node )
298              return pushBack( info );
299
300          if( node->parent( ) ) {
301              newnode->setParent( node->parent( ) );
302              if( node->parent( )->leftChild( ) == node )
303                  node->parent( )->setLeftChild( newnode );
304              else
305                  node->parent( )->setRightChild( newnode );
306          } else
307              m_root =newnode;
308
309          if( replaceBehavior == DELETE_EXISTING ) {
310
311              deleteRecursive( node );
312          }
313          else if( replaceBehavior == MOVE_EXISTING ) {
314              if( alignRight )
315                  newnode->setRightChild( node );
316              else
317                  newnode->setLeftChild( node );
318              node->setParent( newnode );
319          }
320          return node;
321      }
322
323      /**
324      * @function   remove( )
325      * @abstract   removes and deletes node or subtree
326      * @param      n, node or subtree to be removed and deleted
327      * @post       after remove(), n points to an invalid address
328      **/
329      virtual void remove( node_t *n ) {
330          if( !n )
331              return;
332          if( n->parent( ) ) {
333              if( n->parent( )->leftChild( ) == n )
334                  n->parent( )->setLeftChild( 0 );
335              else if( n->parent( )->rightChild( ) == n )
336                  n->parent( )->setRightChild( 0 );
337          }

```

```

338         deleteRecursive( n );
339     }
340
341 /**
342  * @function   clear( )
343  * @abstract   clears entire tree
344  * @pre        tree may be empty
345  * @post       all nodes and data are deallocated
346  */
347 void clear( ) {
348     deleteRecursive( m_root );
349     m_root =0;
350 }
351
352 /**
353  * @function   empty( )
354  * @abstract   test if tree is empty
355  * @return     true when empty
356  */
357 bool isEmpty( ) const {
358     return !m_root;
359 }
360
361 /**
362  * @function   root( )
363  * @abstract   returns address of the root of the tree
364  * @return     the adress of the root of the tree is returned
365  * @pre        there needs to be a tree
366  */
367 node_t* root( ){
368     return m_root;
369 }
370
371 /**
372  * @function   row( )
373  * @abstract   returns an entire row/level in the tree
374  * @param      level, the desired row. Zero gives just the root.
375  * @return     a list containing all node pointers in that row
376  * @pre        level must be positive or zero
377  * @post
378  */
379 nodelist row( int level ) {
380     nodelist rlist;
381     getRowRecursive( m_root, rlist, level );
382     return rlist;
383 }
384
385 /**
386  * @function   find( )
387  * @abstract   find the first occurrence of info and returns its node ptr
388  * @param      haystack, the root of the (sub)tree we want to look in
389  *             null if we want to start at the root of the tree
390  * @param      needle, the needle in our haystack
391  * @return     a pointer to the first occurrence of needle

```



```

392     * @post      there may be multiple occurrences of needle, we only return
393     *            one. A null-pointer is returned if no needle is found
394     **/
395     virtual node_t* find( node_t* haystack, const INFO_T& needle ) {
396         if( haystack == 0 ) {
397             if( m_root )
398                 haystack =m_root;
399             else
400                 return 0;
401         }
402         return findRecursive( haystack, needle );
403     }
404
405     /**
406     * @function    contains( )
407     * @abstract    determines if a certain content (needle) is found
408     * @param       haystack, the root of the (sub)tree we want to look in
409     *              null if we want to start at the root of the tree
410     * @param       needle, the needle in our haystack
411     * @return      true if needle is found
412     **/
413     bool contains( node_t* haystack, const INFO_T& needle ) {
414         return find( haystack, needle );
415     }
416
417     /**
418     * @function    toDot( )
419     * @abstract    writes tree in Dot-format to a stream
420     * @param       out, ostream to write to
421     * @pre         out must be a valid stream
422     * @post        out (file or cout) with the tree in dot-notation
423     **/
424     void toDot( ostream& out, const string & graphName ) {
425         if( isEmpty( ) )
426             return;
427         map< node_t *, int> addresses;
428         typename map< node_t *, int >::iterator adrIt;
429         int i =1;
430         int p;
431         iterator_pre it;
432         iterator_pre tempit;
433         addresses[m_root] =0;
434         out << "digraph " << graphName << '{ ' << endl << "' ' << 0 << "' ";
435         for( it =begin_pre( ); it != end_pre( ); ++it ) {
436             adrIt =addresses.find( &(*it) );
437             if( adrIt == addresses.end( ) ) {
438                 addresses[&(*it)] =i;
439                 p =i;
440                 i ++;
441             }
442             if( (&(*it))->parent( ) != &(*tempit) )
443                 out << ' ' << endl << "' '
444                     << addresses.find( (&(*it))->parent( ))->second << "' ";
445             if( (&(*it)) != m_root )

```

```

446         out << " ->  \" << p << \"';
447         tempit =it;
448     }
449     out << ',' << endl;
450     for ( adrIt =addresses.begin( ); adrIt != addresses.end( ); ++adrIt )
451         out << adrIt->second << " [label=\"
452             << adrIt->first->info( ) << "\"]";
453     out << '}'';
454 }
455
456 /**
457  * @function   copyFromNode( )
458  * @abstract   copies the the node source and its children to the node
459  *             dest
460  * @param      source, the node and its children that need to be copied
461  * @param      dest, the node who is going to get the copied children
462  * @param      left, this is true if it's a left child.
463  * @pre        there needs to be a tree and we can't copy to a root.
464  * @post       the subtree that starts at source is now also a child of
465  *             dest
466  */
467 void copyFromNode( node_t *source, node_t *dest, bool left ) {
468     if (!source)
469         return;
470     node_t *acorn =new node_t( dest );
471     if(left) {
472         if( dest->leftChild( ))
473             return;
474         dest->setLeftChild( acorn );
475     }
476     else {
477         if( dest->rightChild( ))
478             return;
479         dest->setRightChild( acorn );
480     }
481     cloneRecursive( source, acorn );
482 }
483
484 Tree<INFO_T>& operator=( const Tree<INFO_T>& tree ) {
485     clear( );
486     if( tree.m_root ) {
487         m_root =new node_t( (node_t*)0 );
488         cloneRecursive( tree.m_root, m_root );
489     }
490     return *this;
491 }
492
493 protected:
494 /**
495  * @function   cloneRecursive( )
496  * @abstract   cloning a subtree to a node
497  * @param      source, the node we want to start the cloning process from
498  * @param      dest, the node we want to clone to
499  * @post       the subtree starting at source is cloned to the node dest

```

```

500     **/
501     void cloneRecursive( node_t *source, node_t* dest ) {
502         dest->info() =source->info();
503         if( source->leftChild( ) ) {
504             node_t *left =new node_t( dest );
505             dest->setLeftChild( left );
506             cloneRecursive( source->leftChild( ), left );
507         }
508         if( source->rightChild( ) ) {
509             node_t *right =new node_t( dest );
510             dest->setRightChild( right );
511             cloneRecursive( source->rightChild( ), right );
512         }
513     }
514
515     /**
516     * @function    deleteRecursive( )
517     * @abstract    delete all nodes of a given tree
518     * @param       root, starting point, is deleted last
519     * @post        the subtree has been deleted
520     */
521     void deleteRecursive( node_t *root ) {
522         if( !root )
523             return;
524         deleteRecursive( root->leftChild( ) );
525         deleteRecursive( root->rightChild( ) );
526         delete root;
527     }
528
529     /**
530     * @function    getRowCountRecursive( )
531     * @abstract    calculate the maximum depth/row count in a subtree
532     * @param       root, starting point
533     * @param       level, starting level
534     * @return      maximum depth/rows in the subtree
535     */
536     int getRowCountRecursive( node_t* root, int level ) {
537         if( !root )
538             return level;
539         return max(
540             getRowCountRecursive( root->leftChild( ), level+1 ),
541             getRowCountRecursive( root->rightChild( ), level+1 ) );
542     }
543
544     /**
545     * @function    getRowRecursive( )
546     * @abstract    compile a full list of one row in the tree
547     * @param       root, starting point
548     * @param       rlist, reference to the list so far
549     * @param       level, how many level still to go
550     * @post        a list of a row in the tree has been made.
551     */
552     void getRowRecursive( node_t* root, nodelist &rlist, int level ) {
553         // Base-case

```

```

554         if( !level ) {
555             rlist.push_back( root );
556         } else if( root ){
557             level--;
558             if( level && !root->leftChild( ) )
559                 for( int i =0; i < (level<<1); ++i )
560                     rlist.push_back( 0 );
561             else
562                 getRowRecursive( root->leftChild( ), rlist, level );
563
564             if( level && !root->rightChild( ) )
565                 for( int i =0; i < (level<<1); ++i )
566                     rlist.push_back( 0 );
567             else
568                 getRowRecursive( root->rightChild( ), rlist, level );
569         }
570     }
571
572     /**
573     * @function   findRecursive( )
574     * @abstract   first the first occurrence of needle and return its node
575     *             ptr
576     * @param      haystack, root of the search tree
577     * @param      needle, copy of the data to find
578     * @return     the node that contains the needle
579     */
580     node_t *findRecursive( node_t* haystack, const INFO_T &needle ) {
581         if( haystack->info( ) == needle )
582             return haystack;
583
584         node_t *n =0;
585         if( haystack->leftChild( ) )
586             n =findRecursive( haystack->leftChild( ), needle );
587         if( !n && haystack->rightChild( ) )
588             n =findRecursive( haystack->rightChild( ), needle );
589         return n;
590     }
591
592     friend class TreeNodeIterator_pre<INFO_T>;
593     friend class TreeNodeIterator_in<INFO_T>;
594     friend class SplayTree<INFO_T>;
595     TreeNode<INFO_T> *m_root;
596
597 private:
598     /**
599     * @function   getFirstEmptySlot( )
600     * @abstract   when a row has a continuous empty space on the right,
601     *             find the left-most parent in the above row that has
602     *             at least one empty slot.
603     * @param      level, how many level still to go
604     * @return     the first empty slot where we can put a new node
605     * @pre        level should be > 1
606     */
607     node_t *getFirstEmptySlot( int level ) {

```

```

608         node_t *p =0;
609         nodelist rlist =row( level-1 ); // we need the parents of this level
610         /** changed auto to int **/
611         for( auto it =rlist.rbegin( ); it !=rlist.rend( ); ++it ) {
612             if( !(*it)->hasChildren( ) )
613                 p =(*it);
614             else if( !(*it)->rightChild( ) ) {
615                 p =(*it);
616                 break;
617             } else
618                 break;
619         }
620         return p;
621     }
622 };
623
624 #endif

```

## TreeNode.h

```

1  /**
2   * Treenode:
3   *
4   * @author Micky Faas (s1407937)
5   * @author Lisette de Schipper (s1396250)
6   * @file   Treenode.h
7   * @date   26-10-2014
8   */
9
10 #ifndef TREENODE.H
11 #define TREENODE.H
12
13 using namespace std;
14
15 template <class INFO_T> class Tree;
16 class ExpressionTree;
17
18 template <class INFO_T> class TreeNode
19 {
20     public:
21         /**
22          * @function   TreeNode( )
23          * @abstract   Constructor, creates a node
24          * @param      info, the contents of a node
25          * @param      parent, the parent of the node
26          * @post       A node has been created.
27          */
28         TreeNode( const INFO_T& info, TreeNode<INFO_T>* parent =0 )
29             : m_lchild( 0 ), m_rchild( 0 ) {
30             m_info =info;
31             m_parent =parent;
32         }
33
34         /**

```

```

35     * @function   TreeNode( )
36     * @abstract   Constructor, creates a node
37     * @param      parent, the parent of the node
38     * @post       A node has been created.
39     **/
40     TreeNode( TreeNode<INFO_T>* parent =0 )
41         : m_lchild( 0 ), m_rchild( 0 ) {
42         m_parent =parent;
43     }
44
45     /**
46     * @function   =
47     * @abstract   Sets a nodes content to N
48     * @param      n, the contents you want the node to have
49     * @post       The node now has those contents.
50     **/
51     void operator =( INFO_T n ) { m_info =n; }
52
53     /**
54     * @function   INFO_T( ), info( )
55     * @abstract   Returns the content of a node
56     * @return     m_info, the contents of the node
57     **/
58     operator INFO_T( ) const { return m_info; }
59     const INFO_T &info( ) const { return m_info; }
60     INFO_T &info( ) { return m_info; }
61     /**
62     * @function   atRow( )
63     * @abstract   returns the level or row-number of this node
64     * @return     row, an int of row the node is at
65     **/
66     int atRow( ) const {
67         const TreeNode<INFO_T> *n =this;
68         int row =0;
69         while( n->parent( ) ) {
70             n =n->parent( );
71             row++;
72         }
73         return row;
74     }
75
76     /**
77     * @function   parent( ), leftChild( ), rightChild( )
78     * @abstract   returns the adress of the parent, left child and right
79     *              child respectively
80     * @return     the adress of the requested family member of the node
81     **/
82     TreeNode<INFO_T> *parent( ) const { return m_parent; }
83     TreeNode<INFO_T> *leftChild( ) const { return m_lchild; }
84     TreeNode<INFO_T> *rightChild( ) const { return m_rchild; }
85
86     /**
87     * @function   swapWith( )
88     * @abstract   Swaps this node with another node in the tree

```

```

89      * @param      n, the node to swap this one with
90      * @pre        both this node and n must be in the same parent tree
91      * @post       n will have the parent and children of this node
92      *             and vice verse. Both nodes retain their data.
93      **/
94      void swapWith( TreeNode<INFO_T>* n ) {
95          bool this_wasLeftChild =false, n_wasLeftChild =false;
96          if( parent( ) && parent( )->leftChild( ) == this )
97              this_wasLeftChild =true;
98          if( n->parent( ) && n->parent( )->leftChild( ) == n )
99              n_wasLeftChild =true;
100
101          // Swap the family info
102          TreeNode<INFO_T>* newParent =
103              ( n->parent( ) == this ) ? n : n->parent( );
104          TreeNode<INFO_T>* newLeft =
105              ( n->leftChild( ) == this ) ? n : n->leftChild( );
106          TreeNode<INFO_T>* newRight =
107              ( n->rightChild( ) == this ) ? n : n->rightChild( );
108
109          n->setParent( parent( ) == n ? this : parent( ) );
110          n->setLeftChild( leftChild( ) == n ? this : leftChild( ) );
111          n->setRightChild( rightChild( ) == n ? this : rightChild( ) );
112
113          setParent( newParent );
114          setLeftChild( newLeft );
115          setRightChild( newRight );
116
117          // Restore applicable pointers
118          if( n->leftChild( ) )
119              n->leftChild( )->setParent( n );
120          if( n->rightChild( ) )
121              n->rightChild( )->setParent( n );
122          if( leftChild( ) )
123              leftChild( )->setParent( this );
124          if( rightChild( ) )
125              rightChild( )->setParent( this );
126          if( n->parent( ) ) {
127              if( this_wasLeftChild )
128                  n->parent( )->setLeftChild( n );
129              else
130                  n->parent( )->setRightChild( n );
131          }
132          if( parent( ) ) {
133              if( n_wasLeftChild )
134                  parent( )->setLeftChild( this );
135              else
136                  parent( )->setRightChild( this );
137          }
138      }
139
140      /**
141      * @function  replace( )
142      * @abstract  Replaces the node with another node in the tree

```

```

143      * @param      n, the node we replace the node with, this one gets deleted
144      * @pre        both this node and n must be in the same parent tree
145      * @post       The node will be replaced and n will be deleted.
146      **/
147      void replace( TreeNode<INFO_T>* n ) {
148          bool n_wasLeftChild =false;
149
150          if( n->parent( ) && n->parent( )->leftChild( ) == n )
151              n_wasLeftChild =true;
152
153          // Swap the family info
154          TreeNode<INFO_T>* newParent =
155              ( n->parent( ) == this ) ? n : n->parent( );
156          TreeNode<INFO_T>* newLeft =
157              ( n->leftChild( ) == this ) ? n :n->leftChild( );
158          TreeNode<INFO_T>* newRight =
159              ( n->rightChild( ) == this ) ? n :n->rightChild( );
160
161          setParent( newParent );
162          setLeftChild( newLeft );
163          setRightChild( newRight );
164          m_info = n->m_info;
165
166          // Restore applicable pointers
167          if( leftChild( ) )
168              leftChild( )->setParent( this );
169          if( rightChild( ) )
170              rightChild( )->setParent( this );
171
172          if( parent( ) ) {
173              if( n_wasLeftChild )
174                  parent( )->setLeftChild( this );
175              else
176                  parent( )->setRightChild( this );
177          }
178          delete n;
179      }
180
181
182
183      /**
184      * @function sibling( )
185      * @abstract returns the address of the sibling
186      * @return    the address to the sibling or zero if there is no sibling
187      **/
188      TreeNode<INFO_T>* sibling( ) {
189          if( parent( )->leftChild( ) == this )
190              return parent( )->rightChild( );
191          else if( parent( )->rightChild( ) == this )
192              return parent( )->leftChild( );
193          else
194              return 0;
195      }
196

```



```

197     /**
198     * @function   hasChildren( ), hasParent( ), isFull( )
199     * @abstract   Returns whether the node has children, has parents or is
200     *             full (has two children) respectively
201     * @param
202     * @return     true or false, depending on what is requested from the node.
203     *             if hasChildren is called and the node has children, it will
204     *             return true, otherwise false.
205     *             If hasParent is called and the node has a parent, it will
206     *             return true, otherwise false.
207     *             If isFull is called and the node has two children, it will
208     *             return true, otherwise false.
209     */
210     bool hasChildren( ) const { return m_lchild || m_rchild; }
211     bool hasParent( ) const { return m_parent; }
212     bool isFull( ) const { return m_lchild && m_rchild; }
213
214     protected:
215         friend class Tree<INFO_T>;
216         friend class ExpressionTree;
217
218     /**
219     * @function   setParent( ), setLeftChild( ), setRightChild( )
220     * @abstract   sets the parent, left child and right child of the
221     *             particular node respectively
222     * @param       p, the node we want to set a certain family member of
223     * @return      void
224     * @post        The node now has a parent, a left child or a right child
225     *             respectively.
226     */
227     void setParent( TreeNode<INFO_T> *p ) { m_parent =p; }
228     void setLeftChild( TreeNode<INFO_T> *p ) { m_lchild =p; }
229     void setRightChild( TreeNode<INFO_T> *p ) { m_rchild =p; }
230
231     private:
232         INFO_T m_info;
233         TreeNode<INFO_T> *m_parent;
234         TreeNode<INFO_T> *m_lchild;
235         TreeNode<INFO_T> *m_rchild;
236     };
237
238     /**
239     * @function   <<
240     * @abstract   the contents of the node are returned
241     * @param       out, in what format we want to get the contents
242     * @param       rhs, the node of which we want the contents
243     * @return      the contents of the node.
244     */
245     template <class INFO_T> ostream &operator <<(ostream& out, const TreeNode<INFO_T> & r
246         out << rhs.info( );
247         return out;
248     }
249
250 #endif

```

## TreeNodeIterator.h

```
1  /**
2   * TreeNodeIterator: Provides a set of iterators that follow the STL-standard
3   *
4   * @author Micky Faas (s1407937)
5   * @author Lisette de Schipper (s1396250)
6   * @file   TreeNodeIterator.h
7   * @date   26-10-2014
8   */
9
10 #include <iterator>
11 #include "TreeNode.h"
12
13 template <class INFO_T> class TreeNodeIterator
14     : public std::iterator<std::forward_iterator_tag,
15                           TreeNode<INFO_T>> {
16 public:
17     typedef TreeNode<INFO_T> node_t;
18
19     /**
20      * @function   TreeNodeIterator( )
21      * @abstract   (copy)constructor
22      * @pre        TreeNodeIterator is abstract and cannot be constructed
23      */
24     TreeNodeIterator( node_t* ptr = 0 ) : p( ptr ) { }
25     TreeNodeIterator( const TreeNodeIterator& it ) : p( it.p ) { }
26
27     /**
28      * @function   (in)equality operator overload
29      * @abstract   Test (in)equality for two TreeNodeIterators
30      * @param      rhs, right-hand side of the comparison
31      * @return     true if both iterators point to the same node (==)
32      *             false if both iterators point to the same node (!=)
33      */
34     bool operator == (const TreeNodeIterator& rhs) { return p==rhs.p; }
35     bool operator != (const TreeNodeIterator& rhs) { return p!=rhs.p; }
36
37     /**
38      * @function   operator*( )
39      * @abstract   Cast operator to node_t reference
40      * @return     The value of the current node
41      * @pre        Must point to a valid node
42      */
43     node_t& operator*( ) { return *p; }
44
45     /**
46      * @function   operator++( )
47      * @abstract   pre- and post increment operators
48      * @return     TreeNodeIterator that has iterated one step
49      */
50     TreeNodeIterator &operator++( ) { next( ); return *this; }
51     TreeNodeIterator operator++( int )
52     { TreeNodeIterator tmp( *this ); operator++( ); return tmp; }
```

```

53     protected:
54
55         /**
56          * @function next( ) //(pure virtual)
57          * @abstract Implement this function to implement your own iterator
58          */
59         virtual bool next( ){ return false; }// =0;
60         node_t *p;
61     };
62
63     template <class INFO_T> class TreeNodeIterator_pre
64         : public TreeNodeIterator<INFO_T> {
65     public:
66         typedef TreeNode<INFO_T> node_t;
67
68         TreeNodeIterator_pre( node_t* ptr =0 )
69             : TreeNodeIterator<INFO_T>( ptr ) { }
70         TreeNodeIterator_pre( const TreeNodeIterator<INFO_T>& it )
71             : TreeNodeIterator<INFO_T>( it ) { }
72         TreeNodeIterator_pre( const TreeNodeIterator_pre& it )
73             : TreeNodeIterator<INFO_T>( it.p ) { }
74
75         TreeNodeIterator_pre &operator++( ) { next( ); return *this; }
76         TreeNodeIterator_pre operator++( int )
77             { TreeNodeIterator_pre tmp( *this ); operator++( ); return tmp; }
78
79     protected:
80         using TreeNodeIterator<INFO_T>::p;
81
82         /**
83          * @function next( )
84          * @abstract Takes one step in pre-order traversal
85          * @return returns true if such a step exists
86          */
87         bool next( ) {
88             if( !p )
89                 return false;
90             if( p->hasChildren( ) ) { // a possible child that can be the next
91                 p =p->leftChild( ) ? p->leftChild( ) : p->rightChild( );
92                 return true;
93             }
94             else if( p->hasParent( ) // we have a right brother
95                     && p->parent( )->rightChild( )
96                     && p->parent( )->rightChild( ) != p ) {
97                 p =p->parent( )->rightChild( );
98                 return true;
99             }
100             else if( p->hasParent( ) ) { // just a parent, thus we go up
101                 TreeNode<INFO_T> *tmp =p->parent( );
102                 while( tmp->parent( ) ) {
103                     if( tmp->parent( )->rightChild( )
104                         && tmp->parent( )->rightChild( ) != tmp ) {
105                         p =tmp->parent( )->rightChild( );
106                         return true;

```

```

107         }
108         tmp =tmp->parent( );
109     }
110 }
111 // Nothing left
112 p =0;
113 return false;
114 }
115
116 };
117
118 template <class INFO_T> class TreeNodeIterator_in
119     : public TreeNodeIterator<INFO_T>{
120 public:
121     typedef TreeNode<INFO_T> node_t;
122
123     TreeNodeIterator_in( node_t* ptr =0 )
124         : TreeNodeIterator<INFO_T>( ptr ) { }
125     TreeNodeIterator_in( const TreeNodeIterator<INFO_T>& it )
126         : TreeNodeIterator<INFO_T>( it ) { }
127     TreeNodeIterator_in( const TreeNodeIterator_in& it )
128         : TreeNodeIterator<INFO_T>( it.p ) { }
129
130     TreeNodeIterator_in &operator++( ) { next( ); return *this; }
131     TreeNodeIterator_in operator++( int )
132         { TreeNodeIterator_in tmp( *this ); operator++( ); return tmp; }
133
134 protected:
135     using TreeNodeIterator<INFO_T>::p;
136     /**
137     * @function next( )
138     * @abstract Takes one step in in-order traversal
139     * @return returns true if such a step exists
140     */
141     bool next( ) {
142         if( p->rightChild( ) ) {
143             p =p->rightChild( );
144             while( p->leftChild( ) )
145                 p =p->leftChild( );
146             return true;
147         }
148         else if( p->parent( ) && p->parent( )->leftChild( ) == p ) {
149             p =p->parent( );
150             return true;
151         } else if( p->parent( ) && p->parent( )->rightChild( ) == p ) {
152             p =p->parent( );
153             while( p->parent( ) && p == p->parent( )->rightChild( ) ) {
154                 p =p->parent( );
155             }
156             if( p )
157                 p =p->parent( );
158             if( p )
159                 return true;
160             else

```

```

161         return false;
162     }
163     // Er is niks meer
164     p =0;
165     return false;
166 }
167 };
168
169 template <class INFO_T> class TreeNodeIterator_post
170     : public TreeNodeIterator<INFO_T>{
171     public:
172         typedef TreeNode<INFO_T> node_t;
173
174         TreeNodeIterator_post( node_t* ptr =0 )
175             : TreeNodeIterator<INFO_T>( ptr ) { }
176         TreeNodeIterator_post( const TreeNodeIterator<INFO_T>& it )
177             : TreeNodeIterator<INFO_T>( it ) { }
178         TreeNodeIterator_post( const TreeNodeIterator_post& it )
179             : TreeNodeIterator<INFO_T>( it.p ) { }
180
181         TreeNodeIterator_post &operator++( ) { next( ); return *this; }
182         TreeNodeIterator_post operator++( int )
183             { TreeNodeIterator_post tmp( *this ); operator++( ); return tmp; }
184
185     protected:
186         using TreeNodeIterator<INFO_T>::p;
187         /**
188          * @function next( )
189          * @abstract Takes one step in post-order traversal
190          * @return returns true if such a step exists
191          */
192         bool next( ) {
193
194             if( p->hasParent( ) // We have a right brother
195                 && p->parent( )->rightChild( )
196                 && p->parent( )->rightChild( ) != p ) {
197                 p =p->parent( )->rightChild( );
198                 while( p->leftChild( ) )
199                     p =p->leftChild( );
200                 return true;
201             } else if( p->parent( ) ) {
202                 p =p->parent( );
203                 return true;
204             }
205             // Nothing left
206             p =0;
207             return false;
208         }
209     };

```