Swinburne University of Technology

Faculty of Science, Engineering and Technology

ASSIGNMENT COVER SHEET

Subject Code: COS30008
Subject Title: Data Structures and Patterns
Assignment number and title: 4, Binary Search Trees & In-Order Traversal
Due date: May 26, 2022, 14:30
Lecturer: Dr. Markus Lumpe

Your name: Jamie Kozminska **Your student id:** 101114436

Check Tutorial	Mon 10:30	Mon 14:30	Tues 08:30	Tues 10:30	Tues 12:30	Tues 14:30	Tues 16:30	Wed 08:30	Wed 10:30	Wed 12:30	Wed 14:30
						X					

Marker's comments:

Problem	Marks	Obtained
1	94	
2	42	
3	8+86=94	
Total	230	

Extension certification:

This assignment has been given an extension and is now due on _____

Signature of Convener:_____

```
 1
 2  // COS30008, Problem Set 4, Problem 1, 2022
 3
 4  #pragma once
 5
 6  #include <stdexcept>
 7  #include <algorithm>
 8
 9  template<typename T>
10  struct BinaryTreeNode
11  {
12      using BNode = BinaryTreeNode<T>; //BNode represents the btree
13      using BTreeNode = BNode*; //BtreeNode represents the pointer
14
15      T key; //data
16      BTreeNode left;
17      BTreeNode right;
18
19      static BNode NIL;
20
21      const T& findMax() const
22      {
23          if ( empty() )
24          {
25              throw std::domain_error( "Empty tree encountered." );
26          }
27
28          return right->empty() ? key : right->findMax(); // if right is empty
              then return the node, otherwise keep going
29      }
30
31      const T& findMin() const
32      {
33          if ( empty() )
34          {
35              throw std::domain_error( "Empty tree encountered." );
36          }
37
38          return left->empty() ? key : left->findMin(); // if left is empty then
              return the node, otherwise keep going
39      }
40
41      bool remove( const T& aKey, BTreeNode aParent )
42      {
43          BTreeNode x = this; // BNode<S>* x
44          BTreeNode y = aParent;
45
46          while ( !x->empty() ) //while current node is not empty
47          {
```

```
48              if ( aKey == x->key ) //if the given key matches the node we break
49              {
50                  break;
51              }
52
53              y = x;                                          // new parent (current node becomes the parent)
54
55              x = aKey < x->key ? x->left : x->right;         //ternary for if the given key is less than current then its set to left, otherwise set to right
56          }
57
58          if ( x->empty() )
59          {
60              return false;                                   // delete failed
61          }
62
63          if ( !x->left->empty() )
64          {
65              const T& lKey = x->left->findMax();         // find max to left
66              x->key = lKey;
67              x->left->remove( lKey, x );
68          }
69          else
70          {
71              if ( !x->right->empty() )
72              {
73                  const T& lKey = x->right->findMin();    // find min to right
74                  x->key = lKey;
75                  x->right->remove( lKey, x );
76              }
77              else
78              {
79                  if ( y != &NIL )                        // y can be NIL
80                  {
81                      if ( y->left == x )
82                      {
83                          y->left = &NIL;
84                      }
85                      else
86                      {
87                          y->right = &NIL;
88                      }
89                  }
90
91                  delete x;                               // free deleted node
92              }
93          }
```

```
94
95          return true;
96      }
97
98      // PS4 starts here
99
100
101     ~BinaryTreeNode()
102     {
103         if (!empty())
104         {
105             if (left != &NIL)
106             {
107                 delete left;
108             }
109             if (right != &NIL)
110             {
111                 delete right;
112             }
113         }
114     }
115
116     const bool empty()
117     {
118         return this == &NIL;
119     }
120
121     const bool leaf()
122      {
123          return left == &NIL && right == &NIL;
124      }
125
126
127     const size_t height()
128     {
129         if (empty())
130         {
131             throw std::domain_error("Empty tree encountered.");
132         }
133
134         if (leaf())
135         {
136             return 0;
137         }
138
139         size_t left_side = 0;
140         try {
141             left_side = this->left->height(); //get the max height of the left
                    subtree recursively(could also do it iteratively)
```

```
142         }
143         catch(domain_error e) {
144             left_side = 0;
145         }
146 
147         size_t right_side = 0;
148         try {
149              right_side = this->right->height(); //get the max height of the
                  right subtree
150         }
151         catch (domain_error e) {
152             right_side = 0;
153         }
154 
155         if (left_side > right_side) //compare which side is higher + 1  and
              return max height
156         {
157             return left_side + 1;
158         }
159         else
160         {
161             return right_side + 1;
162         }
163     }
164 
165     const size_t max(size_t left, size_t right)
166     {
167         if (left > right) {
168             return left;
169         }
170         return right;
171     }
172 
173 
174     bool insert(const T& aKey)
175     {
176         BTreeNode x = this;  //  BTreeNode x == BNode<S>* x
177         BTreeNode y = &NIL;
178 
179         while (!x->empty()) //if root node is NOT empty
180         {
181             y = x;  //copy it into y
182 
183             if (aKey == x->key) // if there is a duplicate key then theres an
                  error
184             {
185                 return false;
186             }
187 
```

```
188             x = aKey < x->key ? x->left : x->right; //otherwise if the key is less than current insert to left otherwise to the right
189         }
190
191         BTreeNode z = new BinaryTreeNode(aKey);
192
193         if (y->empty())
194         {
195             return false;   // if y is empty x was empty, insertion will fail (NIL)
196         }
197         else
198         {
199             if (aKey < y->key)  //if given key is less than current key, insert left
200             {
201                 y->left = z;
202             }
203             else
204             {
205                 y->right = z; //otherwise insert right
206             }
207         }
208         return true;
209     }
210
211     BinaryTreeNode() :
212         key(T()),
213         left(&NIL),
214         right(&NIL)
215     {}
216
217     BinaryTreeNode(const T& aKey):
218         key(aKey),
219         left(&NIL),
220         right(&NIL)
221     {}
222
223     BinaryTreeNode(T&& aKey)
224         :
225         key(std::move(aKey)),
226         left(&NIL),
227         right(&NIL)
228     {}
229 };
230
231 template<typename T>
232 BinaryTreeNode<T> BinaryTreeNode<T>::NIL;
233
```

234
235

```
1
2  // COS30008, Problem Set 4, Problem 2, 2022
3
4  #pragma once
5
6  #include "BinaryTreeNode.h"
7
8  #include <stdexcept>
9
10 // Problem 3 requirement
11 template<typename T>
12 class BinarySearchTreeIterator;
13
14 template<typename T>
15 class BinarySearchTree
16 {
17 private:
18
19     using BNode = BinaryTreeNode<T>;
20     using BTreeNode = BNode*;
21
22     BTreeNode fRoot;
23
24 public:
25
26     BinarySearchTree()
27     {
28         fRoot = &BNode::NIL;
29     }
30
31
32     ~BinarySearchTree() {
33         if (!empty())
34         {
35             if (fRoot->left != &BNode::NIL)
36             {
37                 delete fRoot->left;
38             }
39             if (fRoot->right != &BNode::NIL)
40             {
41                 delete fRoot->right;
42             }
43         }
44     }
45
46     bool empty() const
47     {
48         return fRoot->empty();
49     }
```

```
50
51      size_t height() const
52      {
53          if (fRoot == NULL)
54          {
55              return 0;
56          }
57          return fRoot->height();
58      }
59
60      bool insert(const T& aKey)
61      {
62          if (empty()) {
63              fRoot = new BNode(aKey);
64              return true;
65          }
66          return fRoot->insert(aKey);
67      }
68
69      bool remove(const T& aKey) {
70
71        //return fRoot->remove(aKey, &BTreeNode::NIL);
72          //return fRoot->remove(aKey, fRoot->NIL);
73          return false;
74
75      }
76
77      // Problem 3 methods
78
79      using Iterator = BinarySearchTreeIterator<T>;
80
81      // Allow iterator to access private member variables
82      friend class BinarySearchTreeIterator<T>;
83
84      Iterator begin() const
85      {
86          return BinarySearchTree<T>(*this);
87      }
88      Iterator end() const
89      {
90          return begin().end();
91      }
92  };
93
```

```
1
2  // COS30008, Problem Set 4, Problem 3, 2022
3
4  #pragma once
5
6  #include "BinarySearchTree.h"
7
8  #include <stack>
9
10 template<typename T>
11 class BinarySearchTreeIterator
12 {
13 private:
14
15     using BSTree = BinarySearchTree<T>;
16     using BNode = BinaryTreeNode<T>;
17     using BTreeNode = BNode*;
18     using BTNStack = std::stack<BTreeNode>;
19
20     const BSTree& fBSTree;      // binary search tree
21     BTNStack fStack;            // DFS traversal stack
22
23     void pushLeft(BTreeNode aNode) {
24         while (!aNode->empty())
25         {
26             if (!fBSTree.fRoot->empty())
27             {
28                 fStack.push(aNode); // pushes aNode to the root
29                 fBSTree.fRoot = fBSTree.fRoot->left;  // makes root = left node
30             }
31         }
32
33     }
34
35 public:
36
37     using Iterator = BinarySearchTreeIterator<T>;
38
39     BinarySearchTreeIterator(const BSTree& aBSTree)
40         :
41         fBSTree(&BNode::NIL),
42         fStack(&BNode::NIL)
43     {
44
45     }
46
47     const T& operator*() const //get key
48     {
49         if (fStack.empty() == false)
```

```
50          {
51              return fStack.top();
52          }
53      }
54
55      Iterator& operator++()
56      {
57          if ( fStack.empty() == false)
58          {
59              fStack.pop();
60              //fBSTree = fBSTree.fRoot->right;
61              return *this;
62          }
63      }
64
65      Iterator operator++(int)
66      {
67          BinarySearchTreeIterator temp = *this;
68          ++(*this);
69          return temp;
70      }
71
72      bool operator==( const Iterator& aOtherIter ) const
73      {
74          return fStack.size() == aOtherIter.fStack.size();
75      }
76
77      bool operator!=(const Iterator& aOtherIter) const
78      {
79          return !(*this == aOtherIter);
80      }
81
82      Iterator begin() const {
83          return BinarySearchTreeIterator<T>(*this);
84      }
85      Iterator end() const
86      {
87          return begin().end();
88      }
89  };
90
```