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: Trac Duc Anh Luong **Your student id:** 103488117

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  #pragma once
 2  #include <stdexcept>
 3  #include <algorithm>
 4  using namespace std;
 5  template<typename T>
 6  struct BinaryTreeNode
 7  {
 8      using BNode = BinaryTreeNode<T>;
 9      using BTreeNode = BNode*;
10
11      T key;
12      BTreeNode left;
13      BTreeNode right;
14
15      static BNode NIL;
16      const T& findMax() const
17      {
18          if (empty())
19          {
20              throw domain_error("Empty tree encountered");
21          }
22          if (right->empty())
23          {
24              return key;
25          }
26          return right->findMax();
27      }
28      const T& findMin() const
29      {
30          if (empty())
31          {
32              throw domain_error("Empty tree encountered");
33          }
34          if (left->empty())
35          {
36              return key;
37          }
38          return left->findMin();
39      }
40      bool remove(const T& aKey, BTreeNode aParent)
41      {
42          BTreeNode x = this;
43          BTreeNode y = aParent;
44          while (!x->empty())
45          {
46              if (aKey == x->key)
47              {
48                  break;
49              }
```

```
50              y = x;
51              x = aKey < x->key ? x->left : x->right;
52          }
53          if (x->empty())
54          {
55              return false;
56          }
57
58          if (!x->left->empty())
59          {
60              const T& lKey = x->left->findMax();
61              x->key = lKey;
62              x->left->remove(lKey, x);
63          }
64          else
65          {
66              if (!x->right->empty())
67              {
68                  const T& lKey = x->right->findMin();
69                  x->key = lKey;
70                  x->right->remove(lKey, x);
71              }
72              else
73              {
74                  if (y != &NIL)
75                  {
76                      if (y->left == x)
77                      {
78                          y->left = &NIL;
79                      }
80                      else
81                      {
82                          y->right = &NIL;
83                      }
84                  }
85                  delete x;
86              }
87          }
88          return true;
89      }
90
91      BinaryTreeNode(): key(T()), left(&NIL), right(&NIL){}
92      BinaryTreeNode(const T& aKey): key(aKey), left(&NIL), right(&NIL){}
93      BinaryTreeNode(T&& aKey): key(move(aKey)), left(&NIL), right(&NIL){}
94      ~BinaryTreeNode()
95      {
96          if (!left->empty())
97          {
98              delete left;
```

```
 99          }
100          if (!right->empty())
101          {
102              delete right;
103          }
104      }
105
106      bool empty() const
107      {
108          return this == &NIL;
109      }
110      bool leaf() const
111      {
112          return left->empty() && right->empty();
113      }
114      size_t height() const
115      {
116          if (empty())
117          {
118              throw domain_error("Empty tree encountered");
119          }
120          if (leaf())
121          {
122              return 0;
123          }
124          const int left_height = left->empty() ? 1 : left->height() + 1;
125          const int right_height = right->empty() ? 1 : right->height() + 1;
126          return max(left_height, right_height);
127      }
128      bool insert(const T& aKey)
129      {
130          if (empty())
131          {
132              return false;
133          }
134          if (aKey > key)
135          {
136              if (right->empty())
137              {
138                  right = new BNode(aKey);
139              }
140              else
141              {
142                  return right->insert(aKey);
143              }
144              return true;
145          }
146          if (aKey < key)
147          {
```

```
148             if (left->empty())
149             {
150                 left = new BNode(aKey);
151             }
152             else
153             {
154                 return left->insert(aKey);
155             }
156             return true;
157         }
158         return false;
159     }
160 };
161 template<typename T>
162 BinaryTreeNode<T> BinaryTreeNode<T>::NIL;
```

```
1  #pragma once
2  #include "BinaryTreeNode.h"
3  #include <stdexcept>
4  // Problem 3 requirement
5  template<typename T>
6  class BinarySearchTreeIterator;
7  template<typename T>
8  class BinarySearchTree
9  {
10 private:
11     using BNode = BinaryTreeNode<T>;
12     using BTreeNode = BNode*;
13     BTreeNode fRoot;
14
15 public:
16     BinarySearchTree() : fRoot((&BNode::NIL)) {}
17     ~BinarySearchTree()
18     {
19         if (!fRoot->empty())
20         {
21             delete fRoot;
22         }
23     }
24     bool empty() const
25     {
26         return fRoot->empty();
27     }
28     size_t height() const
29     {
30         if (empty())
31         {
32             throw domain_error("Empty tree has no height.");
33         }
34         return fRoot->height();
35     }
36
37     bool insert(const T& aKey)
38     {
39         if (empty())
40         {
41             fRoot = new BNode(aKey);
42             return true;
43         }
44         return fRoot->insert(aKey);
45     }
46     bool remove(const T& aKey)
47     {
48         if (empty())
49         {
```

```
50              throw domain_error("Cannot remove in empty tree.");
51          }
52          if (fRoot->leaf())
53          {
54              if (fRoot->key != aKey)
55              {
56                  return false;
57              }
58              fRoot = &BNode::NIL;
59              return true;
60          }
61          return fRoot->remove(aKey, &BNode::NIL);
62      }
63      // Problem 3 methods
64
65      using Iterator = BinarySearchTreeIterator<T>;
66      // Allow iterator to access private member variables
67      friend class BinarySearchTreeIterator<T>;
68      Iterator begin() const
69      {
70          return Iterator(*this).begin();
71      }
72      Iterator end() const
73      {
74          return Iterator(*this).end();
75      }
76  };
```

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

```
49      }
50      bool operator==(const Iterator& aOtherIter) const
51      {
52          return &fBSTree == &aOtherIter.fBSTree && fStack ==
              aOtherIter.fStack;
53      }
54      bool operator!=(const Iterator& aOtherIter) const
55      {
56          return !(*this == aOtherIter);
57      }
58
59      Iterator begin() const
60      {
61          Iterator temp = *this;
62          temp.fStack = BTNStack();
63          temp.pushLeft(temp.fBSTree.fRoot);
64          return temp;
65      }
66      Iterator end() const
67      {
68          Iterator temp = *this;
69          temp.fStack = BTNStack();
70          return temp;
71      }
72  };
```