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: Nguyen Thi Thanh Minh **Your student id:** 104169617

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	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, 2024
 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>;
13      using BTreeNode = BNode*;
14
15      T key;
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();
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();
39      }
40
41      bool remove( const T& aKey, BTreeNode aParent )
42      {
43          BTreeNode x = this;
44          BTreeNode y = aParent;
45
46          while ( !x->empty() )
47          {
48              if ( aKey == x->key )
49              {
```

```
50                  break;
51              }
52
53              y = x;                                          // new parent
54
55              x = aKey < x->key ? x->left : x->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      // default constructor
101      BinaryTreeNode() :
102          key(T()),
103          left(&NIL),
104          right(&NIL)
105      { }
106
107      // constructor with key
108      BinaryTreeNode(const T& aKey) :
109          key(aKey),
110          left(&NIL),
111          right(&NIL)
112      { }
113
114      // move constructor
115      BinaryTreeNode( T&& aKey ) :
116          key(std::move(aKey)),
117          left(&NIL),
118          right(&NIL)
119      { }
120
121      // destructor
122      ~BinaryTreeNode()
123      {
124          // delete left and right subtrees
125          // if they are not NIL
126          if (!left->empty())
127          {
128              delete left;
129          }
130
131          if (!right->empty())
132          {
133              delete right;
134          }
135      }
136
137      // Is this node NIL (sentinel)?
138      bool empty() const
139      {
140          return this == &NIL;
141      }
142
143      // Is this node a leaf?
144      bool leaf() const
```

```
145     {
146         return left->empty() && right->empty();
147     }
148
149     // Height of the tree
150     size_t height() const
151     {
152         // If first call is on NIL, throw domain error
153         if (empty())
154         {
155             throw std::domain_error("Empty tree encountered!");
156         }
157
158         // Leaf node has height 0
159         if (leaf())
160         {
161             return 0;
162         }
163
164         // calculate height of left and right subtrees
165         // ignoring NIL nodes (not error)
166         size_t lLeftHeight = left->empty() ? 0 : left->height();
167         size_t lRightHeight = right->empty() ? 0 : right->height();
168
169         // return 1 + max subtree height
170         return 1 + std::max(lLeftHeight, lRightHeight);
171     }
172
173     bool insert(const T& aKey)
174     {
175         // If trying to insert a key into NIL
176         // or duplicate key, return false
177         if (empty() || aKey == key)
178         {
179             return false;
180         }
181
182         if (aKey < key) // insert left
183         {
184             if (left->empty())
185             {
186                 // insert new node as left child
187                 left = new BNode(aKey);
188                 return true;
189             }
190             else
191             {
192                 // recursively insert into left subtree
193                 return left->insert(aKey);
```

```
194             }
195         }
196         else // insert right
197         {
198             if (right->empty())
199             {
200                 // insert new node as right child
201                 right = new BNode(aKey);
202                 return true;
203             }
204             else
205             {
206                 // recursively insert into right subtree
207                 return right->insert(aKey);
208             }
209         }
210     }
211 };
212
213 template<typename T>
214 BinaryTreeNode<T> BinaryTreeNode<T>::NIL;
215
```

```
 1
 2  // COS30008, Problem Set 4, Problem 2, 2025
 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      // default constructor
26      BinarySearchTree() :
27          fRoot(&BNode::NIL)
28      { }
29
30      // destructor
31      ~BinarySearchTree()
32      {
33          // avoid deleting NIL
34          if (!empty())
35          {
36              delete fRoot;
37          }
38      }
39
40      bool empty() const
41      {
42          return fRoot->empty();
43      }
44
45      size_t height() const
46      {
47          if (empty())
48          {
49              throw std::domain_error("Empty tree has no height.");
```

```
50          }
51
52          return fRoot->height();
53      }
54
55      bool insert(const T& aKey)
56      {
57          // If tree is empty, create a new root
58          if (empty())
59          {
60              fRoot = new BNode(aKey);
61              return true;
62          }
63
64          // else, insert into the tree
65          return fRoot->insert(aKey);
66      }
67
68      bool remove(const T& aKey)
69      {
70          if (empty())
71          {
72              throw std::domain_error("Cannot remove from an empty tree.");
73          }
74
75          // If fRoot is the only node in the tree,
76          // delete it and set the root to NIL
77          if (aKey == fRoot->key && fRoot->leaf())
78          {
79              delete fRoot;
80              fRoot = &BNode::NIL;
81              return true;
82          }
83
84          return fRoot->remove(aKey, &BNode::NIL);
85      }
86
87      // Problem 3 methods
88
89      using Iterator = BinarySearchTreeIterator<T>;
90
91      // Allow iterator to access private member variables
92      friend class BinarySearchTreeIterator<T>;
93
94      Iterator begin() const
95      {
96          return Iterator(*this);
97      }
98
```

```
 99      Iterator end() const
100      {
101          return Iterator(*this).end();
102      }
103  };
104
```

```
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     // perform a DFS traversal along the left side of the tree
24     void pushLeft(BTreeNode aNode)
25     {
26         while (!aNode->empty())
27         {
28             fStack.push(aNode);
29             aNode = aNode->left;
30         }
31     }
32
33 public:
34
35     using Iterator = BinarySearchTreeIterator<T>;
36
37     // constructor
38     BinarySearchTreeIterator( const BSTree& aBSTree ) :
39         fBSTree(aBSTree),
40         fStack(BTNStack())
41     {
42         pushLeft(fBSTree.fRoot);
43     }
44
45     // dereference operator
46     const T& operator*() const
47     {
48         return fStack.top()->key;
49     }
```

```
50
51      // prefix increment
52      Iterator& operator++()
53      {
54          BTreeNode lNode = fStack.top();
55          fStack.pop();
56          pushLeft(lNode->right);
57          return *this;
58      }
59
60      // postfix increment
61      Iterator operator++(int)
62      {
63          Iterator lTemp = *this;
64          ++(*this);
65          return lTemp;
66      }
67
68      // comparison operators
69      bool operator==(const Iterator& aOtherIter) const
70      {
71          return (&fBSTree == &aOtherIter.fBSTree)
72              && (fStack == aOtherIter.fStack);
73      }
74
75      bool operator!=(const Iterator& aOtherIter) const
76      {
77          return !(*this == aOtherIter);
78      }
79
80      // return an iterator with initialized stack
81      Iterator begin() const
82      {
83          return Iterator(fBSTree);
84      }
85
86      // return an end iterator with empty stack
87      Iterator end() const
88      {
89          Iterator lIter = *this;
90          lIter.fStack = BTNStack();
91          return lIter;
92      }
93  };
94
```