```
 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
```