```
 1
 2  // COS30008, Final Exam
 3
 4  #pragma once
 5
 6  #include <stdexcept>
 7  #include <algorithm>
 8
 9  template<typename T>
10  class TernaryTreePrefixIterator;
11
12  template<typename T>
13  class TernaryTree
14  {
15  public:
16
17      using TTree = TernaryTree<T>;
18      using TSubTree = TTree*;
19
20  private:
21
22      T fKey;
23      TSubTree fSubTrees[3];
24
25      // private default constructor used for declaration of NIL
26      TernaryTree() :
27          fKey(T())
28      {
29          for ( size_t i = 0; i < 3; i++ )
30          {
31              fSubTrees[i] = &NIL;
32          }
33      }
34
35  public:
36
37      using Iterator = TernaryTreePrefixIterator<T>;
38
39      static TTree NIL;           // sentinel
40
41      // getters for subtrees
42      const TTree& getLeft() const { return *fSubTrees[0]; }
43      const TTree& getMiddle() const { return *fSubTrees[1]; }
44      const TTree& getRight() const { return *fSubTrees[2]; }
45
46      // add a subtree
47      void addLeft( const TTree& aTTree ) { addSubTree( 0, aTTree ); }
48      void addMiddle( const TTree& aTTree ) { addSubTree( 1, aTTree ); }
49      void addRight( const TTree& aTTree ) { addSubTree( 2, aTTree ); }
```

```
50
51      // remove a subtree, may through a domain error
52      const TTree& removeLeft() { return removeSubTree( 0 ); }
53      const TTree& removeMiddle() { return removeSubTree( 1 ); }
54      const TTree& removeRight() { return removeSubTree( 2 ); }
55
56  ////////////////////////////////////////////////////////////////////////
57  // Problem 1: TernaryTree Basic Infrastructure
58
59  private:
60
61      // remove a subtree, may throw a domain error [22]
62      const TTree& removeSubTree(size_t aSubtreeIndex)
63      {
64          if (fSubTrees[aSubtreeIndex] == &NIL)
65          {
66              throw std::domain_error("Empty subtree removed!");
67          }
68
69          TTree* lResult = fSubTrees[aSubtreeIndex];
70
71          fSubTrees[aSubtreeIndex] = &NIL;
72
73          return *lResult;
74      }
75
76      // add a subtree; must avoid memory leaks; may throw domain error [18]
77      void addSubTree(size_t aSubtreeIndex, const TTree& aTTree)
78      {
79          if (aSubtreeIndex > 2)
80          {
81              throw std::out_of_range("Invalid subtree index!");
82          }
83
84          if (fSubTrees[aSubtreeIndex] != &NIL)
85          {
86              throw std::domain_error("Subtree is not empty!");
87          }
88
89          fSubTrees[aSubtreeIndex] = new TTree(aTTree);
90      }
91
92  public:
93
94      // TernaryTree l-value constructor [10]
95      TernaryTree(const T& aKey) :
96          TernaryTree()
97      {
98          fKey = aKey;
```

```
 99      }
100
101      // destructor (free sub-trees, must not free empty trees) [14]
102      ~TernaryTree()
103      {
104          for (size_t i = 0; i < 3; i++)
105          {
106              if (fSubTrees[i] != &NIL)
107              {
108                  delete fSubTrees[i];
109              }
110          }
111      }
112
113      // return key value, may throw domain_error if empty [2]
114      const T& operator*() const
115      {
116          if (empty())
117          {
118              throw std::domain_error("Empty tree!");
119          }
120
121          return fKey;
122      }
123
124      // returns true if this ternary tree is empty [4]
125      bool empty() const
126      {
127          return this == &NIL;
128      }
129
130      // returns true if this ternary tree is a leaf [10]
131      bool leaf() const
132      {
133          for (size_t i = 0; i < 3; i++)
134          {
135              if (fSubTrees[i] != &NIL)
136              {
137                  return false;
138              }
139          }
140
141          return true;
142      }
143
144      // return height of ternary tree, may throw domain_error if empty [48]
145      size_t height() const
146      {
147          if (empty())
```

```
148         {
149             throw std::domain_error("Empty tree has no height!");
150         }
151 
152         if (leaf())
153         {
154             return 0;
155         }
156 
157         size_t lMaxHeight = 0;
158         for (size_t i = 0; i < 3; i++)
159         {
160             if (fSubTrees[i] != &NIL)
161             {
162                 lMaxHeight = std::max(lMaxHeight, fSubTrees[i]->height());
163             }
164         }
165 
166         return 1 + lMaxHeight;
167     }
168 
169 ////////////////////////////////////////////////////////////////////////
170 // Problem 2: TernaryTree Copy Semantics
171 
172     // copy constructor, must not copy empty ternary tree
173     TernaryTree(const TTree& aOtherTTree) :
174         TernaryTree()
175     {
176         *this = aOtherTTree;
177     }
178 
179     // copy assignment operator, must not copy empty ternary tree
180     // may throw a domain error on attempts to copy NIL
181     TTree& operator=(const TTree& aOtherTTree)
182     {
183         if (aOtherTTree.empty())
184         {
185             throw std::domain_error("Empty tree cannot be copied!");
186         }
187 
188         if (this != &aOtherTTree)
189         {
190             this->~TernaryTree();
191 
192             fKey = *aOtherTTree;
193             for (size_t i = 0; i < 3; i++)
194             {
195                 fSubTrees[i] = aOtherTTree.fSubTrees[i]->clone();
196             }
```

```
197         }
198
199         return *this;
200     }
201
202
203     // clone ternary tree, must not copy empty trees
204     TSubTree clone() const
205     {
206         if (empty())
207         {
208             return const_cast<TSubTree>(this);
209         }
210
211         return new TTree(*this);
212     }
213
214 ////////////////////////////////////////////////////////////////////////
215 // Problem 3: TernaryTree Move Semantics
216
217     // TTree r-value constructor
218     TernaryTree(T&& aKey) :
219         TernaryTree()
220     {
221         fKey = std::move(aKey);
222     }
223
224     // move constructor, must not copy empty ternary tree
225     TernaryTree( TTree&& aOtherTTree ) :
226         TernaryTree()
227     {
228         *this = std::move(aOtherTTree);
229     }
230
231     // move assignment operator, must not copy empty ternary tree
232     TTree& operator=(TTree&& aOtherTTree)
233     {
234         if (aOtherTTree.empty())
235         {
236             throw std::domain_error("Empty tree cannot be moved!");
237         }
238
239         if (this != &aOtherTTree)
240         {
241             this->~TernaryTree();
242
243             fKey = std::move(*aOtherTTree);
244
245             for (size_t i = 0; i < 3; i++)
```

```
246             {
247                 fSubTrees[i] = aOtherTTree.fSubTrees[i];
248                 aOtherTTree.fSubTrees[i] = &NIL;
249             }
250         }
251 
252         return *this;
253     }
254 
255 ////////////////////////////////////////////////////////////////////////
256 // Problem 4: TernaryTree Prefix Iterator
257 
258     // return ternary tree prefix iterator positioned at start
259     Iterator begin() const
260     {
261         return Iterator(this);
262     }
263 
264     // return ternary prefix iterator positioned at end
265     Iterator end() const
266     {
267         return begin().end();
268     }
269 };
270 
271 template<typename T>
272 TernaryTree<T> TernaryTree<T>::NIL;
273 
```