```cpp
 1
 2  // COS30008, Final Exam, 2022
 3
 4  #pragma once
 5
 6  #include <stdexcept>
 7  #include <algorithm>
 8
 9  using namespace std;
10
11  template<typename T>
12  class TernaryTreePrefixIterator;
13
14  template<typename T>
15  class TernaryTree
16  {
17  public:
18
19      using TTree = TernaryTree<T>;
20      using TSubTree = TTree*;
21
22  private:
23
24      T fKey;
25      TSubTree fSubTrees[3];
26
27      // private default constructor used for declaration of NIL
28      TernaryTree() :
29          fKey(T())
30      {
31          for ( size_t i = 0; i < 3; i++ )
32          {
33              fSubTrees[i] = &NIL;
34          }
35      }
36
37  public:
38
39      using Iterator = TernaryTreePrefixIterator<T>;
40
41      static TTree NIL;           // sentinel
42
43      // getters for subtrees
44      const TTree& getLeft() const { return *fSubTrees[0]; }
45      const TTree& getMiddle() const { return *fSubTrees[1]; }
46      const TTree& getRight() const { return *fSubTrees[2]; }
47
48      // add a subtree
49      void addLeft( const TTree& aTTree ) { addSubTree( 0, aTTree ); }
```

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

```cpp
 98
 99     TernaryTree(const T& aKey) :fKey(aKey)
100     {
101         for (int i = 0; i < 3; i++)
102         {
103             fSubTrees[i] = &NIL;
104         }
105     }
106
107     // destructor (free sub-trees, must not free empty trees) [14]
108     ~TernaryTree()
109     {
110         if (!empty())
111         {
112             for (int i = 0; i < 3; i++)
113             {
114                 if (!fSubTrees[i]->empty())
115                 {
116                     delete fSubTrees[i];
117                 }
118             }
119         }
120     }
121
122     // return key value, may throw domain_error if empty [2]
123     const T& operator*() const
124     {
125         if (empty())
126         {
127             throw domain_error("Tree is empty");
128         }
129         return fKey;
130     }
131
132     // returns true if this ternary tree is empty [4]
133     bool empty() const { return this == &NIL; }
134
135     // returns true if this ternary tree is a leaf [10]
136     bool leaf() const
137     {
138         for (int i = 0; i < 3; i++)
139         {
140             if (!fSubTrees[i]->empty()) return false;
141         }
142         return true;
143     }
144
145     // return height of ternary tree, may throw domain_error if empty [48]
146     size_t height() const
```

```
147          {
148              if (empty())
149              {
150                  throw domain_error("Operation not supported");
151              }
152              if (leaf()) return 0;
153              size_t height[3] = {};
154
155              for (int i = 0; i < 3; i++)
156              {
157                  height[i] = fSubTrees[i]->empty() ? 0 : fSubTrees[i]->height ↵
                        ();
158              }
159              return *max_element(height, height + 3) + 1;
160          }
161
162  ////////////////////////////////////////////////////////////////////////
163  // Problem 2: TernaryTree Copy Semantics
164
165      // copy constructor, must not copy empty ternary tree
166      TernaryTree( const TTree& aOtherTTree )
167      {
168          for (int i = 0; i < 3; i++)
169          {
170              fSubTrees[i] = &NIL;
171          }
172          *this = aOtherTTree;
173      }
174
175      // copy assignment operator, must not copy empty ternary tree
176      // may throw a domain error on attempts to copy NIL
177      TTree& operator=(const TTree& aOtherTTree)
178      {
179          if (this != &aOtherTTree)
180              if (!aOtherTTree.empty())
181              {
182                  this->~TernaryTree();
183                  fKey = aOtherTTree.fKey;
184                  for (size_t i = 0; i < 3; i++)
185                  {
186                      if (!aOtherTTree.fSubTrees[i]->empty())
187                      {
188                          fSubTrees[i] = aOtherTTree.fSubTrees[i]->clone();
189                      }
190                      else
191                      {
192                          fSubTrees[i] = &NIL;
193                      }
194                  }
```

```
195                    }
196                    else
197                    {
198                        throw domain_error("NIL as source not permitted.");
199                    }
200            return *this;
201        }
202
203        // clone ternary tree, must not copy empty trees
204        TSubTree clone() const
205        {
206            if (empty())
207            {
208                throw domain_error("NIL as source not permitted.");
209            }
210            return new TTree(*this);
211        }
212
213 ////////////////////////////////////////////////////////////////////////////
214 // Problem 3: TernaryTree Move Semantics
215
216        // TTree r-value constructor
217        TernaryTree( T&& aKey ): fKey(std::move(aKey))
218        {
219            for (int i = 0; i < 3; i++)
220            {
221                fSubTrees[i] = &NIL;
222            }
223        }
224
225        // move constructor, must not copy empty ternary tree
226        TernaryTree( TTree&& aOtherTTree )
227        {
228            for (int i = 0; i < 3; i++)
229            {
230                fSubTrees[i] = &NIL;
231            }
232            *this = move(aOtherTTree);
233        }
234
235        // move assignment operator, must not copy empty ternary tree
236        TTree& operator=( TTree&& aOtherTTree )
237        {
238            if (this != &aOtherTTree)
239            {
240                if (!aOtherTTree.empty())
241                {
242                    this->~TernaryTree();
243                    fKey = std::move(aOtherTTree.fKey);
```

```cpp
244                    for (int i = 0; i < 3; i++)
245                    {
246                        if (!aOtherTTree.fSubTrees[i]->empty()) fSubTrees[i] = ⮐
                            const_cast<TSubTree>(&aOtherTTree.removeSubTree(i));
247                        else fSubTrees[i] = &NIL;
248                    }
249                }
250                else
251                {
252                    throw std::domain_error("NIL as source not permitted.");
253                }
254            }
255        }
256
257    /////////////////////////////////////////////////////////////////////////
258    // Problem 4: TernaryTree Prefix Iterator
259
260        // return ternary tree prefix iterator positioned at start
261        Iterator begin() const
262        {
263            return Iterator(this).begin();
264        }
265
266        // return ternary prefix iterator positioned at end
267        Iterator end() const
268        {
269            return Iterator(this).end();
270        }
271    };
272
273    template<typename T>
274    TernaryTree<T> TernaryTree<T>::NIL;
275
```