```
1: #ifndef BINARY_TREE_H
2: #define BINARY_TREE_H
3:
4: #include <vector>
5: #include "VariousExceptions.h"
6:
7: template <typename Object>
8: class BinaryTree {
9:
10: public:
11:   /* Default Constructor.
12:    * Creates a tree with single node (which contains default element)
13:    */
14:   BinaryTree()  : sz(1), rt(new Node) { }
15:
16:   /* Copy Constructor */
17:   BinaryTree(const BinaryTree& orig)
18:     : sz(orig.sz), rt(cloneRecurse(orig,orig.rt)) { }
19:
20:   /* Overloaded Assignment Operator */
21:   BinaryTree& operator=(const BinaryTree& orig) {
22:     if (this != &orig) {
23:       clearRecurse(rt);
24:       rt = cloneRecurse(orig,orig.rt);
25:     }
26:     return *this;
27:   }
28:
29:   /* Destructor */
30:   ~BinaryTree() {
31:     clearRecurse(rt);
32:   }
33:
34: private:
35:   /*****************************************************
36:    * Class Node:  this represents a single node of the tree
37:    */
38:   struct Node {                                        // node in the BinaryTree
39:     Object  element;                                   // element
40:     Node*   parent;                                    // parent node
41:     Node*   left;                                      // left child
42:     Node*   right;                                     // left child
43:     Node(const Object& e = Object(), Node* p = NULL, Node* l = NULL, Node* r = NULL)
44:       : element(e), parent(p), left(l), right(r) { }   // constructor
45:   };
46:   typedef Node* NodePtr;                               // pointer to a Node
47:
48: public:
49:   /*****************************************************
50:    * Class Position:  represents a position in a BinaryTree
51:    */
52:   class Position {
53:   public:
54:     /* Constructor */
55:     Position(NodePtr n = NULL) : node(n) { }
56:
57:     /* Returns the element at the Position */
58:     Object& element() const throw(InvalidPositionException) {
59:       if (node == NULL) throw InvalidPositionException("Null position");
60:       return node->element;
61:     }
62:
63:     /* Determines whether the Position is a 'null' position */
64:     bool isNull() const { return node == NULL; }
65:
66:     /* Overload the equality operator */
67:     bool operator==(const Position& other) {
68:       return (this->node == other.node);
69:     }
70:
71:     /* Overload the equality operator */
72:     bool operator!=(const Position& other) {
73:       return ! operator==(other);
74:     }
75:
76:   private:
77:     friend class BinaryTree<Object>;  // allow access to private member
78:     NodePtr node;
79:     NodePtr validate(const BinaryTree<Object>* tree) const throw(InvalidPositionException) {
80:       if (node == NULL)
81:         throw InvalidPositionException("Cannot use a NULL position");
82:       if (node!=tree->rt && node->parent==NULL)
83:         throw InvalidPositionException("Position appears to involve removed element.");
84:       else return node;
85:     }
86:   };
87:   /*********** end of class Position **************************************/
```

```
 88:
 89:   /*****************************************************************
 90:    * Iterator Classes
 91:    */
 92:   template <typename T>
 93:   class Iterator {
 94:   public:
 95:     /* Are there more items left in iteration? */
 96:     bool hasNext() {
 97:       return (index < items.size());
 98:     }
 99:
100:     /* Returns the next available item in the iteration */
101:     T next() {
102:       return items[index++];
103:     }
104:
105:   private:
106:     friend class BinaryTree;
107:     Iterator() : index(0) { }
108:     std::vector<T> items;  // vector of items
109:     unsigned int index;    // current index
110:   };
111:   typedef Iterator<Position> PositionalIterator;
112:   typedef Iterator<Object> ObjectIterator;
113:   /*************** end of iterator classes ***************************/
114:
115:   /***************************** query methods ***********************/
116:
117:   /* Returns size of tree */
118:   int size() const {
119:     return sz;
120:   }
121:
122:   /* does position correspond to internal node? */
123:   bool isInternal(const Position& p) const
124:     throw(InvalidPositionException)  {
125:     NodePtr v = p.validate(this);
126:     return v->left != NULL;
127:   }
128:
129:   /* does position correspond to external node? */
130:   bool isExternal(const Position& p) const
131:     throw(InvalidPositionException)  {
132:     NodePtr v = p.validate(this);
133:     return v->left == NULL;
134:   }
135:
136:   /* is this the root position? */
137:   bool isRoot(const Position& p) const
138:     throw(InvalidPositionException)  {
139:     NodePtr v = p.validate(this);
140:     return v == rt;
141:   }
142:
143:   /***************************** accessor methods ********************/
144:
145:   /* return position of root of tree */
146:   Position root() const  {
147:     return Position(rt);
148:   }
149:
150:   /* return position of parent of given position */
151:   Position parent(const Position& p) const
152:     throw(BoundaryViolationException, InvalidPositionException) {
153:     NodePtr v = p.validate(this);
154:     if (v==rt) throw BoundaryViolationException("Cannot traverse parent of root");
155:     return Position(v->parent);
156:   }
157:
158:   /* return position of left child of given position */
159:   Position leftChild(const Position& p) const
160:     throw(BoundaryViolationException, InvalidPositionException) {
161:     NodePtr v = p.validate(this);
162:     if (v->left==NULL) throw BoundaryViolationException("Cannot traverse child of external position");
163:     return Position(v->left);
164:   }
165:
166:   /* return position of right child of given position */
167:   Position rightChild(const Position& p) const
168:     throw(BoundaryViolationException, InvalidPositionException) {
169:     NodePtr v = p.validate(this);
170:     if (v->right==NULL) throw BoundaryViolationException("Cannot traverse child of external position");
171:     return Position(v->right);
172:   }
```

```
173:
174:   /* return position of sibling of given position */
175:   Position sibling(const Position& p) const
176:     throw(BoundaryViolationException, InvalidPositionException) {
177:     NodePtr v = p.validate(this);
178:     if (v==rt) throw BoundaryViolationException("Cannot traverse sibling of root");
179:     NodePtr parent = v->parent;
180:     NodePtr lc = parent->left;
181:     NodePtr s = (v==lc ? parent->right : lc);
182:     return Position(s);
183:   }
184:
185:   /* return iterator of all children of given position */
186:   PositionalIterator children(const Position& p) const
187:     throw(InvalidPositionException) {
188:     NodePtr v = p.validate(this);
189:     PositionalIterator PI;
190:
191:     if (v->left!=NULL) {
192:       PI.items.push_back(Position(v->left));
193:       PI.items.push_back(Position(v->right));
194:     }
195:
196:     return PI;
197:   }
198:
199:   /* return iterator of all positions of the tree */
200:   PositionalIterator positions() const {
201:     PositionalIterator PI;
202:     recurseAdd(PI,rt);
203:     return PI;
204:   }
205:
206:   /* return iterator of all elements stored in the tree */
207:   ObjectIterator elements() const {
208:     ObjectIterator OI;
209:     recurseAdd(OI,rt);
210:     return OI;
211:   }
212:
213:   /***************************** update methods ***************************/
214:
215:   /* replace the element at the given position */
216:   void replaceElement(const Position& p, const Object& element)
217:     throw(InvalidPositionException) {
218:     NodePtr v  = p.validate(this);
219:     v->element = element;
220:   }
221:
222:   /* swap the elements stored at the given positions */
223:   void swapElements(const Position& p, const Position& q)
224:     throw(InvalidPositionException) {
225:     NodePtr v  = p.validate(this);
226:     NodePtr w  = q.validate(this);
227:     Object e = v->element;
228:     v->element = w->element;
229:     w->element = e;
230:   }
231:
232:   /* Converts external position into an internal node with two newly
233:    * created external children (each of which have default element)
234:    */
235:   void expandExternal(const Position& p)
236:     throw(InvalidPositionException,BoundaryViolationException) {
237:     NodePtr v  = p.validate(this);
238:     if (v->left==NULL) {
239:       v->left = new Node(Object(),v,NULL,NULL);
240:       v->right = new Node(Object(),v,NULL,NULL);
241:       sz+=2;
242:     } else {
243:       throw BoundaryViolationException("Cannot expand internal node");
244:     }
245:   }
```

```
246:   /* Replaces the external position p with a subtree which mirrors the
247:    * contents of a second tree T2.  Existing positions of the second
248:    * tree remain valid in the result.
249:    *
250:    *  Note well: the external node as well as the second tree itself are
251:    *  destroyed as a side effect.
252:    */
253:   void replaceExternalWithSubtree(const Position& p, BinaryTree& T2)
254:     throw(InvalidPositionException,BoundaryViolationException) {
255:     NodePtr v  = p.validate(this);
256:     if (v->left==NULL) {
257:       sz+=T2.sz-1;
258:       if (v==rt) {
259:         rt = T2.rt;
260:       } else {
261:         NodePtr parent = v->parent;
262:         T2.rt->parent = parent;
263:         if (v==parent->left)
264:           parent->left = T2.rt;
265:         else
266:           parent->right = T2.rt;
267:       }
268:
269:       // deallocate original external node
270:       delete v;
271:
272:       // convert T2 back to a default tree (in a way so that its
273:       // original nodes are not destroyed, as they are now part of
274:       // this tree)
275:       T2.sz = 1;
276:       T2.rt = new Node;
277:     } else {
278:       throw BoundaryViolationException("Cannot replace internal node");
279:     }
280:   }
281:
282:   /* Takes an external position w of tree, deletes w and the parent of
283:    * w from the tree, promoting the sibling of w into the parent's
284:    * place (see Figure 6.13)
285:    */
286:   Position removeAboveExternal(const Position& w)
287:     throw(InvalidPositionException,BoundaryViolationException) {
288:     NodePtr v  = w.validate(this);
289:     if (v==rt)
290:       throw BoundaryViolationException("Cannot use replaceAboveExternal on root");
291:
292:     if (v->left!=NULL)
293:       throw BoundaryViolationException("Cannot use replaceAboveExternal on internal node");
294:
295:     NodePtr parent = v->parent;
296:     NodePtr s = sibling(w).node;
297:     if (isRoot(parent))
298:       rt = s;
299:     else {
300:       NodePtr grand = parent->parent;
301:       if (parent==grand->left)
302:         grand->left = s;
303:       else
304:         grand->right = s;
305:       s->parent = grand;
306:     }
307:     sz-=2;
308:     delete parent;
309:     delete v;
310:     return Position(s);
311:   }
312:
313: private:
314:   int       sz;       // number of items
315:   NodePtr   rt;       // root of the tree
316:
317:   /* Utilities used for tree iterators */
318:   void recurseAdd(PositionalIterator &pi, const NodePtr v) const {
319:     if (v->left!=NULL) recurseAdd(pi,v->left);
320:     pi.items.push_back(Position(v));
321:     if (v->right!=NULL) recurseAdd(pi,v->right);
322:   }
323:
324:   void recurseAdd(ObjectIterator &oi, const NodePtr v) const {
325:     if (v->left!=NULL) recurseAdd(oi,v->left);
326:     oi.items.push_back(v->element);
327:     if (v->right!=NULL) recurseAdd(oi,v->right);
328:   }
```

```
329:   /* Utilities used for copy constructor, assignment operator, and destructor */
330:   void clearRecurse(const NodePtr v) {
331:     if (v!=NULL) {
332:       if (v->left!=NULL) {
333:         clearRecurse(v->left);
334:         clearRecurse(v->right);
335:       }
336:       delete v;
337:     }
338:   }
339:
340:   NodePtr cloneRecurse(const BinaryTree& orig, const NodePtr v) {
341:     NodePtr n = new Node(v->element);
342:     if (v->left!=NULL) {
343:       n->left = cloneRecurse(orig,v->left);
344:       n->left->parent = n;
345:       n->right = cloneRecurse(orig,v->right);
346:       n->right->parent = n;
347:     }
348:     return n;
349:   }
350: };
351:
352: #endif
```