```
 1: #ifndef BINARY_TREE_H
 2: #define BINARY_TREE_H
 3:
 4: /** Class for a binary tree. Updated: 17 November 2008 */
 5: #include <cstddef>
 6: #include <stdexcept>
 7:
 8: template<typename Item_Type>
 9: class Binary_Tree
10: {
11:
12: protected:
13:   //---------------  inner class for node  ------------------
14:   class BTNode
15:   {
16:   private:
17:     // Data Fields
18:     BTNode  *_parent, *_left, *_right;
19:     Item_Type _data;
20:
21:   public:
22:
23:     // Constructor
24:     BTNode() :  _parent(NULL), _left(NULL), _right(NULL) { }
25:
26:     // Destructor
27:     virtual ~BTNode() { }
28:
29:     Item_Type& data() { return _data; }
30:
31:     bool isEnd() const { return this == this->_left; }
32:
33:     bool hasParent() const { return _parent != NULL; }
34:
35:     BTNode* parent() const { return _parent; }
36:
37:     bool hasLeft() const { return !_left->isEnd(); }
38:
39:     BTNode* left() const { return _left; }
40:
41:     bool hasRight() const { return !_right->isEnd(); }
42:
43:     BTNode* right() const { return _right; }
44:
45:     bool isLeaf() const { return !(hasLeft() || hasRight()); }
46:
47:     BTNode* sibling() {           // should not be called upon root
48:       if (this == parent->_left)
49:         return parent->_right;
50:       else
51:         return parent->_left;
52:     }
53:
54:   private:
55:     friend class Binary_Tree<Item_Type>;  // give access to outer class
56:
57:     void linkLeft(BTNode* other) {
58:       _left = other;
59:       _left->_parent = this;
60:     }
61:
62:     void linkRight(BTNode* other) {
63:       _right = other;
64:       _right->_parent = this;
65:     }
66:
67:   }; // End BTNode
68:
69: private:
70:   // data members
71:   BTNode* sentinel;    // Fixed node used to desginate non-existent children
```

```
 72:   BTNode* root;       // A reference to the root of the tree
 73:   size_t num_items;    // The number of nodes in the tree
 74:
 75: public:
 76:   /** Construct a tree with zero nodes */
 77:   Binary_Tree() : sentinel(createSentinel()), root(sentinel), num_items(0) { }
 78:
 79:   /** Return size of the tree. */
 80:   size_t size() const { return num_items; }
 81:
 82:   /** Return true if tree is empty. */
 83:   bool empty() const { return num_items == 0; }
 84:
 85: protected:
 86:   //---------------  utilities for housekeeping  ------------------
 87:   /** Factory function for instantiating a new node instance. */
 88:   virtual BTNode* createNode() {
 89:     return new BTNode();
 90:   }
 91:
 92: private:
 93:   BTNode* createSentinel() {
 94:     BTNode* temp = createNode();
 95:     temp->linkLeft(temp);
 96:     temp->linkRight(temp);
 97:     return temp;
 98:   }
 99:
100: protected:
101:   BTNode* createLeaf(const Item_Type& the_data = Item_Type()) {
102:     BTNode* temp = createNode();
103:     temp->data() = the_data;
104:     temp->linkLeft(sentinel);
105:     temp->linkRight(sentinel);
106:     return temp;
107:   }
108:
109:   /** Clone this subtree, returning pointer to new root.
110:    *  Note that the parent of the new root is NULL.
111:    */
112:   BTNode* cloneSubtree(BTNode* node) {
113:     if (node->isEnd())
114:       return sentinel;   // do not clone it
115:     else {
116:       BTNode* new_root = createLeaf(node->data());
117:       new_root->linkLeft(cloneSubtree(node->left()));
118:       new_root->linkRight(cloneSubtree(node->right()));
119:       return new_root;
120:     }
121:   }
122:
123:   /** Destroy this node and all descendents.
124:    *  However it leaves the sentinel alone.
125:    *  The num_items is accurately decremented to reflect the changes.
126:    */
127:   void destroySubtree(BTNode* node) {
128:     if (!node->isEnd()) {
129:       destroySubtree(node->left());
130:       destroySubtree(node->right());
131:       delete node;
132:       num_items--;
133:     }
134:   }
135:
136: public:
137:   /** Construct a tree with given item at root and copies of indicated subtrees.
138:    *  The original subtrees are not affected.
139:    */
140:   Binary_Tree(const Item_Type& item,
141:               const Binary_Tree<Item_Type>& left_subtree = Binary_Tree(),
142:               const Binary_Tree<Item_Type>& right_subtree = Binary_Tree()) :
```

```
143:     sentinel(createSentinel()), root(createLeaf(item)),
144:     num_items(1 + left_subtree.size() + right_subtree.size()) {
145:     root->linkLeft(cloneSubtree(left_subtree.root));
146:     root->linkRight(cloneSubtree(right_subtree.root));
147:   }
148:
149:   //---------------  housekeeping functions  ------------------
150:   Binary_Tree(const Binary_Tree& other) : sentinel(createSentinel()),
151:     root(cloneSubtree(other.root)), num_items(other.num_items) { }
152:
153:   Binary_Tree& operator=(const Binary_Tree& other) {
154:     if (this != &other) {
155:       destroySubtree(root);
156:       root = cloneSubtree(other.root);
157:       num_items = other.num_items;
158:     }
159:     return *this;
160:   }
161:
162:   virtual ~Binary_Tree() {
163:     destroySubtree(root);
164:     delete sentinel;
165:   }
166:
167: protected:
168:   //---------------  some static utility methods  ------------------
169:
170:   /** Find leftmost node within a subtree.
171:    *  Node is its own leftmost descendent when left subtree empty.
172:    */
173:   static BTNode* findLeftmostDescendent(BTNode* node) {
174:     while (node->hasLeft())
175:       node = node->left();
176:     return node;
177:   }
178:
179:   /** Find rightmost node within a subtree.
180:    *  Node is its own rightmost descendent when right subtree empty.
181:    */
182:   static BTNode* findRightmostDescendent(BTNode* node) {
183:     while (node->hasRight())
184:       node = node->right();
185:     return node;
186:   }
187:
188:   /**
189:    * Returns a pointer to the preceding node as per an in-order
190:    * traversal.  Will return NULL in the case that the given node is
191:    * the first and therefore has no predecessor.
192:    */
193:   static BTNode* findPredecessor(BTNode* node) {
194:     if (node->hasLeft())
195:       return findRightmostDescendent(node->left());
196:     else {
197:       while (node->hasParent() && node == node->parent()->left())
198:         node = node->parent();
199:       return node->parent();    // will be NULL when there is no predecessor
200:     }
201:   }
202:
203:   /**
204:    * Returns a pointer to the successor node as per an in-order
205:    * traversal.  Will return NULL in the case that the given node is
206:    * the last and therefore has no successor.
207:    */
208:   static BTNode* findSuccessor(BTNode* node) {
209:     if (node->hasRight())
210:       return findLeftmostDescendent(node->right());
211:     else {
212:       while (node->hasParent() && node == node->parent()->right())
213:         node = node->parent();
```

```
214:       return node->parent(); // will be NULL when there is no successor
215:     }
216:   }
217:
218: public:
219:   //---------------  iterator class  ------------------
220:   class iterator {
221:     friend class Binary_Tree<Item_Type>;  // Give access to outer class.
222:
223:   protected:
224:     const Binary_Tree<Item_Type>* tree;
225:     typename Binary_Tree<Item_Type>::BTNode* current;
226:
227:     iterator(const Binary_Tree<Item_Type>* tree, BTNode* pos)
228:       : tree(tree), current(pos) { }
229:
230:   public:
231:     /** Default constructor makes an invalid iterator. */
232:     iterator() : tree(NULL), current(NULL) { }
233:
234:     const Item_Type& operator*() const {
235:       return current->data();
236:     }
237:
238:     const Item_Type* operator->() const {
239:       return &current->data();
240:     }
241:
242:     bool operator==(const iterator& other) const {
243:       return (tree == other.tree && current == other.current);
244:     }
245:
246:     bool operator!=(const iterator& other) const {
247:       return !(*this == other);
248:     }
249:
250:     bool isRoot() const { return current == tree->root; }
251:
252:     bool hasLeft() const { return current->hasLeft(); }
253:
254:     bool hasRight() const { return current->hasRight(); }
255:
256:     bool hasParent() const { return current->hasParent(); }
257:
258:     bool isLeaf() const { return current->isLeaf(); }
259:
260:     iterator parent() const {
261:       iterator result(tree, current->parent());
262:       if (!result.current)
263:         result.current = tree->sentinel;
264:       return result;
265:     }
266:
267:     iterator left() const {
268:       return iterator(tree, current->left());
269:     }
270:
271:     iterator right() const {
272:       return iterator(tree, current->right());
273:     }
274:
275:     iterator& operator++() {    // pre-increment version
276:       current = findSuccessor(current);
277:       if (!current)
278:         current = tree->sentinel;
279:       return *this;
280:     }
281:
282:     iterator operator++(int) {    // post-increment version
283:       iterator fixed(*this);
284:       ++(*this);
```

```
285:       return fixed;
286:     }
287:
288:     iterator& operator--() {    // pre-increment version
289:       if (current->isEnd())
290:         current = findRightmostDescendent(tree->getRoot().current);
291:       else
292:         current = findPredecessor(current);
293:       return *this;
294:     }
295:
296:     iterator operator--(int) {    // post-increment version
297:       iterator fixed(*this);
298:       --(*this);
299:       return fixed;
300:     }
301:   };  // end of iterator class
302:
303:
304:   //---------------  tree methods involving iterators  ------------------
305:   iterator getRoot() const {
306:     return iterator(this, root);
307:   }
308:
309:   iterator begin() const {
310:     return iterator(this, findLeftmostDescendent(root));
311:   }
312:
313:   iterator end() const {
314:     return iterator(this, sentinel);
315:   }
316:
317: protected:
318:   //---------------  iterator-based mutators  ------------------
319:   void resetData(const iterator& pos, const Item_Type& item) {
320:     pos.current->data() = item;
321:   }
322:
323:   /**
324:    *  Replace the subtree rooted at this position by the given structure.
325:    *  By default, replaces the subtree with an empty tree.
326:    *  The given parameter is unaffected.
327:    *  Returns an iterator to the root of the newly created subtree.
328:    */
329:   iterator replaceThisSubtree(const iterator& pos,
330:                               const Binary_Tree<Item_Type>& other=Binary_Tree()) {
331:     BTNode* node = pos.current;
332:     BTNode* clone = cloneSubtree(other.root);
333:     BTNode* parent = node->parent();
334:     bool leftOrient = (parent && node == parent->left());
335:
336:     num_items += other.size();
337:     destroySubtree(node);
338:     if (parent) {
339:       if (leftOrient)
340:         parent->linkLeft(clone);
341:       else
342:         parent->linkRight(clone);
343:     } else {
344:       root = clone;
345:     }
346:
347:     if (clone)
348:       return iterator(this, clone);
349:     else
350:       return this->end();
351:   }
352:
353:   /**
354:    *  Replace the left subtree of this position by the given subtree.
355:    *  By default, replaces the subtree with empty subtree.
```

```
356:    *  The given parameter is unaffected.
357:    */
358:   iterator replaceLeftSubtree(const iterator& pos,
359:                               const Binary_Tree<Item_Type>& other=Binary_Tree()) {
360:     BTNode* node = pos.current;
361:     BTNode* clone = cloneSubtree(other.root);
362:     num_items += other.size();
363:     destroySubtree(node->left());
364:     node->linkLeft(clone);
365:     return iterator(this, node->left());
366:   }
367:
368:   /**
369:    *  Replace the right subtree of this position by the given subtree.
370:    *  By default, replaces the subtree with empty subtree.
371:    *  The given parameter is unaffected.
372:    */
373:   iterator replaceRightSubtree(const iterator& pos,
374:                               const Binary_Tree<Item_Type>& other=Binary_Tree()) {
375:     BTNode* node = pos.current;
376:     BTNode* clone = cloneSubtree(other.root);
377:     num_items += other.size();
378:     destroySubtree(node->right());
379:     node->linkRight(clone);
380:     return iterator(this, node->right());
381:   }
382:
383:   /**
384:    *  Assuming that given position has zero or one children,
385:    *  Removes the node from the tree, and reconnects parent
386:    *  to remaining child (if any).
387:    */
388:   void spliceOut(const iterator& pos) {
389:     BTNode* node = pos.current;
390:     if (node->isEnd())
391:       throw std::invalid_argument("Invalid position in the tree");
392:     if (node->hasLeft() && node->hasRight())
393:       throw std::invalid_argument("Cannot splice out node with two children");
394:
395:     BTNode* parent = node->parent();
396:     BTNode* child = (node->hasLeft() ? node->left() : node->right());
397:     if (parent) {
398:       if (node == parent->left())
399:         parent->linkLeft(child);
400:       else
401:         parent->linkRight(child);
402:     } else {
403:       root = child;
404:       root->_parent = NULL;
405:     }
406:
407:     delete node;
408:     num_items--;
409:   }
410:
411:   /**
412:    *  Performs a "rotation" along the edge between this node and its parent.
413:    */
414:   void rotateUpward(const iterator& pos) {
415:     BTNode* node = pos.current;
416:     BTNode* p = node->parent();
417:     if (p) {
418:       BTNode* grand = p->parent();  // mark grandparent while we have it
419:
420:       if (node == p->left()) {
421:         p->linkLeft(node->right());
422:         node->linkRight(p);
423:       } else {
424:         p->linkRight(node->left());
425:         node->linkLeft(p);
426:       }
```

```
427:       if (grand) {
428:         if (grand->left() == p)
429:           grand->linkLeft(node);
430:         else
431:           grand->linkRight(node);
432:       } else {
433:         root = node;
434:         root->_parent = NULL;
435:       }
436:     }
437:   }
438:
439: };  // End Binary_Tree
440: #endif
```