```
01 /**
02  * 二叉树的模板实现
03  */
04
05 #ifndef PROJECT_BITTREE_H
06 #define PROJECT_BITTREE_H
07
08 #include <utility>
09 #include <iostream>
10 #include <vector>
11 #include <stack>
12 #include <queue>
13 #include <functional>
14
15 template<typename Comparable>
16 class BitTree {
17     struct BitNode;
18     using BitNodePtr = BitNode *;
19
20 #define EASY_VISIT_LAMBDA [](const BitNode *node) {std::cout ⤸
21 << node->data << " " <<std::flush; }
22
23 public:
24     BitTree() : root(nullptr), header(nullptr) {};
25     BitTree(const BitTree &rhs) : header(nullptr) { root = ⤸
26     clone(rhs.root); }
27     BitTree(BitTree &&rhs) noexcept: root(rhs.root), header(⤸
28     nullptr) { rhs.root = nullptr; }
29     ~BitTree() {
30         makeEmpty();
31     };
32
33     /** 判断树是否为空 */
34     bool isEmpty() const {
35         return root == nullptr;
36     }
37
```

```
38      /** 将树清空 */
39      void makeEmpty() {
40          makeEmpty(root);
41          if (header != nullptr)
42              delete header;
43      }
44
45      /** 以 dot 格式打印图的内容 */
46      void printGraph(std::ostream &out = std::cout) const {
47          std::queue<BitNode *> nodeQueue;
48          BitNode *node = root;
49          out << "digraph tree{" << std::endl;
50          if (header != nullptr) {
51              if (header->left != nullptr)
52                  out << R"(  "header" -> ")" << header->left->
53                  data
54                      << R"("[label = "left"];)" << std::endl;
55              if (header->right != nullptr)
56                  out << R"(  "header" -> ")" << header->right-
57                  >data
58                      << R"("[label = "right"];)" << std::endl;
59          }
60          visitTreeCover([&](BitNode *bitNode) {
61              if (bitNode->left != nullptr && bitNode->left !=
62              header)
63                  out << "  \"" << bitNode->data << "\" -> \""
64                  << bitNode->left->data << R"("[label =
65                  "left"];)"
66                      << std::endl;
67              else if (bitNode->left != nullptr)
68                  out << "  \"" << bitNode->data << R"(" ->
69                  "header"[label = "left"];)" << std::endl;
70
71              if (bitNode->right != nullptr && bitNode->right
72              != header)
73                  out << "  \"" << bitNode->data << "\" -> \""
74                  << bitNode->right->data << R"("[label =
```

```
75                  "right"];)"
76                      << std::endl;
77              else if (bitNode->right != nullptr)
78                  out << "  \"" << bitNode->data << R"(" -> )
79                  "header"[label = "right"];)" << std::endl;
80          });
81          out << "}" << std::endl;
82      }
83
84      /**
85       * L:left
86       * D:data
87       * R:right
88       */
89      enum Mode { DLR, LDR, LRD };
90
91      /** 迭代器 */
92      class Iterator {
93      public:
94          Iterator(BitNode *Root, Mode theMode) : header(Root),
95          mode(theMode), isEnd(header == nullptr) {
96              switch (mode) {
97              default:
98              case DLR:curr = header->left;
99                  break;
100             case LDR:
101             case LRD:curr = header->left;
102                 while (curr->lTag && curr != nullptr)
103                     curr = curr->left;
104             }
105             isEnd = curr == header;
106         }
107         const BitNode *operator*() {
108             return curr;
109         }
110         void operator++(int) {
111             switch (mode) {
```

```
112             default:
113             case DLR:preNext();
114                 isEnd = (curr == header);
115                 break;
116             case LDR:inNext();
117                 isEnd = (curr == header);
118                 break;
119             case LRD:postNext();
120                 isEnd = (curr == header);
121                 break;
122             }
123         }
124         const bool &IsEnd() { return isEnd; }
125     private:
126
127         const BitNodePtr header;
128         Mode mode;
129         const BitNode *curr;
130         bool isEnd;
131
132         /**
133          * 前序遍历的后继结点:
134          * (1)
135          P 的左子树不为空，此时 P 的后继结点就
136          P 的左儿子;
137          * (2)
138          P 的左子树为空但右子树不为空，此时 P
139          后继结点就是 P 的右儿子;
140          * (3)
141          P 的左右子树均为空，此时在从 P 开始的
142          线索序列中，第一个有右儿子的节点的
143          右儿子或者头结点就是 P 的后继结点。
144          */
145         void preNext() {
146             if (curr->lTag)
147                 curr = curr->left;
148             else {
```

```
149                 while (!curr->rTag) {
150                     curr = curr->right;
151                     if (curr == header)
152                         return;
153                 }
154                 curr = curr->right;
155             }
156         }
157         /**
158          * 中序遍历的后继结点:
159          * (1)
160          若一个节点的右子树为空，此时右线索
161          指节点即为所求;
162          * (2)
163          若这个节点的右子树不为空，此时它的
164          继结点是其右子树的最左节点。
165          */
166         void inNext() {
167             if (!curr->rTag)
168                 curr = curr->right;
169             else {
170                 curr = curr->right;
171                 while (curr->lTag)
172                     curr = curr->left;
173             }
174         }
175 
176         /**
177          *  后序遍历的后继结点:
178          *  (1)
179          当一个节点是它的双亲节点的右孩子时
180          它的后序遍历的后继就是父节点。
181          *  (2)
182          当它是父节点的左孩子，且父节点没有
183          子树时，它的后序遍历的后继也是父
184          点。
185          *  (3)
```

```
186          当它是父节点的左孩子，且父节点有右 ⤸
187          树时，它的后序遍历的后继是父节点 ⤸
188          子树中的最左节点（如果左子树为空，⤸
189          右子树不空，则为右子树的最左节点）
190          */
191         void postNext() {
192             const BitNode *father = findFather(curr);
193             if (father->right == curr || (father->left == ⤸
194             curr && (!father->rTag || father->right == ⤸
195             nullptr)))
196                 curr = father;
197             else if (father->left == curr && father->rTag) {
198                 curr = father->right;
199                 do {
200                     while (curr->lTag) {
201                         curr = curr->left;
202                     }
203                     if (curr->rTag)
204                         curr = curr->right;
205                 } while (curr->lTag);
206 
207             }
208         }
209 
210         /**
211          * 查找一个节点的父节点
212          * @param child
213          * @return
214          * @details
215          */
216         const BitNode *findFather(const BitNode *child) {
217             const BitNode *father = child;
218             while (father->rTag)
219                 father = father->right;
220             const BitNode *lFather = father->left;
221             if (lFather->right == child && lFather->rTag)
222                 return lFather;
```

```
223             const BitNode *rFather = father->right;
224             if (rFather->left == child && rFather->lTag)
225                 return rFather;
226 
227             father = child;
228             while (father->lTag)
229                 father = father->left;
230             lFather = father->left;
231             if (lFather->right == child && lFather->rTag)
232                 return lFather;
233             rFather = father->right;
234             if (rFather->left == child && rFather->lTag)
235                 return rFather;
236         }
237 
238     };
239 
240     /** 层序建树 */
241     static BitTree CreateBitTree(std::vector<Comparable> vec,
242     bool skip = false, Comparable END = Comparable()) {
243         BitTree<Comparable> bitTree;
244         std::queue<BitTree::BitNode **> nodeQueue;
245         nodeQueue.push(&bitTree.root);
246         for (const auto &data:vec) {
247             BitTree::BitNode **node = nodeQueue.front();
248             nodeQueue.pop();
249             if (skip && data == END) {
250                 continue;
251             }
252             *node = new BitNode(data, nullptr, nullptr);
253             nodeQueue.push(&(*node)->left);
254             nodeQueue.push(&(*node)->right);
255 
256         }
257         return bitTree;
258     }
259 
```

```
260     /** 先序 + 中序建树 中序 + 后序建树 */
261     static BitTree CreateBitTree(std::vector<Comparable> ↵
262     vecA,
263                                  std::vector<Comparable> ↵
264                                  vecB,
265                                  Mode mode = DLR) {
266         BitTree<Comparable> bitTree;
267         if (vecA.size() == 0 || vecA.size() != vecB.size())
268             bitTree.root = nullptr;
269         else if (mode == DLR)
270             bitTree.root = CreateNodePreMiddle(vecA, vecB, 0,↵
271             vecA.size() - 1, 0, vecB.size() - 1);
272         else
273             bitTree.root = CreateNodeMiddlePost(vecA, vecB, ↵
274             0, vecA.size() - 1, 0, vecB.size() - 1);
275         return bitTree;
276     }
277
278     /** 将二叉树线索化 */
279     static void BitTreeThreading(BitTree &tree) {
280         tree.header = new BitNode(Comparable(), tree.root, ↵
281         nullptr);
282         tree.header->right = tree.header;
283         tree.header->rTag = false;
284         BitNodePtr *curr = &tree.root;
285         BitNodePtr *pre = &tree.header;
286         if (*curr == nullptr)
287             return;
288         std::stack<BitNodePtr *> nodeStack;
289         while (*curr != nullptr || !nodeStack.empty()) {
290             while (*curr != nullptr) {
291                 //中序遍历，将左子树入栈
292                 nodeStack.push(curr);
293                 curr = &(*curr)->left;
294             }
295             if (!nodeStack.empty()) {
296                 //当前栈顶是最左子树
```

```
297                 curr = nodeStack.top();
298                 //访问左子树，而后出栈，访问栈
299                 的右子树
300                 (*curr)->lTag = (*curr)->left != nullptr;
301                 (*curr)->rTag = (*curr)->right != nullptr;
302
303                 if (!(*curr)->lTag) {
304                     (*curr)->left = *pre;
305                 }
306
307                 if ((*pre)->right == nullptr) {
308                     (*pre)->rTag = false;
309                     (*pre)->right = *curr;
310                 }
311
312                 pre = curr;
313                 nodeStack.pop();
314                 curr = &(*curr)->right;
315             }
316         }
317         (*pre)->right = tree.header;
318         tree.header->right = *pre;
319     }
320
321     /** 获取不同访问模式的迭代器 */
322     Iterator getIterator(Mode mode = Mode::DLR) {
323         if (header == nullptr)
324             BitTreeThreading(*this);
325         return Iterator(header, mode);
326     }
327
328     /** 线索遍历各节点，可更改模式 */
329     void threadVisit(Mode mode = DLR, const std::function<
330     void(const BitNode *)> &visit = EASY_VISIT_LAMBDA) {
331         Iterator iter = getIterator(mode);
332         while (!iter.IsEnd()) {
333             visit(*iter);
```

```
334             iter++;
335         }
336     }
337
338     /** 选用不同模式进行遍历，递归式 */
339     void visitTreeR(Mode mode = DLR, const std::function<
340     void(BitNode *)> &visit = EASY_VISIT_LAMBDA) const {
341         switch (mode) {
342         default:
343         case DLR:visitTreeR(root, visit);
344             break;
345         case LDR:visitTreeMR(root, visit);
346             break;
347         case LRD:visitTreeBR(root, visit);
348             break;
349         }
350     }
351
352     /** 选用不同模式进行遍历，非递归式 */
353     void visitTree(Mode mode = DLR, const std::function<void(
354     BitNode *)> &visit = EASY_VISIT_LAMBDA) const {
355         switch (mode) {
356         default:
357         case DLR:visitTree(root, visit);
358             break;
359         case LDR:visitTreeM(root, visit);
360             break;
361         case LRD:visitTreeB(root, visit);
362             break;
363         }
364     }
365
366     /** 层序便利各节点 */
367     void visitTreeCover(const std::function<void(BitNode *)>
368     &visit = EASY_VISIT_LAMBDA) const {
369         std::queue<BitNode *> nodeQueue;
370         BitNode *node = root;
```

```
371         while (node != nullptr) {
372             if (node->left != nullptr && node->lTag)
373                 nodeQueue.push(node->left);
374             if (node->right != nullptr && node->rTag)
375                 nodeQueue.push(node->right);
376             visit(node);
377             if (!nodeQueue.empty()) {
378                 node = nodeQueue.front();
379                 nodeQueue.pop();
380             } else
381                 node = nullptr;
382         }
383     }
384
385     /** 将 x 插入树中，忽略重复项 */
386     void insertR(const Comparable &x) {
387         insertR(x, root);
388     }
389     void insert(const Comparable &x) {
390         insert(x, root);
391     }
392
393     /** 拷贝赋值 */
394     BitTree &operator=(const BitTree &rhs) {
395         makeEmpty();
396         root = clone(rhs.root);
397         return *this;
398     }
399
400 private:
401     struct BitNode {
402         // 约定 lTag 为 true 时，left 或 right 是有效值
403         Comparable data;
404         bool lTag;
405         bool rTag;
406         BitNode *left;
407         BitNode *right;
```

```
408         BitNode(const Comparable &theData, BitNode *lt, ↩
409         BitNode *rt)
410             : data(theData), left(lt), right(rt), lTag(true),↩
411             rTag(true) {}
412         BitNode(const Comparable &&theData, BitNode *lt, ↩
413         BitNode *rt)
414             : data(std::move(theData)), left(lt), right(rt), ↩
415             lTag(true), rTag(true) {
416         }
417     };
418
419     /** 树的根节点 */
420     BitNode *root;
421     /** 线索树的头 */
422     BitNode *header;
423
424     void makeEmpty(BitNode *&t) {
425         if (t != nullptr) {
426             if (t->lTag)
427                 makeEmpty(t->left);
428             if (t->rTag)
429                 makeEmpty(t->right);
430             delete t;
431         }
432         t = nullptr;
433     }
434     void insertR(const Comparable &x, BitNode *&t) {
435         if (t == nullptr)
436             t = new BitNode(x, nullptr, nullptr);
437         else if (x < t->data)
438             insertR(x, t->left);
439         else if (t->data < x)
440             insertR(x, t->right);
441         else
442             return;
443     }
444     void insert(const Comparable &x, BitNode *&t) {
```

```
445         if (t == nullptr) {
446             t = new BitNode(x, nullptr, nullptr);
447         } else {
448             BitTree<Comparable>::BitNode *node = t;
449             while (true) {
450                 //相等则忽略
451                 if (x == node->data)
452                     return;
453                 if (x < node->data) {
454                     if (node->left != nullptr)
455                         node = node->left;
456                     else {
457                         node->left = new BitNode(x, nullptr, 
458                         nullptr);
459                         return;
460                     }
461                 } else {
462                     if (node->right != nullptr)
463                         node = node->right;
464                     else {
465                         node->right = new BitNode(x, nullptr,
466                         nullptr);
467                         return;
468                     }
469                 }
470             }
471         }
472     }
473 
474     /** 先序遍历 */
475     void visitTreeR(BitNode *t, const std::function<void(
476     BitNode *)> &visit) const {
477         if (t != nullptr) {
478             visit(t);
479             if (t->lTag)
480                 visitTreeR(t->left, visit);
481             if (t->rTag)
```

```
482                 visitTreeR(t->right, visit);
483             }
484         }
485         /** 中序遍历 */
486         void visitTreeMR(BitNode *t, const std::function<void(
487         BitNode *)> &visit) const {
488             if (t != nullptr) {
489                 if (t->lTag)
490                     visitTreeMR(t->left, visit);
491                 visit(t);
492                 if (t->rTag)
493                     visitTreeMR(t->right, visit);
494             }
495         }
496         /** 后序遍历 */
497         void visitTreeBR(BitNode *t, const std::function<void(
498         BitNode *)> &visit) const {
499             if (t != nullptr) {
500                 if (t->lTag)
501                     visitTreeBR(t->left, visit);
502                 if (t->rTag)
503                     visitTreeBR(t->right, visit);
504                 visit(t);
505             }
506         }
507
508         /** 先序遍历 */
509         void visitTree(BitNode *t, const std::function<void(
510         BitNode *)> &visit) const {
511             if (t == nullptr)
512                 return;
513             std::stack<BitTree<Comparable>::BitNode *> nodeStack;
514             BitTree<Comparable>::BitNode *curr = t;
515             while (curr != nullptr || !nodeStack.empty()) {
516                 while (curr != nullptr) {
517                     //先序遍历，所以可以直接访问，
518                     后入栈，访问左子树
```

```
519                 visit(curr);
520                 nodeStack.push(curr);
521                 if (curr->lTag)
522                     curr = curr->left;
523                 else
524                     break;
525             }
526             //栈不空
527             if (!nodeStack.empty()) {
528                 //之前左子树已经访问过了，栈顶
529                 定是之前的最左子树
530                 //将其出栈访问其右子树
531                 curr = nodeStack.top();
532                 nodeStack.pop();
533                 if (curr->rTag)
534                     curr = curr->right;
535                 else
536                     curr = nullptr;
537             }
538         }
539 
540     }
541     /** 中序遍历 */
542     void visitTreeM(BitNode *t, const std::function<void(
543     BitNode *)> &visit) const {
544         if (t == nullptr)
545             return;
546         std::stack<BitTree<Comparable>::BitNode *> nodeStack;
547         BitTree<Comparable>::BitNode *curr = t;
548         while (curr != nullptr || !nodeStack.empty()) {
549             while (curr != nullptr) {
550                 //中序遍历，将左子树入栈
551                 nodeStack.push(curr);
552                 if (curr->lTag)
553                     curr = curr->left;
554                 else
555                     break;
```

```
556             }
557             if (!nodeStack.empty()) {
558                 //当前栈顶是最左子树
559                 curr = nodeStack.top();
560                 //访问左子树，而后出栈，访问栈
561                 的右子树
562                 visit(curr);
563                 nodeStack.pop();
564                 if (curr->rTag)
565                     curr = curr->right;
566                 else
567                     curr = nullptr;
568             }
569         }
570     }
571     /** 后序遍历 */
572     void visitTreeB(BitNode *t, const std::function<void(
573     BitNode *)> &visit) const {
574         if (t == nullptr)
575             return;
576         std::stack<BitTree<Comparable>::BitNode *> nodeStack;
577         BitTree<Comparable>::BitNode *curr = t;
578         BitTree<Comparable>::BitNode *visited = nullptr;
579         while (curr != nullptr || !nodeStack.empty()) {
580             while (curr != nullptr) {
581                 //左子树全部入栈
582                 nodeStack.push(curr);
583                 if (curr->lTag)
584                     curr = curr->left;
585                 else
586                     break;
587             }
588             //取得最左子树
589             curr = nodeStack.top();
590             if (curr->right == nullptr || !curr->rTag ||
591             curr->right == visited) {
592                 //右子树为空或者已经访问过了
```

```
593                 //说明当前该访问中间了
594                 visit(curr);
595                 visited = curr;
596                 nodeStack.pop();
597                 curr = nullptr;
598             } else 
599             //右子树没有访问过，访问右子树
600             if (curr->rTag)
601                 curr = curr->right;
602             else
603                 curr = nullptr;
604         }
605     }
606 
607     /** 前序 + 中序建树 */
608     static BitNode *CreateNodePreMiddle(const std::vector<
609     Comparable> &vec,
610                                         const std::vector<
611                                         Comparable> &vecM,
612                                         unsigned long start,
613                                         unsigned long end,
614                                         unsigned long startM,
615                                         unsigned long endM) {
616         auto *root = new BitNode(vec[start], nullptr, 
617         nullptr);
618         if (startM == endM && start == end && startM == 
619         start)
620             return root;
621 
622         unsigned long rootM;
623         for (rootM = startM; rootM <= endM; ++rootM)
624             if (vecM[rootM] == vec[start])
625                 break;// 找到中序的根节点
626 
627         // |root|   left   |   right   |
628         // |   left   |root|   right   |
629 
```

```
630         if (rootM > startM)
631             root->left = CreateNodePreMiddle(vec, vecM, ↩
632             start + 1, start + rootM - startM, startM, rootM ↩
633             - 1);
634 
635         if (rootM < endM)
636             root->right = CreateNodePreMiddle(vec, vecM, ↩
637             start + rootM - startM + 1, end, rootM + 1, endM)↩
638             ;
639 
640         return root;
641     }
642 
643     /** 中序 + 后序建树 */
644     static BitNode *CreateNodeMiddlePost(const std::vector<↩
645     Comparable> &vecM,
646                                          const std::vector<↩
647                                          Comparable> &vec,
648                                          unsigned long ↩
649                                          startM,
650                                          unsigned long endM,
651                                          unsigned long start,
652                                          unsigned long end) {
653         auto *root = new BitNode(vec[end], nullptr, nullptr);
654         if (startM == endM && start == end && start == ↩
655         startM)
656             return root;
657 
658         unsigned long rootM;
659         for (rootM = startM; rootM <= endM; ++rootM)
660             if (vecM[rootM] == vec[end])
661                 break;// 找到中序的根节点
662 
663         // |   left   |   right   |root|
664         // |   left   |root|   right   |
665 
666         if (rootM > startM)
```

```
667             root->left = CreateNodeMiddlePost(vecM, vec, ↩
668             startM, rootM - 1, start, start + (rootM - 1 - ↩
669             startM));
670 
671         if (rootM < endM)
672             root->right = CreateNodeMiddlePost(vecM, vec, ↩
673             rootM + 1, endM, start + rootM - startM, end - 1)↩
674             ;
675 
676         return root;
677     }
678 
679     BitNode *clone(BitNode *t) const {
680         if (t == nullptr)
681             return nullptr;
682         else if (t->lTag && t->rTag)
683             return new BitNode(t->data, clone(t->left), ↩
684             clone(t->right));
685         else if (t->lTag)
686             return new BitNode(t->data, clone(t->left), ↩
687             nullptr);
688         else if (t->rTag)
689             return new BitNode(t->data, nullptr, clone(t->↩
690             right));
691     }
692 
693 };
694 
695 #endif //PROJECT_BITTREE_H
696 
```