试实现贝努里队列类

【解】贝努里队列和其他的优先级队列的实现有些不同。它不是用一棵树来表示而是用一个森林来表示。在贝努里队列的实现中，我们可以借助于第 5 章中实现的 tree 类。用 tree 类的对象保存贝努里树。保存一个贝努里队列就是保存一个树根的数组。数组的 0 号单元存储 B0 的根结点的地址，1 号单元存储 B1 的根结点的地址，以此类推。贝努里队列的公有行为是优先级队列的行为，再加上归并优先级队列。贝努里队列类的定义见代码清单 6-18。

代码清单 6-18 贝努里队列类的定义

```
1.  template <class T>
2.  class Binomial {
3.      tree<T>::node **forest;      // 存放贝努里森林中的每棵树的树根的数组
4.      int noOfTree;                // 数组规模
5.
6.      tree<T>::node *merge(tree<T>::node *t1,tree<T>::node *t2);//归并贝努里树
7.      int findmin();               // 找出根结点值最小的树
8.      void deleteTree(tree<T>::node *rt) {   // 释放以 rt 为根的树的空间
9.          tree<T>::node *son = rt->son, *t;
10.         while (son != NULL) {
11.             t = son;
12.             son = son->brother;
13.             deleteTree(t);
14.         }
15.         delete rt;
16.     }
17.
18.  public:
19.      Binomial(int n = 100)              // n 是队列中预计元素个数
20.      {    noOfTree = int (log(n) / log(2)) + 1; // 计算所需的贝努里树的阶数
21.          forest = new tree<T>::node*[noOfTree];
22.          for (int i = 0; i < noOfTree; ++i)      forest[i] = NULL;
23.      }
24.
25.    ~Binomial()
26.    {  for (int i =0 ; i < noOfTree; ++i)  // 删除每一棵树
27.              if (forest[i] != NULL) deleteTree(forest[i]);
28.          delete [] forest;              // 删除存储树根的数组
29.      }
30.      void enQueue(T x);
31.      T deQueue();
32.      bool isEmpty();
33.      T getHead();
34.      void merge(Binomial &other);
35. };
```

贝努里队列类有两个数据成员，表示一个动态数组。forest 是一个指向树类结点指针数组，指向每一棵贝努里树的树根。noOfTree 是数组的规模。它的公有行为有构造、析构、入队、出队、判队空、获取队头元素和归并两个优先级队列。构造函数有一个参数 n，表示预计的队列的元素个数。构造函数根据 n 计算出存储 n 个结点需要多少棵贝努里树，按照计算的结果申请数组的空间。构造时队列为空，所以将所有的树都置成空树。析构函数释放所有的树占用的空间，最后释放存储树根的数组空间。由于在第 5 章定义的 tree 类中没有定义析构函数，于是我们自己定义了一个私有的成员函数 deleteTree 删除一棵以 rt 为根的树 。deleteTree 函数采用递归实现的方式。删除一棵树就是删除它的所有子树，最后删除根结点。

贝努里队列主要的功能是实现归并。而且其他的操作，如入队和出队也是依赖于归并操作。下面我们讨论一下归并操作的实现。归并函数 merge 有一个参数 other，它将贝努里队列 other 归并入当前对象。归并操作的实现见代码清单 6-19。

代码清单 6-19 归并函数的实现

```
1.  template< class T>
2.  void Binomial<T>::merge(Binomial &other)
3.  {    tree<T>::node **tmp = forest, *carry;  // 当前对象暂存 tmp 中
4.       int tmpSize = noOfTree;              // 当前对象规模暂存在 tmpSize
5.       int min = noOfTree < other.noOfTree ? noOfTree : other.noOfTree;
6.       int i;
7.
8.       if (noOfTree < other.noOfTree) {      // 确定归并后的森林规模
9.            noOfTree = other.noOfTree;
10.           if (other.forest[noOfTree] != NULL) ++noOfTree;
11.      }
12.      else if (forest[noOfTree] != NULL) ++noOfTree;
13.
14.      forest = new tree<T>::node*[noOfTree]; // 为归并后的队列申请空间
15.      for (i = 0; i < noOfTree; ++i) forest[i] = NULL;
16.
17.      carry = NULL;                        // 初始时进位为空
18.   for (i = 0; i < min; ++i) {           // 归并两个队列中阶数相同的树
19.           if (carry == NULL) {             // 没有进位
20.                if (tmp[i] == NULL) forest[i] = other.forest[i];
21.                else { if (other.forest[i] == NULL) forest[i] = tmp[i];
22.                       else  carry = merge(other.forest[i], tmp[i]);  }
23.           }
24.           else {                           // 有进位
25.                if (tmp[i] != NULL && other.forest[i] != NULL) {
26.                     forest[i] = carry;
27.                     carry = merge(other.forest[i], tmp[i]);
28.                }
29.                else {
```

```
30.                         if (tmp[i] == NULL && other.forest[i] == NULL) {
31.                             forest[i] = carry; carry = NULL;}
32.                       else if (tmp[i] == NULL) carry = merge(other.forest[i], carry);
33.                             else carry = merge(tmp[i], carry);
34.                   }
35.             }
36.       }
37.
38.     // 一个队列已结束
39.       if (other.noOfTree == min) {              // other 已结束
40.             for (; i < tmpSize; ++i)
41.                   if (carry == NULL) forest[i] = tmp[i];
42.                   else if (tmp[i] == NULL)
43.                         { forest[i] = carry; carry = NULL;}
44.                       else carry = merge(tmp[i], carry);
45.       }
46.       else {                                    // 当前队列已结束
47.             for (; i < other.noOfTree; ++i)
48.                   if (carry == NULL) forest[i] = other.forest[i];
49.                   else if (other.forest[i] == NULL) {
50.                             forest[i] = carry; carry = NULL;}
51.                       else  carry = merge(other.forest[i], carry);
52.       }
53.     if (carry != NULL) forest[i] = carry;
54.
55.       for (i = 0; i < other.noOfTree; ++i) other.forest[i] = NULL;
56.       delete [] tmp;
57.  }
```

归并两个贝努里队列需要按照贝努里树的阶数由低到高依次检查两个森林中的树。为此，我们先为归并的结果森林申请存储的空间。归并结果存放在当前对象中，于是先将当前对象中的森林临时存放在变量 tmp 中，然后计算新的森林的规模。取两个森林中规模大的作为结果的规模，然后再检查规模大的那个森林是不是存在最高阶数的贝努里树。如果存在，在归并过程中可能会增加出一棵比它规模更大的树，于是将结果森林的规模再增加 1。按照新的规模申请存储空间并初始化。从第 17 行开始了归并的过程。第 17 行的循环处理的是两个森林中都有尚未归并的树，归并两个森林中阶数相同的树。在归并时还需要考虑上一次归并有没有产生一棵更高阶的树。所以在每次归并时要考虑三棵树。如果三棵树中有两棵是空树，则将那棵非空树放入结果森林。如果有两棵树非空，将这两棵树归并起来作为进位。如果有三棵非空树，将上次归并的进位作为结果存入结果森林，将另外两棵树归并起来作为下一次归并的进位。

当两个森林中有一个已经结束时，需要将另一个森林的剩余部分与进位继续归并。最后检查最后一次归并有没有产生进位，如果有则存入结果森林。归并结束。

最后还有一些善后的工作要做。由于队列 other 已经归并入当前队列，所以必须将 other 置空，再释放 tmp 的空间。

在这个归并过程中经常会用到归并两棵同阶的贝努里树。我们把这个过程设计为贝努里队列类的私有成员函数。它的过程如代码清单 6-20 所示。

代码清单 6-20 归并两棵贝努里树

```
1.  template< class T>
2.  tree<T>::node *Binomial<T>::merge(tree<T>::node *t1, tree<T>::node *t2)
3.  {
4.      tree<T>::node *min, *max;
5.
6.      if (t1->data < t2->data)                    // 确定树根
7.        {   min = t1; max = t2;}
8.      else { min = t2; max = t1;}
9.
10.     if (min->son ==NULL) min->son = max;
11.     else {
12.         tree<T>::node *t = min->son;
13.         while (t->brother != NULL)      t = t->brother;
14.         t->brother = max;
15.     }
16.     return min;
17. }
```

归并两棵贝努里树就是将根结点值大的树作为根结点值小的树的儿子。函数首先决定哪棵树作为树根。min 是树根，max 要被作为 min 的儿子。如果 min 没有儿子，max 作为他的第一个儿子。如果 min 有儿子，则沿着 min 儿子的兄弟链找到最后一个儿子，将 max 作为这个儿子的兄弟。

有了归并函数以后，入队和出队操作就很容易实现。入队操作先生成一个只有一个元素的队列。将这个队列归并到当前队列。该过程如代码清单 6-21 所示。

代码清单 6-21 enQueue 函数的实现

```
1.  template< class T>
2.  void Binomial<T>::enQueue(T x)
3.  {   Binomial tmp(1);
4.      tmp.forest[0] = new tree<T>::node(x);
5.      merge(tmp);
6.  }
```

deQueue 操作先找到根结点值最小的树的位置。由于这个操作在 getHead 中也要用到，于是我们设计了一个私有的成员函数 findmin，它的实现如代码清单 6-23 所示。有了这个函数 deQueue 的实现就很容易。首先调用 findmin 找到根结点最小的树，返回他在

森林中的下标值。这个下标值决定了这棵树的子树个数。如果下标值为 k，则删除根结点后会产生 k 棵子树。deQueue 函数先保存根结点的值，在当前队列中删除这棵树，然后将这 k 棵子树生成一个优先级队列，并归并到当前队列中。这个过程如代码清单 6-22 所示。

代码清单 6-22 deQueue 函数的实现

```
1.  template< class T>
2.  T Binomial<T>::deQueue()
3.  {   T value;
4.      int min = findmin();      // 找到根结点最小的这棵树
5.
6.      if (min == 0) {          // 删除 0 阶树
7.          value = forest[0]->data;
8.          delete forest[0];
9.      forest[0] = NULL;
10.         return value;
11.     }
12.
13.     tree<T>::node *t = forest[min], *son, *brother; //t：包含最小结点的树根
14.     int sizeOfQueue = int(pow(2,min) - 1);
15.     Binomial tmp(sizeOfQueue);                  // 存储产生的子树
16.     value = t->data;
17.     forest[min] = NULL;              // 从当前树中删除根结点值最小的树 t
18.
19.     son = t->son;                   // 找到第一棵子树
20.     delete t;                      // 释放结点 t
21.     int i = 0;
22.     do {                          // 将所有子树存入 tmp
23.         tmp.forest[i++] = son;
24.         brother = son->brother;
25.         son->brother = NULL;
26.     } while ((son = brother) != NULL);
27.
28.     merge(tmp);            // 归并 tmp
29.
30.     return value;
31. }
```

Findmin 函数顺序扫描每棵树的树根，从中找出根结点值最小的树，返回这棵树的阶数。它的实现见代码清单 6-23。

代码清单 6-23 findmin 函数的实现

```
1.  template< class T>
2.  int Binomial<T>::findmin()
3.  {   int min;
```

```
4.
5.          // 找到第一棵非空树 min
6.      for (int i = 0; i < noOfTree && forest[i] == NULL; ++i);
7.          min  = i;
8.
9.          for (; i < noOfTree ; ++i)    // 扫描后续的树，找出根结点最小的树
10.             if (forest[i] != NULL && forest[i]->data < forest[min]->data) min = i;
11.
12.         return min;
13. }
```

剩下的两个函数实现都很简单。isEmpty 函数扫描整个森林，只要森林中有一棵树存在，队列就非空。getHead 函数首先调用 findmin 函数得到根结点值最小的树，然后返回它的值。这两个函数的实现见代码清单 6-24。

代码清单 6-24 isEmpty 和 getHead 函数的实现

```
14. template< class T>
15. bool Binomial<T>::isEmpty()
16. {     for (int i =0; i < noOfTree; ++i)
17.             if ( forest[i] != NULL) return false;
18.         return true;
19.
20. }
21.
22. template< class T>
23. T Binomial<T>::getHead()
24. {     int pos = findmin();
25.         return forest[pos]->data;
26. }
```