

# 数据结构与算法 II 上机实验 (11.24)

中国人民大学 信息学院 崔冠宇 2018202147

注：请使用支持 C++17 或以上标准的编译器！

上机题 1 实现 Fibonacci 堆的插入、合并、抽取最小值操作。

## 一、问题描述

根据老师课堂讲解，实现 Fibonacci 堆的基本操作：插入节点、合并两堆以及抽取最小值。

1. 输入：因为本次试验是实现数据结构，无指定输入，因此在源文件中给出一些插入、合并、抽取最小值等操作序列来代替输入。
2. 输出：无指定输出，因此在执行一部分操作后打印堆的情况来代替输出。

## 二、算法基本思路

首先给出 Fibonacci 堆的结构（我的实现与老师课堂上的实现方法略有不同）：

```
1 // Fibonacci 堆中节点的结构
2 class FibNode
3 {
4     KeyType key;           // 关键字
5     FibNode * parent;      // 父节点指针
6     list children;         // 子节点双向循环链表
7     bool mark;             // 剪枝标记
8     // 不用存度数，可以直接调用 list.size() 获得
9     // 不用存左右指针，可以改为将节点放在链表中，
10    // 链表会有迭代器，从而代替了左右指针
11 }
12 // Fibonacci 堆的结构
13 class FibHeap
14 {
15     list roots;            // 根表
16     list_iterator * min;    // 指向根表中最小值的迭代器
17
18     insert(KeyType key);    // 插入关键字为key的节点
19     merge(FibHeap other);   // 与另一个堆合并（避免与union关键字重名）
20     deleteMin();            // 删除最小值
21 }
```

接下来分析各操作的实现方法：

## 1. 插入节点：

插入一个新节点的步骤如下：

- (a) 新建一个单节点；
- (b) 在根表中 `min` 节点的左边插入该节点；
- (c) 比较新节点的关键字与 `min` 节点的关键字，决定是否修改 `min` 迭代器的位置。

下面给出 `insert` 操作的伪代码：

```
1 FibHeap::insert(key):
2     // 创建节点
3     node = new FibNode(key)
4     // 插在 min 之前
5     roots.insert(min, node)
6     // 可能更新最小值节点
7     if not (roots.size() > 1 and (*min) -> key < key):
8         min--
```

## 2. 合并操作：

合并两堆的步骤如下：

- (a) 将两堆的根表合并；
- (b) 比较两堆的 `min` 节点的大小，决定是否修改 `min` 迭代器的位置。

下面给出 `merge` 操作的伪代码：

```
1 FibHeap::merge(other):
2     // 如果本堆为空
3     if roots.size() == 0:
4         min = other.min;
5     // 如果另一堆为空
6     else if other.roots.size() == 0:
7         pass
8     // 两堆都不空且另一堆最小值小于本堆最小值
9     else if (*(other.min)) -> getKey() < (*(this -> min)) -> getKey():
10         min = other.min;
11     // 合并链表
12     roots.splice(roots.end(), other.roots);
```

### 3. 抽取最小值:

在 Fibonacci 堆中抽取并删除最小值的步骤如下:

- (a) 将最小值节点移出根表;
- (b) 将最小值节点的所有子节点的父亲设为空, 并加入根表;
- (c) 从头到尾扫描根表, 按最小堆的方式合并节点以保证各根的度数不同, 同时维护 `min` 迭代器。

```
1 FibHeap::deleteMin():
2     // 空堆
3     if roots.size() == 0:
4         return NULL
5     // 移出根表
6     tmpMin = min
7     remove min from roots
8     // 父亲设为空, 加入根表
9     set parents of children of min to NULL
10    add children of min to roots
11    // 扫描根表, 合并节点保证根的度数不同
12    min = roots.begin()
13    for current = roots.begin() to roots.end():
14        update min
15        // 维护一个度数-指针数组
16        merge nodes while degrees are the same
17    return tmpMin.key
```

## 三、算法复杂性分析

对于 Fibonacci 堆  $H$ , 记  $t(H)$  为根表中节点的个数, 即最小堆的个数; 记  $m(H)$  为堆中被标记的节点个数; 再记势能  $\Phi(H) = t(H) + 2m(H)$ 。

### 1. 先分析 `insert` 操作的复杂度:

#### (a) 时间复杂度:

因为堆维护了 `min` 指针, 而向根表这一双向循环链表中插入节点是  $O(1)$  的, 所以整个 `insert` 算法的时间复杂度是  $O(1)$ 。

#### (b) 空间复杂度:

算法工作时需要常数级的辅助空间, 所以算法的空间复杂度是  $O(1)$  的。

### 2. 然后分析 `merge` 操作的复杂度:

(a) 时间复杂度：

`merge` 的主要操作是合并链表，而链表的合并只需要修改指针，所以仍是  $O(1)$  的。

(b) 空间复杂度：

算法工作时需要常数级的辅助空间，所以算法的空间复杂度是  $O(1)$  的。

3. 最后分析 `deleteMin` 操作的复杂度：设  $n$  节点的 Fibonacci 堆中最大度为  $D(n)$ 。

(a) 时间复杂度：

算法首先扫描 `min` 的子节点，把它们的父亲设为空，并加入根表，这个过程最多扫描了  $D(n)$  个节点，是  $O(D(n))$  的；算法之后再次扫描根表，至多扫描  $t(H) + D(n) - 1$  个节点，这个过程是  $O(t(H) + D(n))$  的；又因为合并之后根表的度数各不相同，所以有  $t(H') \leq D(H) + 1$ ，因此摊还代价  $\hat{c}_i \leq t(H) + D(n) - 1 + (D(n) + 1 + 2m(H)) - (t(H) + 2m(H)) = D(n)$ 。所以时间摊还代价为  $O(D(n))$ 。在仅支持上述三种操作时算法每次合并的根节点具有度数相同的特点，因此  $D(n) \leq \lfloor \log_2 n \rfloor$ ，所以该算法时间摊还代价  $O(\log n)$ 。

(b) 空间复杂度：

算法仅额外维护一个根度数-指针数组，因为度数最大为  $D(n)$ ，因此算法的空间复杂度是  $O(D(n)) = O(\log n)$  的。

## 四、程序源代码

Fibonacci 堆定义及相关操作: **FibHeap.h**:

```
1 #ifndef FIBHEAP_H
2 #define FIBHEAP_H
3
4 #include <iostream>
5 #include <list>
6 #include <unordered_map>
7
8 template <typename K>
9 class FibHeapNode
10 {
11     //friend class FibHeapNode<K>;
12 public:
13     FibHeapNode(K key);
14     ~FibHeapNode();
15     size_t getDegree();
16     K getKey();
17     std::list<FibHeapNode<K> *> & getChildren();
18     bool setParent(FibHeapNode<K> * newParent);
19     void print();
```

```
20     private:
21         // 左右节点指针不用，可以用链表代替
22         // 父节点
23         FibHeapNode<K> * parent;
24         // 子节点们
25         std::list<FibHeapNode<K> *> children;
26         // 关键字
27         K key;
28         // 度
29         // size_t degree = children.size();
30         // 剪枝标记
31         bool mark;
32 };
33
34 template <typename K>
35 FibHeapNode<K>::FibHeapNode(K key)
36 : parent(nullptr), key(key), mark(false){}
37
38 template <typename K>
39 FibHeapNode<K>::~~FibHeapNode()
40 {
41     for(auto i = children.begin(); i != children.end(); i++)
42     {
43         delete (*i);
44     }
45 }
46
47 template <typename K>
48 size_t FibHeapNode<K>::getDegree()
49 {
50     return children.size();
51 }
52
53 template <typename K>
54 K FibHeapNode<K>::getKey()
55 {
56     return key;
```

```
57 }
58
59 template <typename K>
60 std::list<FibHeapNode<K> *> & FibHeapNode<K>::getChildren()
61 {
62     return children;
63 }
64
65 template <typename K>
66 bool FibHeapNode<K>::setParent(FibHeapNode<K> * newParent)
67 {
68     this -> parent = newParent;
69     return true;
70 }
71
72 template <typename K>
73 void FibHeapNode<K>::print()
74 {
75     std::cout << "节点 " << key << " : " << std::endl << "\t";
76     if(parent == nullptr)
77     {
78         std::cout << "(无父节点)";
79     }
80     else
81     {
82         std::cout << "父节点: ( " << parent -> key << " )";
83     }
84     std::cout << std::endl << "\t";
85     if(children.size() == 0)
86     {
87         std::cout << "(无子节点)" << std::endl;
88     }
89     else
90     {
91         std::cout << "子节点: ( ";
92         for(auto i = children.begin(); i != children.end(); i++)
93         {
```

```

94         std::cout << (*i) -> key << " ";
95     }
96     std::cout << ")" << std::endl;
97     for(auto i = children.begin(); i != children.end(); i++)
98     {
99         (*i) -> print();
100    }
101 }
102 }
103
104
105 template <typename K>
106 class FibHeap
107 {
108     public:
109         FibHeap();
110         ~FibHeap();
111         // 插入新节点
112         bool insert(K key);
113         // 合并两个 Fib 堆
114         bool merge(FibHeap<K> & other);
115         // 删除最小值
116         K deleteMin();
117         void print();
118     private:
119         std::list<FibHeapNode<K>*> roots;
120         typename std::list<FibHeapNode<K>*>::iterator min;
121 };
122
123
124 template <typename K>
125 FibHeap<K>::FibHeap()
126 :min(roots.end()){}
127
128 template <typename K>
129 FibHeap<K>::~~FibHeap()
130 {

```

```
131     for(auto i = roots.begin(); i != roots.end(); i++)
132     {
133         delete (*i);
134     }
135 }
136
137 template <typename K>
138 bool FibHeap<K>::insert(K key)
139 {
140     FibHeapNode<K> * node = new FibHeapNode<K>(key);
141     // 分配失败
142     if(node == nullptr)
143     {
144         return false;
145     }
146     // 插入 min 前边
147     roots.insert(min, node);
148     // 更新最小值节点
149     if(!(roots.size() > 1 && (*min) -> getKey() < key))
150     {
151         min--;
152     }
153     return true;
154 }
155
156 template <typename K>
157 bool FibHeap<K>::merge(FibHeap<K> & other)
158 {
159     // 如果本堆为空
160     if(roots.size() == 0)
161     {
162         min = other.min;
163     }
164     // 如果另一堆为空
165     else if(other.roots.size() == 0)
166     {
167         // min = min;
```



```

168     }
169     // 两堆都不空且另一堆最小值小于本堆最小值
170     else if ((*other.min) -> getKey() < (*(this -> min) -> getKey()))
171     {
172         min = other.min;
173     }
174     // 合并链表
175     roots.splice(roots.end(), other.roots);
176     return true;
177 }
178
179 template <typename K>
180 K FibHeap<K>::deleteMin()
181 {
182     if(roots.size() == 0)
183     {
184         return K(0);
185     }
186     // 将最小值节点保存
187     FibHeapNode<K> * nodeToDelete = *min;
188     // min的各子节点父节点指针设为空
189     for(auto i = (*min) -> getChildren().begin(); i != (*min) -> getChildren().end()
190 (); i++)
191     {
192         (*i) -> setParent(nullptr);
193     }
194     // 最小值节点的子树加入根链表
195     roots.splice(min, (*min) -> getChildren());
196     // 摘下最小值节点
197     roots.erase(min);
198
199     // 开始从头扫描，维护根表
200     typename std::list<FibHeapNode<K>*>::iterator current = roots.begin();
201     min = current;
202     // 度数-指针映射，为了使各根的度数不同
203     std::unordered_map<size_t, typename std::list<FibHeapNode<K>*>::iterator>
degNodeMapping;

```

```

203     for(; current != roots.end(); current++)
204     {
205         // 先更新最小节点
206         if ((*current) -> getKey() < (*min) -> getKey())
207         {
208             min = current;
209         }
210         // 再进行冲突处理
211         size_t deg = (*current) -> getDegree();
212         // 若有冲突
213         if(degNodeMapping.find(deg) != degNodeMapping.end())
214         {
215             // 不断处理冲突
216             while(degNodeMapping.find(deg) != degNodeMapping.end())
217             {
218                 // 摘下两个冲突发生的节点
219                 // 之前的节点
220                 FibHeapNode<K> * leftNodeToMerge = *(degNodeMapping[deg]);
221                 // 摘下之前的节点
222                 roots.erase(degNodeMapping[deg]);
223                 // 当前节点
224                 FibHeapNode<K> * rightNodeToMerge = (*current);
225                 // 摘下当前节点，指针后移一下
226                 current = roots.erase(current);
227
228                 // 左边的键值小于右边的键值，右边的加到左边的根表里，左边的加回到链
表
229                 if(leftNodeToMerge -> getKey() < rightNodeToMerge -> getKey())
230                 {
231                     leftNodeToMerge -> getChildren().insert(leftNodeToMerge ->
getChildren().end(), rightNodeToMerge);
232                     rightNodeToMerge -> setParent(leftNodeToMerge);
233                     // 指针移回来
234                     current = roots.insert(current, leftNodeToMerge);
235                 }
236                 // 相反情况
237                 else

```

```

238         {
239             rightNodeToMerge -> getChildren().insert(rightNodeToMerge ->
getChildren().end(), leftNodeToMerge);
240             leftNodeToMerge -> setParent(rightNodeToMerge);
241             current = roots.insert(current, rightNodeToMerge);
242         }
243         // 更新度数-节点列表
244         degNodeMapping.erase(deg);
245         deg = (*current) -> getDegree();
246         //degNodeMapping[deg] = current;
247     }
248     degNodeMapping[deg] = current;
249 }
250 // 没有冲突
251 else
252 {
253     degNodeMapping[deg] = current;
254 }
255 }
256 K ret = nodeToDelete -> getKey();
257 delete nodeToDelete;
258 return ret;
259 }
260
261 template <typename K>
262 void FibHeap<K>::print()
263 {
264     std::cout << "Fibonacci 堆" << std::endl;
265     std::cout << "根表: ( ";
266     for(auto i = roots.begin(); i != roots.end(); i++)
267     {
268         std::cout << (*i) -> getKey() << " ";
269     }
270     std::cout << ")" << std::endl;
271     for(auto i = roots.begin(); i != roots.end(); i++)
272     {
273         (*i) -> print();

```

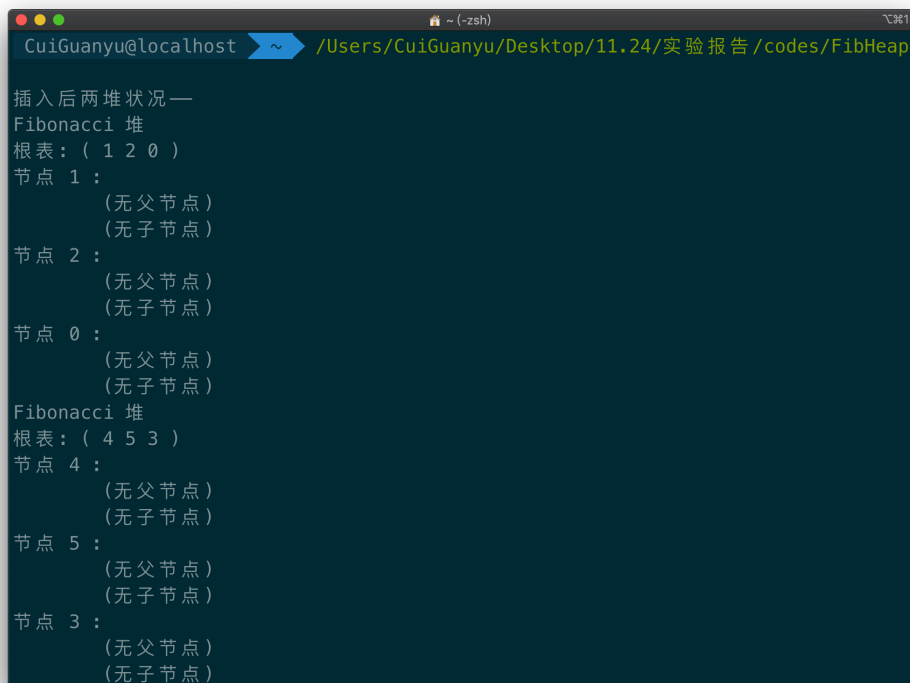
```
274     }
275 }
276
277 #endif
```

#### Fibonacci 堆测试: **FibHeap.cpp**:

```
1 #include <iostream>
2 #include "FibHeap.h"
3
4 int main(int argc, char *argv[])
5 {
6     const size_t N = 6;
7     FibHeap<size_t> h;
8     for(size_t i = 0; i < N / 2; i++)
9     {
10         h.insert(i);
11     }
12     FibHeap<size_t> h1;
13     for(size_t i = N / 2; i < N; i++)
14     {
15         h1.insert(i);
16     }
17     std::cout << "插入后两堆状况——" << std::endl;
18     h.print();
19     h1.print();
20     h.merge(h1);
21     std::cout << "将两堆合并后的状况——" << std::endl;
22     h.print();
23     h.deleteMin();
24     std::cout << std::endl;
25     std::cout << "删除最小值节点后堆的状况——" << std::endl;
26     h.print();
27     return 0;
28 }
```

#### 五、运行结果截图

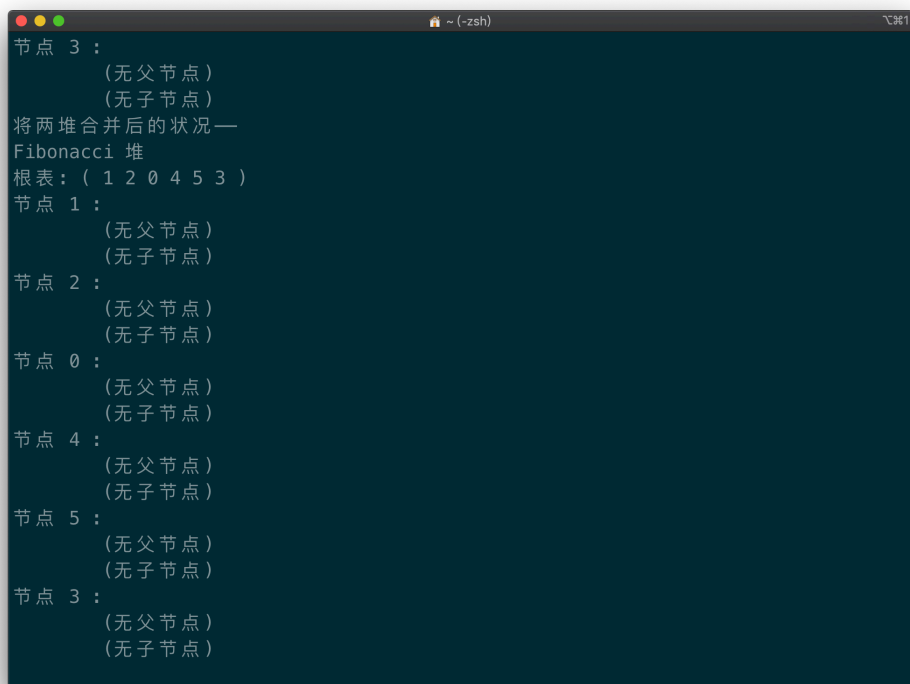
编译运行 `FibHeap.cpp`。程序运行时，先建立两个堆，并分别将 0、1、2 插入第一个堆，将 3、4、5 插入第二个堆；然后将两堆合并，最后将最小值删除。运行截图如下，观察可见程序执行正确。



```
CuiGuanyu@localhost ~ (~zsh) /Users/CuiGuanyu/Desktop/11.24/实验报告/codes/FibHeap

插入后两堆状况—
Fibonacci 堆
根表：( 1 2 0 )
节点 1：
    (无父节点)
    (无子节点)
节点 2：
    (无父节点)
    (无子节点)
节点 0：
    (无父节点)
    (无子节点)
Fibonacci 堆
根表：( 4 5 3 )
节点 4：
    (无父节点)
    (无子节点)
节点 5：
    (无父节点)
    (无子节点)
节点 3：
    (无父节点)
    (无子节点)
```

图 1: Fibonacci 堆插入测试



```
节点 3：
    (无父节点)
    (无子节点)
将两堆合并后的状况—
Fibonacci 堆
根表：( 1 2 0 4 5 3 )
节点 1：
    (无父节点)
    (无子节点)
节点 2：
    (无父节点)
    (无子节点)
节点 0：
    (无父节点)
    (无子节点)
节点 4：
    (无父节点)
    (无子节点)
节点 5：
    (无父节点)
    (无子节点)
节点 3：
    (无父节点)
    (无子节点)
```

图 2: Fibonacci 堆合并测试

```
      (无父节点)
      (无子节点)
节点 3 :
      (无父节点)
      (无子节点)

删除最小值节点后堆的状况 —
Fibonacci 堆
根表: ( 1 3 )
节点 1 :
      (无父节点)
      子节点: ( 2 4 )
节点 2 :
      父节点: ( 1 )
      (无子节点)
节点 4 :
      父节点: ( 1 )
      子节点: ( 5 )
节点 5 :
      父节点: ( 4 )
      (无子节点)
节点 3 :
      (无父节点)
      (无子节点)
CuiGuanyu@localhost ~
```

图 3: Fibonacci 堆删除最小值测试