数据结构与算法 I 实验 9

2019201409 于倬浩

2020 年 12 月 20 日

目录

一、实验内容

实现斐波那契堆。

额外实现了可视化。

二、核心操作 & 接口

所有核心操作命名和算法导论保持一致。

```
struct node { //节点类
    bool mark; //是否丢失过儿子
    int deg; //度数
    int val; //键值
    node *p, *ch, *l, *r; //父亲、儿子、左兄弟、右兄弟指针
};

struct Fibonacci_Heap{ //斐波那契堆类
    node *min; //指向根链表中最小元素
    int n; //当前堆的大小
};

// 以下方法提供给用户，内部实现涉及的其他函数不再列出。
```

```
inline node* fibHeapInsert(Fibonacci_Heap &h,int val);
// 向堆 h 中插入元素 val

inline Fibonacci_Heap fibHeapUnion(Fibonacci_Heap &a, Fibonacci_Heap &b);
// 合并堆 a、b，返回新堆

inline int fibHeapExtractMin(Fibonacci_Heap &h);
// 提取最小元素

inline void fibHeapDecreaseKey(Fibonacci_Heap &h, node *x, int val);
// 减小指定节点的数据

inline void fibHeapDelete(Fibonacci_Heap &h, node *key);
// 删除节点
```

三、算法设计

所有功能均按照算法导论给出的方法实现。

- 插入

  直接将当前元素当作根，插入根链表，单次操作运行时间 $\Theta(1)$。

  ```
  node* fibHeapInsert(Fibonacci_Heap &h,int val) {
      node *x = new node; //新建节点
      x->p = x->ch = null, x->l = x->r = x;
      x->deg = 0, x->val = val, x->mark = false;
      ++h.n;
      if(h.min == null) h.min = x; //根链表为空
      else { // 插入根链表
          x->l = h.min, x->r = h.min->r;
          h.min->r->l = x, h.min->r = x;
          if(x->val < h.min->val) h.min = x;
      }
      return x;
  }
  ```

- 合并堆

  类似插入操作，由于根链表是双向链表，因此可以用 $\Theta(1)$ 的运行时间拆开再合并两个链表，最后维护一下 `h.min` 即可，依旧很懒。

  ```
  Fibonacci_Heap fibHeapUnion(Fibonacci_Heap &a, Fibonacci_Heap &b) {
      if(a.min == null) return b; if(b.min == null) return a;
  ```

```
    Fibonacci_Heap ret;
    ret.min = a.min;
    node *p1 = a.min->r, *p2 = b.min->l; //合并根链表
    a.min->r = b.min, b.min->l = a.min;
    p1->l = p2, p2->r = p1;
    if(b.min->val < a.min->val) ret.min = b.min; //维护 min
    ret.n = a.n + b.n;
    return ret;
}
```

- 提取最小元

  首先判断操作是否非法，即堆是否为空。接下来，把 h.min 的所有儿子放到根链表中，并把 h.min 从根链表中删除。

  之后，再使用 Consolidate 维护堆性质即可。

  Consolidate 的大致做法是，维护一个数组，存储各种度数的节点，如果发现度数重复的就合并为度数 $+1$ 的，直到整个森林没有两棵树度数相同即可。最后，再把这个数组拉成一条链，构成堆的根链表。

  代码见下：

```
inline void Consolidate(Fibonacci_Heap &h) {
    node **a = new node*[h.n + 1]; //维护各种度数的根
    for(int i = 0; i <= h.n; ++i) a[i] = null;
    node *x = h.min, *end = h.min;
    int cnt = 0;
    do{ //统计有多少个根，并把根放入一个单独的数组，用来和修改后的区分
        ++cnt;
        x = x->r;
    }while(x != end);
    node **rootList = new node*[cnt];
    x = h.min, end = h.min, cnt = 0;
    do{
        rootList[cnt++] = x;
        x = x->r;
    }while(x != end);
    for(int i = 0; i < cnt; ++i) { //枚举每个原根链表中的节点
        x = rootList[i];
        int d = x->deg;
        while(a[d] != null) { //合并与当前节点度数相同的，直到不能合并
            node *y = a[d];
            if(x->val > y->val) {
```

```
                node *tmp = x;
                x = y, y = tmp;
            }
            fibHeapLink(h, y, x);
            a[d] = null;
            ++d;
        }
        a[d] = x;
    }
    h.min = null;
    for(int i = 0; i <= h.n; ++i) { //维护新的根链表
        if(a[i] != null) {
            if(h.min == null) {
                h.min = a[i];
                h.min->l = h.min->r = h.min;
            }
            else {
                h.min->l->r = a[i], a[i]->l = h.min->l;
                a[i]->r = h.min, h.min->l = a[i];
                if(a[i]->val < h.min->val) h.min = a[i];
            }
        }
    }
    delete a; //释放空间 防止内存泄漏
    delete rootList;
}
```

- 减小键值

  先处理不合法的情况，修改键值。接下来判断是否违反了堆性质，如果违反了就递归祖先节点，维护每个节点的 mark 即可，具体操作依赖 Cut 和 CascadingCut，实现起来也比较简单。

```
void fibHeapDecreaseKey(Fibonacci_Heap &h, node *x, int val) {
    if(val > x->val) // 输入不合法
        throw std::runtime_error("New key is greater than current key");
    x->val = val; //修改键值
    node *y = x->p;
    if(y != null && y->val > x->val) { //需要修改祖先节点的情况
        Cut(h, x, y);
        CascadingCut(h, y);
    }
```

```
        if(x->val < h.min->val) h.min = x; //维护 min
    }
```

- 删除节点
  将指定的节点 Decrease-Key 成最小的，接下来 Extract-Min 即可。

四、测试 & 可视化

造了一个 30 次操作的操作序列,并把结果做成了动图 result.gif;堆的核心代码见 fib.hpp，测试代码见 fib.cpp。

和之前的实验一样，为了更好地观察数据结构本身的性质以及方便调试，实现了一个可视化函数 dottify()，使用开源软件包 GraphViz 渲染图像。

对于测试程序，支持 Insert，Extract-Min，Decrease-Key。由于其他的几个操作都依赖这三个操作，因此检测了这三个操作的正确性，其他操作的正确性也就得到保证。

下面粘贴几张静态的测试结果：

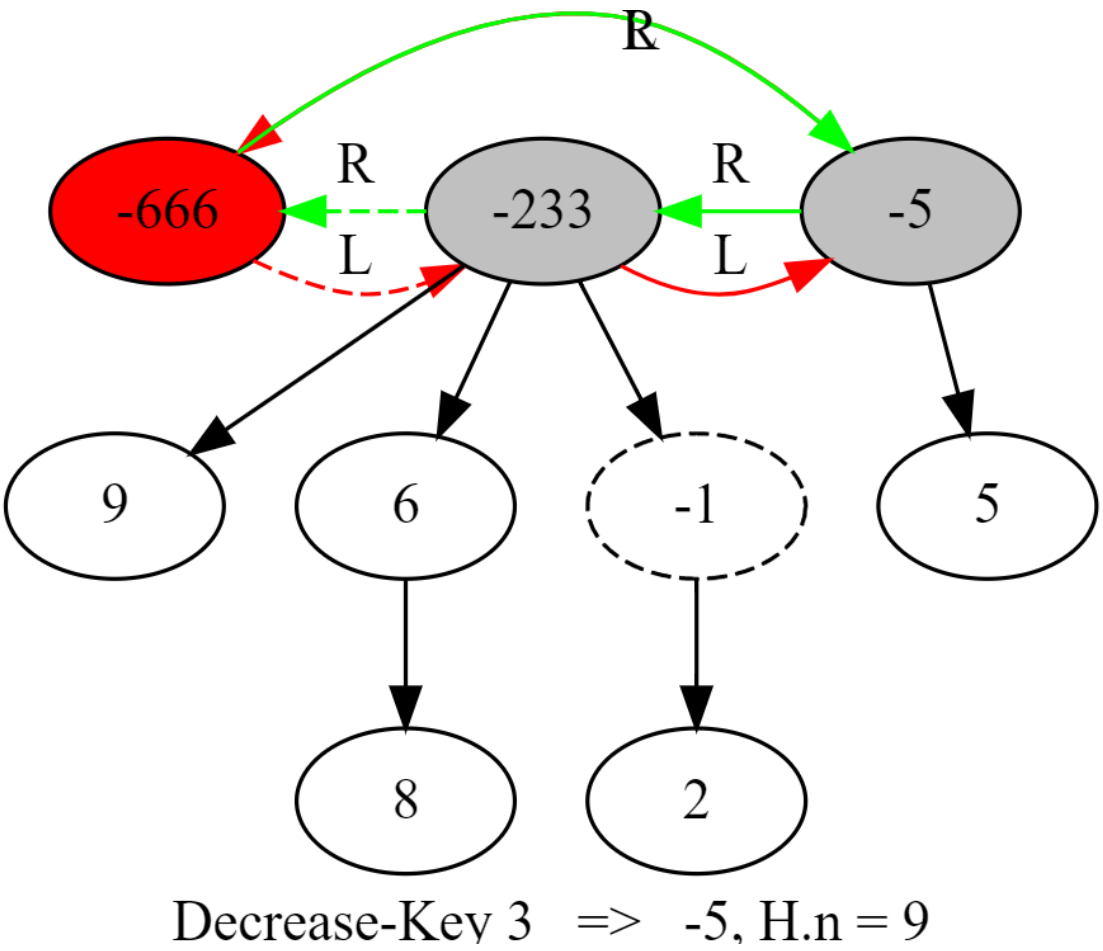

图 1: 一个操作序列对应的结果，上次操作是 Decrease-Key

红色节点表示当前 h.min，灰色节点表示根列表中的点，白色节点表示非根节点，边缘带虚

线的表示 `mark` 为 `true` 的节点，红绿色的边表示根链表，对于其他节点为了简洁起见，直接画出节点之间的关系，省略了链表。

具体交互方法如下：

```
➜ lab9 cd "/home/me/ds1/lab9/" && g++ fib.cpp -o fib && "/home/me/ds1/lab9/"fib
1000
Command #1:1 5
Command #2:1 8
Command #3:1 6
Command #4:1 3
Command #5:1 2
Command #6:1 9
Command #7:1 -1
Command #8:1 -666
Command #9:1 -233
Command #10:2
-666
Command #11:2
-233
Command #12:2
-1
Command #13:3 1 -1000
Command #14:2
-1000
Command #15:
```

图 2: 命令行交互方法

观察操作序列对应每个时刻堆的形态，即可确定算法的正确性。