数据结构与算法 I 实验 8

2019201409 于倬浩

2020 年 12 月 9 日

目录

一、实验内容

实现二项堆的各项操作。

额外实现了可视化，结果动图位于`./Result.gif`。

二、实现操作 & 接口

核心节点类定义如下：

```
struct node{
    int deg;
    //当前节点的度数
    node *fa, *ch, *sib;
    //分别表示当前节点的父亲、最左儿子、兄弟节点
    int* val;
    //指向当前节点的数据域的指针
};
typedef node* BinomialHeap; //简略定义
```

注意到数据类型 `BinomialHeap` 在不维护其他卫星数据的情况下，只有一个 `node*` 是有效的

数据。因此，在仅需最基本的操作的情况下，无需另外定义 BinomialHeap 类，可以使代码减少不必要的细节，更为简洁。

各种接口：

```
inline void init();
// 定义二项堆前必须调用 init() 初始化哨兵节点

inline BinomialHeap binomialHeapUnion(BinomialHeap a, BinomialHeap b);
// 合并 a、b 两个堆（不进行任何拷贝操作）并返回新堆

inline void binomialHeapInsert(BinomialHeap& h, int val);
// 向堆 h 中插入值 val
// * 由于可能修改当前堆的结构，传入引用

inline void decreaseKey(BinomialHeap h, node* key, int val);
// 将堆 h 中的 key 节点键值改为 val
// 使用 assert 保证 val 必须不大于原值，否则程序退出

inline int extractMin(BinomialHeap &h);
// 弹出最小值
// * 由于可能修改当前堆的结构，传入引用

inline void binomialHeapErase(BinomialHeap& h, node *key);
// 从 h 中删除 key 节点
// * 由于可能修改当前堆的结构，传入引用
```

为了代码简洁起见，并没有采取封装设计，堆合并更为简洁。

三、算法设计

大致思路采取第三版 CLRS 上二项堆章节的讲解。

首先有一个重要的操作 binomialLink(a,b) 表示把 b 设为 a 的父亲，且此时 a 一定是 b 度数最大的儿子（每次只会将度数相邻的节点进行 binomialLink），在代码里实现为 node 类的成员函数：

```
inline node* binomialLink(node *x) {
```

```
    if(x == null) return null;
    x->fa = this;
    x->sib = ch;
    ch = x;
    ++deg;
    return this;
}
```

- 合并操作 binomialHeapUnion
  - mergeHeap(a, b)：将两个二项堆 a、b 的二项树构成的森林进行简单的合并，确保合并后的链上度数递增。

    此时对于任意度数，最多有两棵树具有相同的度数。

    由于二项树的节点度数维持在 $O(lgn)$ 级别，该操作时间复杂度为 $O(lgn)$。
  - binomialHeapUnion(a, b)：用户需要调用的合并堆操作。

    首先利用 mergeHeap 拉出一条链，考虑链上的二项树度数递增，如果相邻的两棵树度数已经不同，那么不需要处理。如果相邻的两棵树度数相同，则利用之前的 binomialLink，将键值较小的节点设为父亲即可，既维护了堆性质，又维护了二项堆不能有度数相同的二项树性质。具体实现上，只需维护当前节点的前一个、后一个节点（prev_x、next_x），然后比较度数，分类讨论即可。时间复杂度 $O(lgn)$。
- 插入操作 binomialHeapInsert

  使用合并操作构造即可。

```
inline void binomialHeapInsert(BinomialHeap& h, int val) {
    BinomialHeap n = new node; // 新建节点
    n->val = new int(val); // 新建数据域
    h = binomialHeapUnion(h, n);
}
```

- 减小键值 decreaseKey

  实际上就是从某个节点开始，不断跳父亲指针，如果当前节点的键值小于父亲的，那么交换指向数据的指针（之所以不交换节点本身的指针，是由于每个节点都有父亲指针，如果修改了树的结构，那么复杂度退化为 $O(lg^2n)$；不交换数据是因为在维护的数据较大时，交换指针可以保证运行效率）。

- 提取最小值 `extractMin`

  首先需要遍历所有二项树的根，找到最小的一个。接下来删掉这颗树的根，将树根的儿子拉成一条链，将其他所有二项树根拉成另一条链，再使用 `binomialHeapUnion` 操作，将两条链构成的二项树森林合并度数相同的节点，即维持了二项堆的性质。

- 删除节点 `binomialHeapErase`

  只需利用 `decreaseKey` 和 `extractMin` 构造全局最小值然后弹出即可。

```
inline void binomialHeapErase(BinomialHeap& h, node *key) {
    decreaseKey(h, key, -2147483648);
    extractMin(h);
}
```

四、测试 & 可视化

对于需要使用节点指针作为参数的几项操作，由于不便可视化，使用 gdb 输出节点地址，然后直接输入指针地址进行测试。

对于基本的堆插入/弹出操作，使用 `GraphViz` 处理，每次操作后实时展示二项树森林的结构，可以确保堆合并/插入/弹出等操作的正确性。而其他几项操作又主要依赖合并操作，因此，在确定了合并/弹出最小值两个操作的正确性后，其他操作的正确性也就有了保障。

可视化的具体步骤是，首先造了一组简单的操作序列，含有若干插入/弹出操作。每次操作后，调用 `dotify()` 生成`.dot` 文件，接下来使用 `GraphViz` 的实时预览插件，即可做到每次操作后生成结果，例如下图：

对于另一组数据，包含了一些插入和弹出操作，制作成动图，为文件夹内的 `Result.gif`。

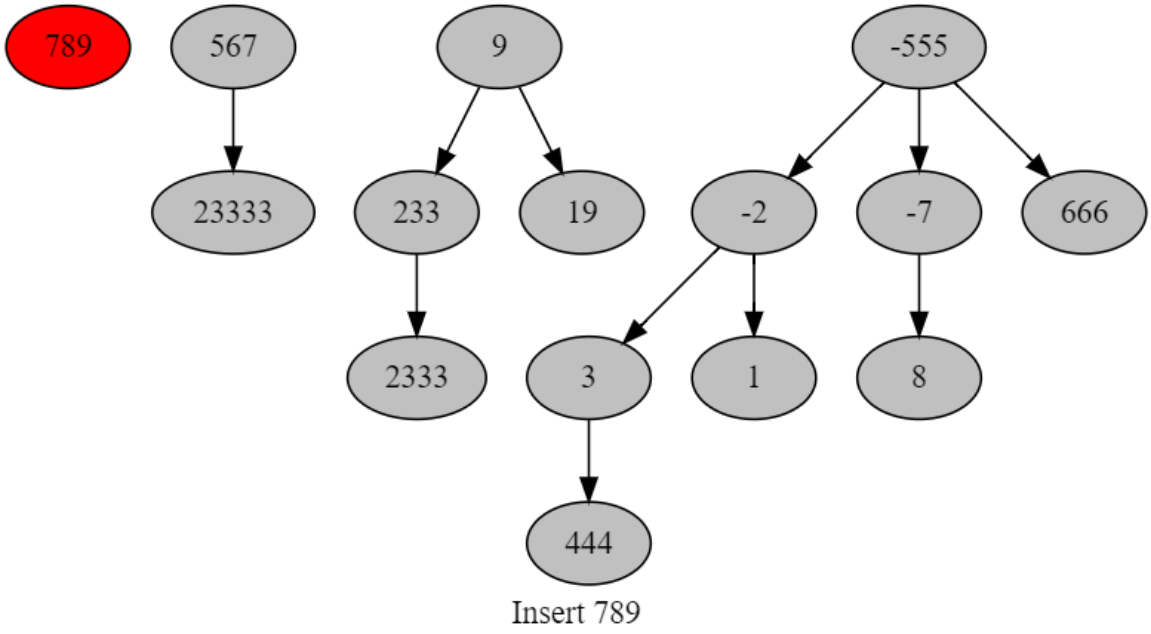

图 1: 可视化结果示例