10.优先级队列

（a）基本实现

邓俊辉

deng@tsinghua.edu.cn

优先级队列

❖ 还记得Huffman编码吗？反复地从森林中，取出权值最小的两棵树，合二为一，再插回其中

❖
```
template <typename T> struct PQ { //priority queue
    virtual void insert(T) = 0; //按照优先级次序插入词条
    virtual T getMax() = 0; //取出优先级最高的词条
    virtual T delMax() = 0; //删除优先级最高的词条
}; //与其说PQ是数据结构，不如说是ADT；其不同的实现方式，效率及适用场合也各不相同
```

❖ Stack和Queue，都是PQ的特例——优先级完全取决于元素的插入次序

Steap和Queap，也是PQ的特例——插入和删除的位置受限

❖ 更一般情况下，优先级如何确定？

应用、算法与特点

- 应用　离散事件模拟

  操作系统：任务调度、中断处理、GUI的MRU、...

  输入法：词频调整

- 作为底层数据结构所支持的高效操作，是很多高效算法的基础

  内部、外部、在线排序

  贪心算法：Huffman编码、Kruskal算法

  平面扫描算法中的事件队列

  ...

- 全序？偏序！

  元素之间或者不能直接比较大小，或者不能低成本地进行比较

基于向量

- 通常，insert()操作远多于delMax()操作
- 故相对而言，采用无序向量反而更为有效 //适用于哪些场合？

	无序向量	（非降）有序向量
insert()	$\Theta(1)$ 直接作为末元素插入	$O(n)$ 二分搜索确定插入位置 $O(\log n)$ 最坏时需移动所有元素 $O(n)$
delMax()	$\Theta(n)$ 遍历以确定最大元素 $\Theta(n)$ 最坏时需移动所有元素 $O(n)$	$\Theta(1)$ 直接摘除末元素

基于列表

- 同样地，通常insert()操作远多于delMax()操作
- 故相对而言，采用无序列表反而更有效
- 能否兼顾两种列表的优点？

	无序列表	有序列表
insert()	Θ(1) 直接作为首元素插入	O(n) 顺序搜索确定插入位置 O(n) 插入新元素 Θ(1)
delMax()	Θ(n) 遍历以确定最大元 Θ(n) 摘除最大元 Θ(1)	Θ(1) 直接摘除首元素

基于BBST

- AVL、Splay、Red-black、...

  insert()和delMax()都仅需$O(\log n)$时间

- 但是，BBST的功能**远远超出**了优先级队列的要求

  比如，这里并不要求提供**search()**接口

  另外，这里也不要求维护所有元素之间的**全序**关系

- 因此，或许有结构**更为简单**、维护**成本更低**的方法

  使得两个基本接口的 渐进时间复杂度依然为$O(\log n)$，而且

  实际效率更高

- 就最坏情况而言，这类结构已属最优——为什么？

统一测试

```
template <typename PQ, typename T> void testHeap( int n ) {
    T* elem = new T[ n/3 ]; //创建由n/3个随机元素组成的数组
    for ( int i = 0; i < n/3; i++ ) elem[i] = dice( (T) 3 * n );
    PQ heap( elem, n/3 ); delete [] elem; //Robert Floyd
    while ( heap.size() < n ) //随机测试
       if ( dice(100) < 70 ) heap.insert( dice( (T) 3*n ) ); //70%概率插入
       else if ( ! heap.empty() ) heap.delMax(); //30%概率删除
    while ( ! heap.empty() ) heap.delMax(); //清空
}
```