队列的应用

循环队列

为了最大化利用空间，解决假溢出问题
理解模运算：% n，对于小于n的数没影响，到n时，一取模，变回0，这也对应着数组的第一位是0。循环就是为了走到尽头时能回头，如果回头发现有空，就能利用。
初始化

```
front=rear=0;
```

队空

```
front==rear;
```

队满：为了和队空区别，规定rear指向front的前一个位置时队满

```
(rear+1)%maxSize==front;
```

入队后

```
rear=(rear+1)%maxSize;
```

出队后

```
front=(front+1)%maxSize;
```

遍历循环队列

```
for(int i=front;i!=rear;i=(i+1)%maxSize);
```

二叉树的层序遍历

```
void LevelPrint(TreeNode<T>* tree) {
                Queue<TreeNode<T>*> s;
                s.EnQueue(tree);
                while (!s.isEmpty()) {
                        s.DeQueue(tree);
                        cout << tree->data;
                        if (tree->leftChild != nullptr)s.EnQueue(tree->leftChild);
                        if (tree->rightChild != nullptr)s.EnQueue(tree->rightChild);
                }
        }
```

堆实现优先级队列(代码以最小堆为例)

优先级队列：

- 与队列相同：队首出队，队尾入队
- 与队列不同：优先级最大/小的先出队，并非先进先出
- 出入队流程：出队/入队—>调整
- 一般实现：按优先级从大到小/从小到大在队中排列
- 一般实现的问题：入队最坏情况要遍历整个队列，效率不高

为什么用堆

优先级队列不需要全部有序（只关注出队的是不是min或max），只是局部有序就能满足需求，且这样做效率高

前置

- 堆序性：最大堆的任结点key>=左右孩子key；最小堆的任结点key<=左右孩子key
- 为了将树存储在数组中，要求堆必须是完全二叉树
- 完全二叉树中，结点序号i，则父节点序号(i-1)/2，左孩子序号2*i*+1，右孩子序号2i+2
- 为了与数组存储相适应，规定整颗树的根节点序号为0

堆的调整算法

- 向下调整siftDown
  从父节点开始向下调整将它和它的孩子调整成堆

1. 终点为每个子树的最后一个节点(序号>整颗树的最后一个结点就结束)
2. 父节点只需与两孩子里大/小的进行比较(取决于是最大堆还是最小堆)
3. 比较后，满足堆序性，调整完毕，退出函数；不满足堆序性，交换位置

```
void siftDown(int start,int PosLastNode) {
                int PosFather = start;
                int PosChild = 2 * start + 1;
                T fatherKey = heap[PosFather];//保存父节点的key值
                while(PosChild <= PosLastNode) {
                        if (PosChild<PosLastNode && heap[PosChild]>heap[PosChild + 1])
                                PosChild++;//选左右孩子里小的那个和父节点比较
                        if (heap[PosFather] <= heap[PosChild])break;//父节点小，不用调整位置，直接
                        if (heap[PosFather] > heap[PosChild]) {
                                //父节点key大，则与孩子交换位置
                                heap[PosFather] = heap[PosChild];
                                heap[PosChild] = fatherKey;
                                //更新父节点和子节点位置，更新父节点key值，将下一个子树调整为最小堆
                                PosFather = 2 * PosFather + 1;
                                PosChild = 2 * PosFather + 1;
                                fatherKey = heap[PosFather];
                        }
                }
        }
```

- 向上调整siftDown
  从子结点开始向上调整将它和它的父结点调整成堆

1. 终点为整颗树的根节点
2. 比较后，满足堆序性，调整完毕，退出函数；不满足堆序性，交换位置

```
void siftUp(int start) {
                int PosChild = start;
                int PosFather = (start - 1) / 2;
                T childKey = heap[PosChild];
                while (PosFather > 0) {
                        if (heap[PosChild] >= heap[PosFather])break;//key值比父大，不用调整
                        if (heap[PosChild] < heap[PosFather]) {
                                //子节点key值比父小，位置交换
                                heap[PosChild] = heap[PosFather];
                                heap[PosFather] = childKey;
                                //更新子节点和父节点位置，更新子节点key值
                                PosChild = PosFather;
                                PosFather = (PosChild - 1) / 2;
                                childKey = heap[PosChild];
                        }
                }
        }
```

创建堆

详细看这篇文章

- 向下调整创建堆
  前提是根结点的左右子树均为小堆或大堆，而数组是乱序的，无法直接从根结点开始向下调整，所以从倒数第一个非叶结点开始向下调整，从下往上调

```
void createBySiftDown(T* arr, int arrSize) {
                //省略了将arr数组复制给heap数组的操作
                int start = (currentSize - 2) / 2;
                int PosLastNode = currentSize - 1;
                while (start >= 0) {
                        siftDown(start, PosLastNode);
                        start--;
                }
        }
```

- 向上调整创建堆
  前提是要调整的节点前面已经是一个合法的堆，所以需要我们从数组的第二个元素开始（也可以理解为第二层的第一个节点)

```
void createBySiftUp(T* arr, int arrSize) {
                //省略了将arr数组复制给heap数组的操作
                int start = 1;
                int PosLastNode = currentSize - 1;
                while (start<= PosLastNode) {
                        siftUp(start);
                        start++;
                }
        }
```

出入队（由于已经是堆了，调整时比创建堆简单）

注意：以下代码没有自动扩容，极有可能在入队时超出容量，也没有报错提示，为了严谨还得继续优化
入队

```
void EnMinHeap(T insert) {
                heap[currentSize++] = insert;
                int start = currentSize - 1;
                siftUp(start);
        }
```

出队
注意：为了保证根节点删除后还是完全二叉树，因此将最后一个结点移到根节点的位置后，再向下调整

```
void DeMinHeap(T getMin) {
                getMin = heap[0];
                heap[0] = heap[currentSize - 1];
                currentSize--;
                siftDown(0, currentSize - 1);
        }
```