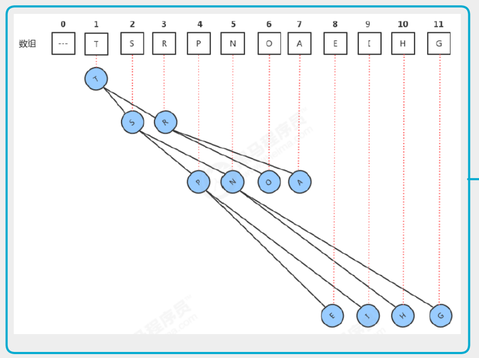
堆

定义

堆通常可以被看做是一棵完全二叉树的数组对象。

图示



API设计

- 类名
  - Heap
- 构造方法
  - Heap(int capacity)：创建容量为capacity的Heap对象
- 成员方法
  - 1.private boolean less(int i,int j)：判断堆中索引i处的元素是否小于索引j处的元素
  - 2.private void exch(int i,int j):交换堆中i索引和j索引处的值
  - 3.public T delMax():删除堆中最大的元素,并返回这个最大元素
  - 4.public void insert(T t)：往堆中插入一个元素
  - 5.private void swim(int k):使用上浮算法，使索引k处的元素能在堆中处于一个正确的位置
  - 6.private void sink(int k):使用下沉算法，使索引k处的元素能在堆中处于一个正确的位置
- 成员变量
  - 1.private T[] imtes : 用来存储元素的数组
  - 2.private int N：记录堆中元素的个数

特性

1.它是完全二叉树，除了树的最后一层结点不需要是满的，其它的每一层从左到右都是满的，如果最后一层结点不是满的，那么要求左满右不满。

2.它通常用数组来实现。
具体方法就是将二叉树的结点按照层级顺序放入数组中，根结点在位置1，它的子结点在位置2和3，而子结点的子结点则分别在位置4,5,6和7，以此类推。
如果一个结点的位置为k，则它的父结点的位置为[k/2],而它的两个子结点的位置则分别为2k和2k+1。这样，在不使用指针的情况下，我们也可以通过计算数组的索引在树中上下移动：从a[k]向上一层，就令k等于k/2,向下一层就令k等于2k或2k+1。

3.每个结点都大于等于它的两个子结点。这里要注意堆中仅仅规定了每个结点大于等于它的两个子结点，但这两个子结点的顺序并没有做规定，跟我们之前学习的二叉查找树是有区别的。

实现

swim上浮算法

```
/**
 * 使用上浮算法，使索引k处的元素能在堆中处于一个正确的位置
 */
private void swim(int k)
{
    // 通过循环，不断的比较当前结点的值和其父结点的值，
    // 如果发现父结点的值比当前结点的值小，则交换位置
    while(k>1){
        //比较当前结点和其父结点
        if (less( i: k/2,k))
            exch( i: k/2,k);
        k = k/2;
    }
}
```

sink下沉算法

```
private void sink(int k){
    // 通过循环不断的对比当前k结点和其左子结点2*k以及右子结点2k+1处
    // 中的较大值的元素大小，如果当前结点小，则需要交换位置
    while( 2*k <= N ){
        int max;
        if (2*k+1<=N) {
            if (less( i: 2*k, j: 2*k+1))
                max=2*k+1;
            else
                max=2*k;
        } else{
            max = 2*k;
        }
        if (!less(k,max)) break;
        exch(k,max);
        k = max;
    }
}
```

insert插入方法

```
//往堆中插入一个元素
public void insert(T t){
    items[++N]=t;
    swim(N);
}
```

delMax删除最大元素方法

```
/**
 * 删除堆中最大的元素,并返回这个最大元素
 */
public T delMax(){
    T max = items[1];
    // 交换索引1处的元素和最大索引处的元素，
    // 让完全二叉树中最右侧的元素变为临时根结点
    exch( i: 1,N);
    //最大索引处的元素删除掉
    items[N]=null;
    //元素个数-1
    N--;
    //通过下沉调整堆，让堆重新有序
    sink( k: 1);
    return max;
}
```

堆排序

实现步骤

1.构造堆；

2.得到堆顶元素，这个值就是最大值；

3.交换堆顶元素和数组中的最后一个元素，此时所有元素中的最大元素已经放到合适的位置；

4.对堆进行调整，重新让除了最后一个元素的剩余元素中的最大值放到堆顶；

5.重复2~4这个步骤，直到堆中剩一个元素为止。

API设计

- 类名
  - HeapSort
- 成员方法
  - 1.public static void sort(Comparable[] source)：对source数组中的数据从小到大排序
  - 2.private static void createHeap(Comparable[] source, Comparable[] heap):根据原数组source，构造出堆heap
  - 3.private static boolean less(Comparable[] heap, int i, int j)：判断heap堆中索引i处的元素是否小于索引j处的元素
  - 4.private static void exch(Comparable[] heap, int i, int j):交换heap堆中i索引和j索引处的值
  - 5.private static void sink(Comparable[] heap, int target, int range):
    在heap堆中，对target处的元素做下沉，范围是0~range。

堆构造过程

创建一个新数组，把原数组0~length-1的数据拷贝到新数组的1~length

再从新数组长度的一半处开始往1索引处扫描（从右往左）

对扫描到的每一个元素做下沉调整

```
/**
 * 根据原数组source，构造出堆heap
 */
private static void createHeap(Comparable[] source, Comparable[] heap) {
    //把source中的元素拷贝到heap中，heap中的元素就形成一个无序的堆
    System.arraycopy(source, srcPos: 0,heap, destPos: 1,source.length);
    //对堆中的元素做下沉调整(从长度的一半处开始，往索引1处扫描)
    for (int i = (heap.length)/2; i>0; i--){
        sink(heap,i, range: heap.length-1);
    }
}
```

堆排序过程

1.将堆顶元素和堆中最后一个元素交换位置；

2.通过对堆顶元素下沉调整堆，把最大的元素放到堆顶(此时最后一个元素不参与堆的调整，因为最大的数据已经到了数组的最右边)

3.重复1~2步骤，直到堆中剩最后一个元素。

```
/**
 * 对source数组中的数据从小到大排序
 */
public static void sort(Comparable[] source) {
    Comparable[] heap = new Comparable[source.length+1];
    createHeap(source,heap);
    int N = heap.length-1;
    while(N!=1){
        exch(heap, i: 1,N);
        sink(heap, target: 1, --N);
    }
    System.arraycopy(heap, srcPos: 1,source, destPos: 0,source.length);
}
```