链滴

数据结构与算法 - 堆

作者：amoslam

原文链接：https://ld246.com/article/1578155128332

来源网站：链滴

概述

堆是一种树，由它实现的优先级队列的插入和删除的时间复杂度都是O(logn)，用堆实现的优先级队虽然和数组实现相比较删除慢了些，但插入的时间快的多了。当速度很重要且有很多插入操作时，可选择堆来实现优先级队列。

种类

堆有最大堆和最小堆两种。其名字很形象，上面的元素小，越向下元素越大，在取堆顶元素的时候总先拿到最小的，最大堆与之相反。

定义

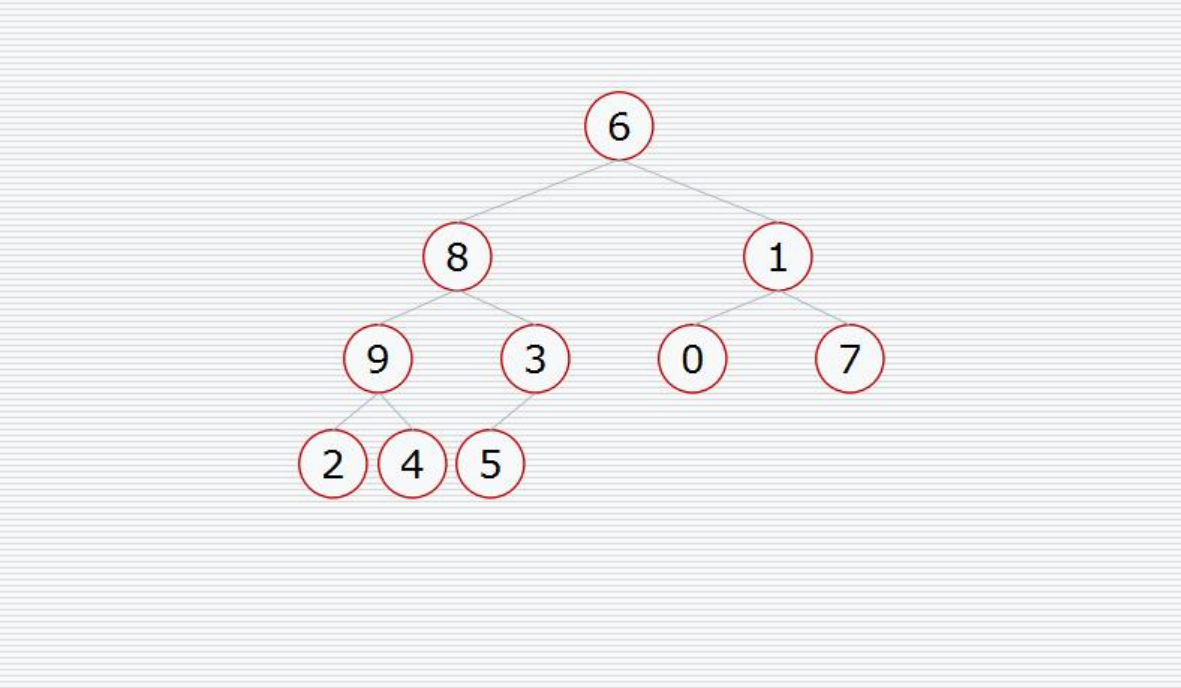
n个关键字序列array[0，...，n-1]，当且仅当满足下列要求：(0 <= i <= (n-1)/2)

① array[i] <= array[2i + 1] 且 array[i] <= array[2i + 2]； 称为小根堆；

② array[i] >= array[2i + 1] 且 array[i] >= array[2i + 2]； 称为大根堆；

动图演示

图片来源：百度



优缺点

优点

插入删除快，对最大数据项的存取很快

缺点

对其他数据项存取慢

使用场景

● 从大数量级数据中筛选出top n 条数据； 比如：从几十亿条订单日志中筛选出金额靠前的1000条据

● 优先队列(用堆实现的数据结构)；很多语言中，都提供了优先级队列的实现，比如，Java 的 Priorit Queue

代码

1.大顶堆

```
/**
 * ================================
 * 作者：amos lam
 * 时间：2020年1月5日上午12:21:45
 * 备注：数据结构与算法 - 堆之大顶堆
 * ================================
 */
public class HeapSort {

    // 构建大根堆：将array看成完全二叉树的顺序存储结构
    private int[] buildMaxHeap(int[] array) {

        // 从最后一个节点array.length-1的父节点（array.length-1-1）/2开始，直到根节点0，反复整堆
        for (int i = (array.length - 2) / 2; i >= 0; i--) {

            adjustDownToUp(array, i, array.length);
        }
        return array;
    }

    // 将元素array[k]自下往上逐步调整树形结构
    private void adjustDownToUp(int[] array, int k, int length) {

        int temp = array[k];
        for (int i = 2 * k + 1; i < length - 1; i = 2 * i + 1) { // i为初始化为节点k的左孩子，沿节点较的子节点向下调整

            if (i < length && array[i] < array[i + 1]) { // 取节点较大的子节点的下标

                i++; // 如果节点的右孩子>左孩子，则取右孩子节点的下标
            }
            if (temp >= array[i]) { // 根节点 >=左右子女中关键字较大者，调整结束

                break;
            } else { // 根节点 <左右子女中关键字较大者

                array[k] = array[i]; // 将左右子结点中较大值array[i]调整到双亲节点上
```

```
            k = i; // 【关键】修改k值，以便继续向下调整
        }
    }
    array[k] = temp; // 被调整的结点的值放人最终位置
}

// 堆排序
public int[] heapSort(int[] array) {

    array = buildMaxHeap(array); // 初始建堆，array[0]为第一趟值最大的元素
    for (int i = array.length - 1; i > 1; i--) {

        int temp = array[0]; // 将堆顶元素和堆低元素交换，即得到当前最大元素正确的排序位置
        array[0] = array[i];
        array[i] = temp;
        adjustDownToUp(array, 0, i); // 整理，将剩余的元素整理成堆
    }
    return array;
}

// 删除堆顶元素操作
public int[] deleteMax(int[] array) {
    // 将堆的最后一个元素与堆顶元素交换，堆底元素值设为-99999
    array[0] = array[array.length - 1];
    array[array.length - 1] = -99999;
    // 对此时的根节点进行向下调整
    adjustDownToUp(array, 0, array.length);
    return array;
}

// 插入操作:向大根堆array中插入数据data
public int[] insertData(int[] array, int data) {

    array[array.length - 1] = data; // 将新节点放在堆的末端
    int k = array.length - 1; // 需要调整的节点
    int parent = (k - 1) / 2; // 双亲节点
    while (parent >= 0 && data > array[parent]) {

        array[k] = array[parent]; // 双亲节点下调
        k = parent;
        if (parent != 0) {

            parent = (parent - 1) / 2; // 继续向上比较
        } else { // 根节点已调整完毕，跳出循环
            break;
        }
    }
    array[k] = data; // 将插入的结点放到正确的位置
    return array;
}

public void toString(int[] array) {

    for (int i : array) {
```

```
            System.out.print(i + " ");
        }
    }

    public static void main(String args[]) {

        HeapSort hs = new HeapSort();

        int[] array = { 87, 45, 78, 32, 17, 65, 53, 9, 122 };

        System.out.print("构建大根堆：");
        hs.toString(hs.buildMaxHeap(array));
        System.out.print("\n" + "删除堆顶元素：");
        hs.toString(hs.deleteMax(array));
        System.out.print("\n" + "插入元素63:");
        hs.toString(hs.insertData(array, 63));
        System.out.print("\n" + "大根堆排序：");
        hs.toString(hs.heapSort(array));
    }
}
```

2.小顶堆

后续完善.....