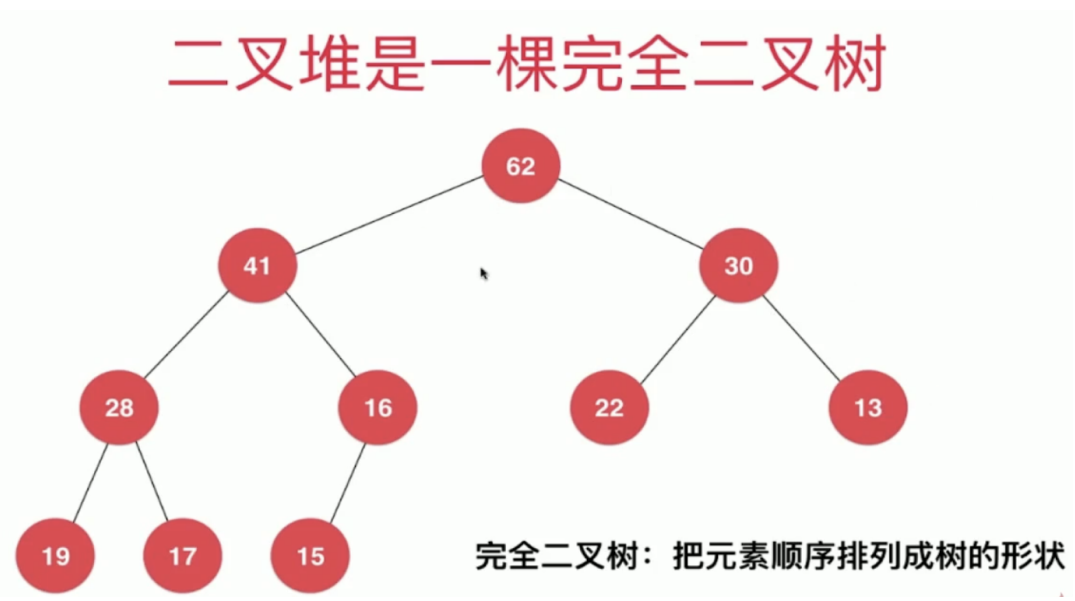
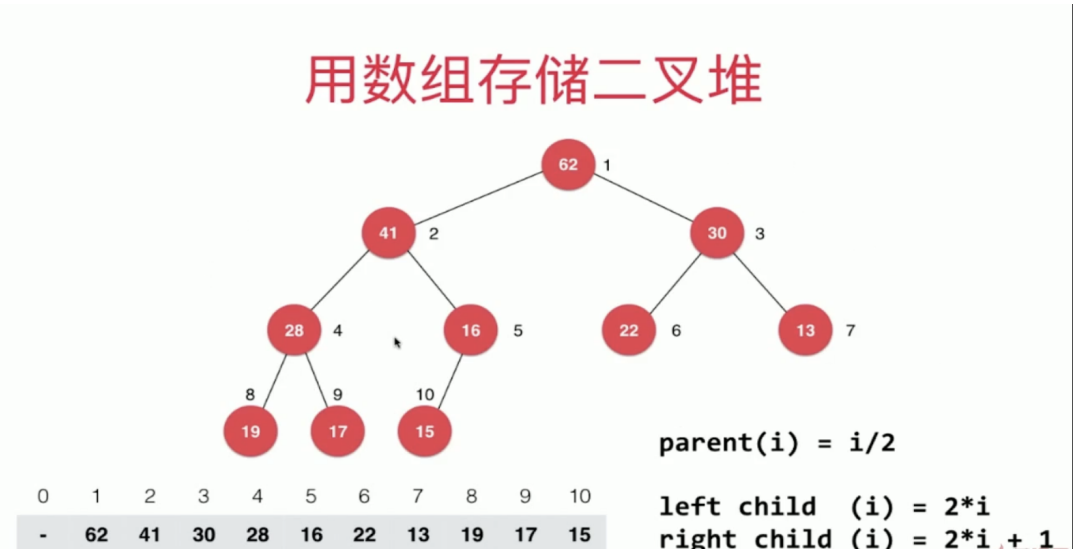
数据结构与算法9-优先队列、二叉堆

笔记本： 我的笔记
创建时间： 2020/10/4 14:47 **更新时间：** 2020/10/4 15:06
作者： liuhouer
标签： 算法



优先队列

	入队	出队（拿出最大元素）
普通线性结构	O(1)	O(n)
顺序线性结构	O(n)	O(1)
堆	O(logn)	O(logn)



二叉堆的性质：

堆中某个节点的值总是不大于其父节点的值
二叉堆是一棵完全二叉树[完全二叉树:把元素顺序排列成树的形状]

1>数组实现最大堆【二叉堆】【堆顶元素最大】【jdk提供的优先队列是最小堆】

```
public class Array<E> {


    private E[] data;
    private int size;


    // 构造函数，传入数组的容量capacity构造Array
    public Array(int capacity){
        data = (E[])new Object[capacity];
        size = 0;
    }


    // 无参数的构造函数，默认数组的容量capacity=10
    public Array(){
        this(10);
    }


    public Array(E[] arr){
        data = (E[])new Object[arr.length];
        for(int i = 0 ; i < arr.length ; i ++)
            data[i] = arr[i];
        size = arr.length;
    }


    // 获取数组的容量
    public int getCapacity(){
        return data.length;
    }


    // 获取数组中的元素个数
    public int getSize(){
        return size;
    }


    // 返回数组是否为空
    public boolean isEmpty(){
        return size == 0;
    }
```

```
    // 在index索引的位置插入一个新元素e
    public void add(int index, E e){


        if(index < 0 || index > size)
            throw new IllegalArgumentException("Add failed. Require index >= 0
and index <= size.");


        if(size == data.length)
            resize(2 * data.length);


        for(int i = size - 1; i >= index ; i --)
            data[i + 1] = data[i];


        data[index] = e;


        size ++;
    }


    // 向所有元素后添加一个新元素
    public void addLast(E e){
        add(size, e);
    }


    // 在所有元素前添加一个新元素
    public void addFirst(E e){
        add(0, e);
    }


    // 获取index索引位置的元素
    public E get(int index){
        if(index < 0 || index >= size)
            throw new IllegalArgumentException("Get failed. Index is
illegal.");
        return data[index];
    }


    // 修改index索引位置的元素为e
    public void set(int index, E e){
        if(index < 0 || index >= size)
            throw new IllegalArgumentException("Set failed. Index is
illegal.");
        data[index] = e;
    }


    // 查找数组中是否有元素e
    public boolean contains(E e){
        for(int i = 0 ; i < size ; i ++){
            if(data[i].equals(e))
                return true;
        }
        return false;
    }


    // 查找数组中元素e所在的索引，如果不存在元素e，则返回-1
    public int find(E e){
        for(int i = 0 ; i < size ; i ++){
            if(data[i].equals(e))
                return i;
        }
        return -1;
    }
```

```
    // 从数组中删除index位置的元素, 返回删除的元素
    public E remove(int index){
        if(index < 0 || index >= size)
            throw new IllegalArgumentException("Remove failed. Index is
illegal.");


        E ret = data[index];
        for(int i = index + 1 ; i < size ; i ++)
            data[i - 1] = data[i];
        size --;
        data[size] = null; // loitering objects != memory leak


        if(size == data.length / 4 && data.length / 2 != 0)
            resize(data.length / 2);
        return ret;
    }


    // 从数组中删除第一个元素, 返回删除的元素
    public E removeFirst(){
        return remove(0);
    }


    // 从数组中删除最后一个元素, 返回删除的元素
    public E removeLast(){
        return remove(size - 1);
    }


    // 从数组中删除元素e
    public void removeElement(E e){
        int index = find(e);
        if(index != -1)
            remove(index);
    }


    public void swap(int i, int j){


        if(i < 0 || i >= size || j < 0 || j >= size)
            throw new IllegalArgumentException("Index is illegal.");


        E t = data[i];
        data[i] = data[j];
        data[j] = t;
    }


    @Override
    public String toString(){


        StringBuilder res = new StringBuilder();
        res.append(String.format("Array: size = %d , capacity = %d\n", size,
data.length));
        res.append('[');
        for(int i = 0 ; i < size ; i ++){
            res.append(data[i]);
            if(i != size - 1)
                res.append(", ");
        }
        res.append(']');
        return res.toString();
    }


    // 将数组空间的容量变成newCapacity大小
    private void resize(int newCapacity){
```

```
        E[] newData = (E[])new Object[newCapacity];
        for(int i = 0 ; i < size ; i ++)
            newData[i] = data[i];
        data = newData;
    }
}
```

```
public class MaxHeap<E extends Comparable<E>> {


    private Array<E> data;


    public MaxHeap(int capacity){
        data = new Array<>(capacity);
    }


    public MaxHeap(){
        data = new Array<>();
    }


    public MaxHeap(E[] arr){
        data = new Array<>(arr);
        for(int i = parent(arr.length - 1) ; i >= 0 ; i --)
            siftDown(i);
    }


    // 返回堆中的元素个数
    public int size(){
        return data.getSize();
    }


    // 返回一个布尔值, 表示堆中是否为空
    public boolean isEmpty(){
        return data.isEmpty();
    }


    // 返回完全二叉树的数组表示中, 一个索引所表示的元素的父亲节点的索引
    private int parent(int index){
        if(index == 0)
            throw new IllegalArgumentException("index-0 doesn't have
parent.");
        return (index - 1) / 2;
    }


    // 返回完全二叉树的数组表示中, 一个索引所表示的元素的左孩子节点的索引
    private int leftChild(int index){
        return index * 2 + 1;
    }


    // 返回完全二叉树的数组表示中, 一个索引所表示的元素的右孩子节点的索引
    private int rightChild(int index){
        return index * 2 + 2;
    }


    // 向堆中添加元素
    public void add(E e){
        data.addLast(e);
        siftUp(data.getSize() - 1);
    }


    private void siftUp(int k){
```

```
        while(k > 0 && data.get(parent(k)).compareTo(data.get(k)) < 0 ){
            data.swap(k, parent(k));
            k = parent(k);
        }
    }


    // 看堆中的最大元素
    public E findMax(){
        if(data.getSize() == 0)
            throw new IllegalArgumentException("Can not findMax when heap is
empty.");
        return data.get(0);
    }


    // 取出堆中最大元素
    public E extractMax(){


        E ret = findMax();


        data.swap(0, data.getSize() - 1);
        data.removeLast();
        siftDown(0);


        return ret;
    }


    private void siftDown(int k){


        while(leftChild(k) < data.getSize()){
            int j = leftChild(k); // 在此轮循环中,data[k]和data[j]交换位置
            if( j + 1 < data.getSize() &&
                    data.get(j + 1).compareTo(data.get(j)) > 0 )
                j ++;
            // data[j] 是 leftChild 和 rightChild 中的最大值


            if(data.get(k).compareTo(data.get(j)) >= 0 )
                break;


            data.swap(k, j);
            k = j;
        }
    }


    // 取出堆中的最大元素，并且替换成元素e
    public E replace(E e){


        E ret = findMax();
        data.set(0, e);
        siftDown(0);
        return ret;
    }
}
```

2>实现优先队列

定义队列

```
public interface Queue<E> {


    int getSize();
    boolean isEmpty();
    void enqueue(E e);
    E dequeue();
```

```
    E getFront();
}
```

利用二叉堆实现优先队列

```
public class PriorityQueue<E extends Comparable<E>> implements Queue<E> {


    private MaxHeap<E> maxHeap;


    public PriorityQueue(){
        maxHeap = new MaxHeap<>();
    }


    @Override
    public int getSize(){
        return maxHeap.size();
    }


    @Override
    public boolean isEmpty(){
        return maxHeap.isEmpty();
    }


    @Override
    public E getFront(){
        return maxHeap.findMax();
    }


    @Override
    public void enqueue(E e){
        maxHeap.add(e);
    }


    @Override
    public E dequeue(){
        return maxHeap.extractMax();
    }
}
```

优先队列的经典问题

在1,000,000个元素中选出前100名?
在N个元素中选出前M个元素
排序? NlogN
使用优先队列? NlogM