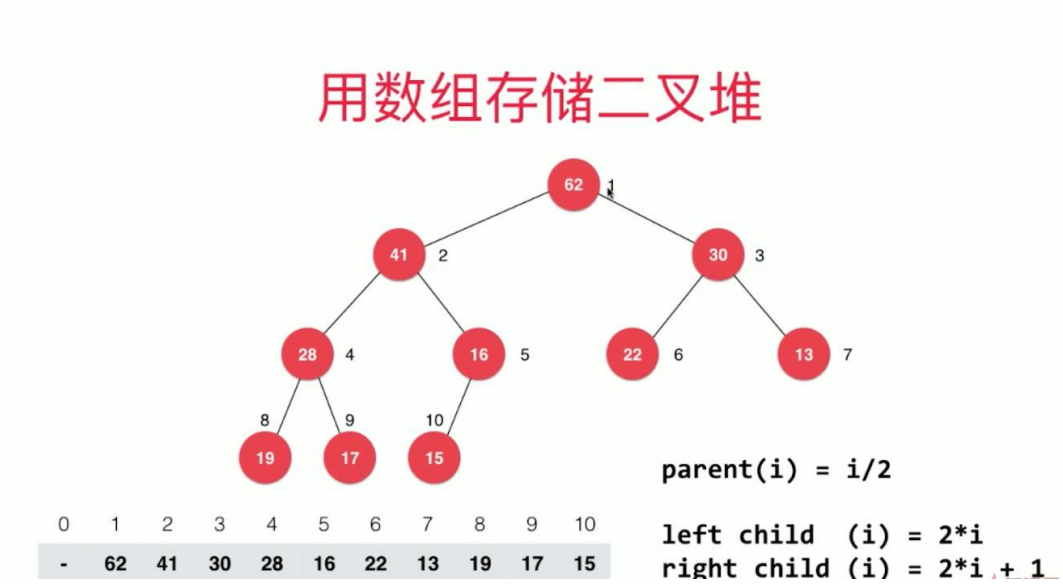
优先队列

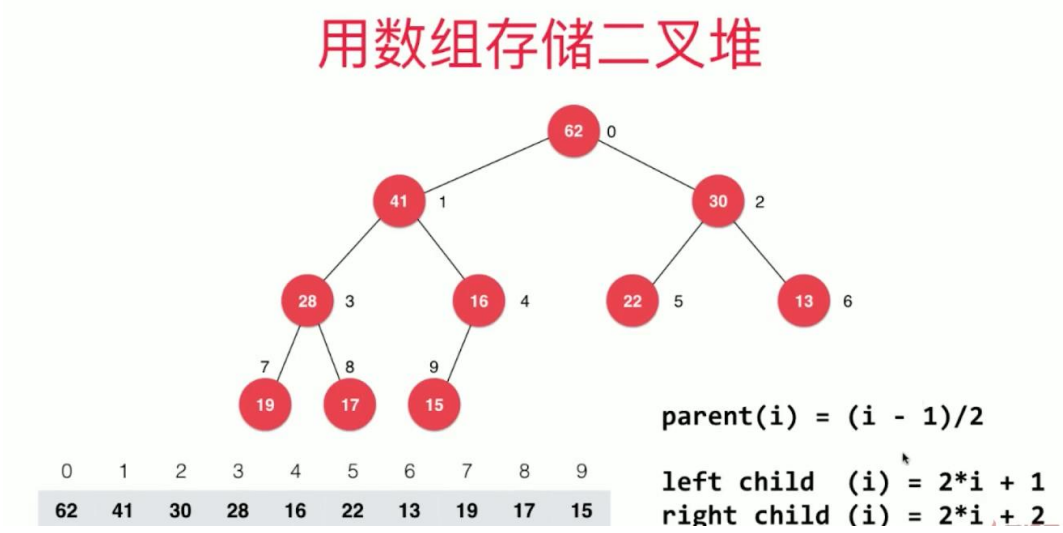
- 什么是优先队列
  - 普通队列先进先出
  - 优先队列在出队列时需要考虑元素的优先级而与入队的顺序无关
- 为什么使用优先队列
  - 需要**动态**的处理任务
  - 例如cpu调度，游戏中的自动攻击时选择优先级高的目标

大顶堆（最大堆）

- 大顶堆一般采用完全二叉树的形式以便于我们使用数组来存储数据（也可以不是完全二叉树）
- 大顶堆需要满足根节点的值比左右孩子节点的值都要大，但是左右孩子之间没有明确的规定
- 大顶堆的存储
  - 借助于完全二叉树的性质，我们一般采用数组来存储堆。对于不同的数组起始位置，父节点与子节点之间有不同的对应规则。
  - 数组从1开始存储



  - 数组从0开始存储



- 大顶堆需要动态数组，可以使用自定义的Array，也可以使用java提供的ArrayList实现

```
//非手动实现
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.appenshixand(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;
    }
}
```

- 基本定义`MaxHeap`

  此处的定义从数组`0`坐标位置开始存储数据，因此，对于节点`i`而言

  父节点：(i - 1) / 2

  左孩子：2 * i + 1

  右孩子：2 * i + 2

```
public class MaxHeap<E extends Comparable<E>>{
    //private int size; Array中维护了size
    private Array<E> data;

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

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

    public int size(){
        return data.size()
    }

    public boolean isEmpty(){
        return data.isEmpty();
    }

    //寻找某一元素在堆中的位置
    public int getIndex(E e){
        for(int i=0;i<data.size();i++){
            if(data[i].equals(e))
                return i;
        }
        return -1;
    }

    //堆中是否包含某一元素
    public boolean contains(E e){
        return getIndex(e)==-1;
    }

    //寻找节点的父节点的索引值
    public int parent(int index){
        if(index == 0)
            throw new IllegalArgumentException("index-0 doesn't have
parent.");
        if(index < 0)
            throw new IllegalArgumentException("index must greater than
0.");
        return (index-1) / 2;
    }

    //寻找左孩子节点
    public int leftChild(int index){
        if(index < 0)
```

```
            throw new IllegalArgumentException("index must be greater than
0.");
        if(index > data.size()-1)
            throw new IllegalArgumentException("index must be samller than
data`s size                        .");
        return (2*index+1)>(data.size()-1)?-1:(2*index+1);
    }

    //寻找右孩子节点
    public int rightChild(int index){
        if(index < 0)
            throw new IllegalArgumentException("index must be greater than
0.");
        if(index > data.size()-1)
            throw new IllegalArgumentException("index must be samller than
data`s size                        .");
        return (2*index+2)>(data.size()-1)?-1:(2*index+2)
    }
}
```

- 向堆中添加元素，取出堆中最大元素，看堆中最大元素
  - 向堆中添加元素

    当我们向堆中添加元素时，新加入的元素可能比其父节点的值要大，从而破坏堆的性质，因此我们需要调整当前的堆，使其再次满足最大堆的相关要求。

    1. 当新增加一个元素时，我们先将其放置在数组的末尾
    2. 如果元素没有父节点，接插入结束，否则继续
    3. 将元素与其父节点比较大小，如果其父节点的值大于该节点的值，则插入过程结束
    4. 如果其父节点的值小于该节点的值，则将父节点与该节点互换
    5. 重复上述步骤

    上述过程我们可以称之为 `shiftUp`

    ```
    private void shiftUp(int index){
        //index > 0 可以控制节点是否有父节点
        if(index > 0 && data.get(parent(index)).compareTo(data.get(index)) <
    0){
            int p = parent(index);
            data.swap(p,index);
            index = p;
        }
    }

    public void add(E e){
        data.addLast(e);
        shiftUp(data.size()-1);
    }
    ```

  - 看堆中的最大值（不删除堆中的最大值点）

```
public E peekMax(){
    if(data.size() == 0)
       throw new IllegalArgumentException("Can not findMax when heap is
empty.");
    return data.get(0);
}
```

- 取出堆中的最大值（删除堆中的最大值点，需要对堆重新排列）。在对堆重新排序时我们可以采取的思路为：

  1. 保存堆中的最大值，并作为返回值
  2. 将位于堆尾的元素放置到堆首，并与其左右孩子中的最大值进行比较
  3. 如果该元素小于最大值，则将其与最大值进行互换
  4. 重复上述步骤，直到该元素大于其左右孩子或者该元素不再存在左孩子

  上述过程是一个自上而下的过程，我们可以称之为`shiftDown`

```
public E extractMax(){
    if(data.size() == 0)
       throw new IllegalArgumentException("Can not findMax when heap is
empty.");
    E max = data.get(0);
    data.swap(0,data.size()-1);
    data.removeLast();
    shiftDown(0);
    return max;
}

private void shiftDown(int index){
    while(leftChild(index) < data.size()){
        int l = leftChild(index);
        //如果有右孩子,且右孩子比左孩子大
        if( l+1 < data.size() && data.get(l).compareTo(data.get(l+1))<0
)
            l++;
        //节点的值大于其左右孩子中的最大值
        if(data.get(l).compareTo(data.get(index))<= 0)
            return;
        data.swap(l,index);
        index = l;
    }
}
```

- 取出堆中的最大元素，并替换成元素 e

```
public E replace(E e){
    E max = data.get(0);
    data.set(0,e);
    shiftDown(0);
    return max;
}
```

- 向堆中添加元素

```
public void add(E e){
    data.addLast(e);
    shiftUp(data.size()-1);
}
```

- `heapify`：将给定的一个数组转换为堆的形式
  1. 对于一个给定的数组，我们可以将其转换为树的形式
  2. 对于树的所有叶子节点而言，其本身就是一个大顶堆不需要进行调整
  3. 我们需要调整的第一个节点就是最后一个叶子节点的父节点，然后将索引值递减就可以调整其他的父节点
  4. 调整过程其实就是执行`shiftDown`的过程
  5. 当调整到根节点时结束过程

```
public MaxHeap(E[] arr){
    //添加数组数据
    data = new Array(arr);
    //调整数据
    for(int i=parent(arr.length() -1 );i>=0;i--){
        shiftDown(i);
    }
}
```

- 其它堆
  - d叉堆
  - 索引堆
  - 二项堆
  - 斐波那契堆

如何实现优先队列

- 我们可以借助大顶堆或者小顶堆实现优先队列
- 优先级的定义需要我们自己完成

```
public class PriorityQueue<E extends Comparable<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.peekMax();
```

```
    }

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

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

- 使用优先队列可以解决类似在大量数据中取出前m个数的问题，类似于在100000000个树中取出前100个
- 练习题：LeetCode 347