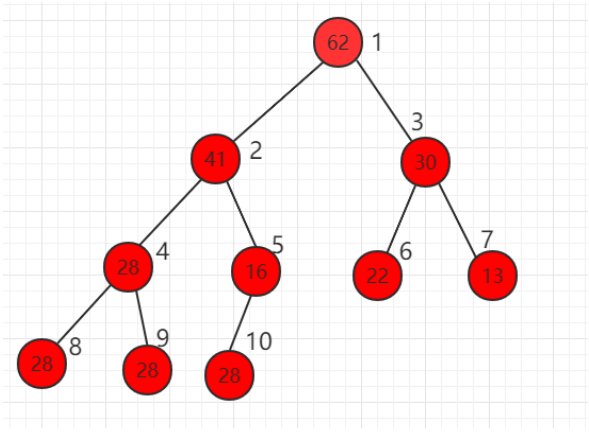
data structure

Heap

完全儿数能实现一个堆，可以用一个数组来存储：




```
parent(i) = (i-1)/2
left child(i) = 2*i+1
right child(i) = 2*i +2
```

```

// 堆初始化：O(logN)
// 如果给定N个数，需要初始化成一个堆：O(NlogN)
// 给定一个数组arr,可以从开始到结束来构建一个大根堆/小根堆

// 实现逻辑：（大根堆）
// 首先是index=0：判断根节点就是本身，不做处理
// 对于index=i：  比较和根节点的大小，如果大，就交换，然后一直向上，就把一个
// 大值不断移到了堆顶

// 如果index从0-->n-1，这样一个数组就完全成为了大根堆

void heapInsert(vector<int> &arr,int index)
{
    // heap insert algorithm
    while (arr[index]>arr[(index-1)/2]) {
        swap(arr,index,(index-1)/2);
        index=(index-1)/2;
    }
}
```

```cpp
// heapFiy操作是通过heapsize来记录堆的大小，index表示，
// 从哪个位置向下进行恢复堆的结构
// 判断index的左孩子是否存在，如果不存在，就返回，如果存在，然后
// 判断这个index是否有右孩子，如果没有就比较index和左孩子的值
// 如果有就比较左右孩子和index的值，把大值交换到index，继续往下找，知道超过heapsize

void heapFiy(vector<int> & arr,int index,int heapSize)
{
    int l = index*2+1;
    while(l<heapSize)
    {
        int largest = l+1<heapSize && arr[l+1] >arr[l] ? l+1: l;
        largest = arr[largest]>arr[index] ? largest:index;
        if(largest==index) break;
        swap(arr,index,largest);
        index = largest;
        l = index*2+1;
    }
}

```

```cpp
// 更高效的一个堆的初始化方式 O(N)

// 从最后一个节点开始，每个节点做heapfiy

for(int i=static_cast<int>(arr.size())-1;i>=0;i--)
    {
        heapFiy(arr, i,arr.size() );
    }

```

分析时间复杂度：

如果堆的大小为N，那么最下面一层的叶节点，都只需要调整一次且节点个数为N/2,倒数第二层的节点个数为N/4，需要调整的深度为2，一次类推：

$$T(N) = N/2 + N/4 * 2 + N/8 * 3 + \ldots + 1 * N \tag{1}$$

使用错位相加的方式求解的T(N)=N

堆排序