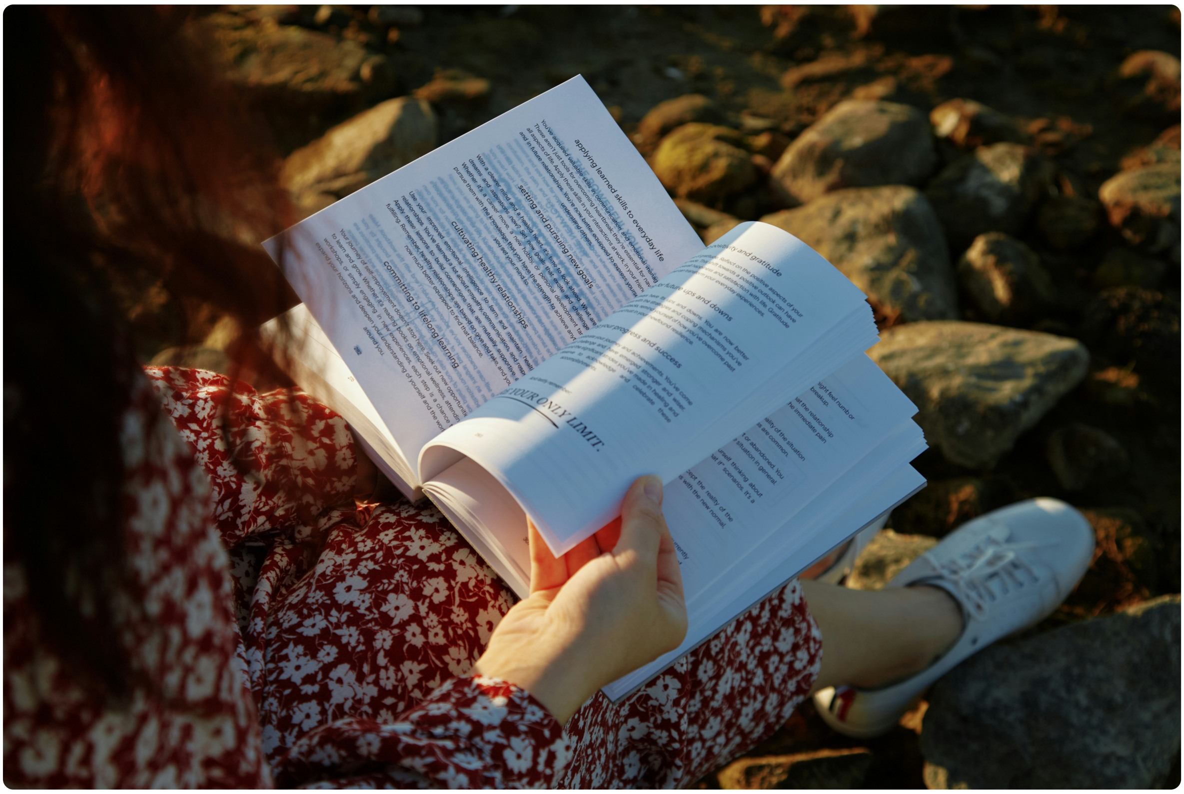
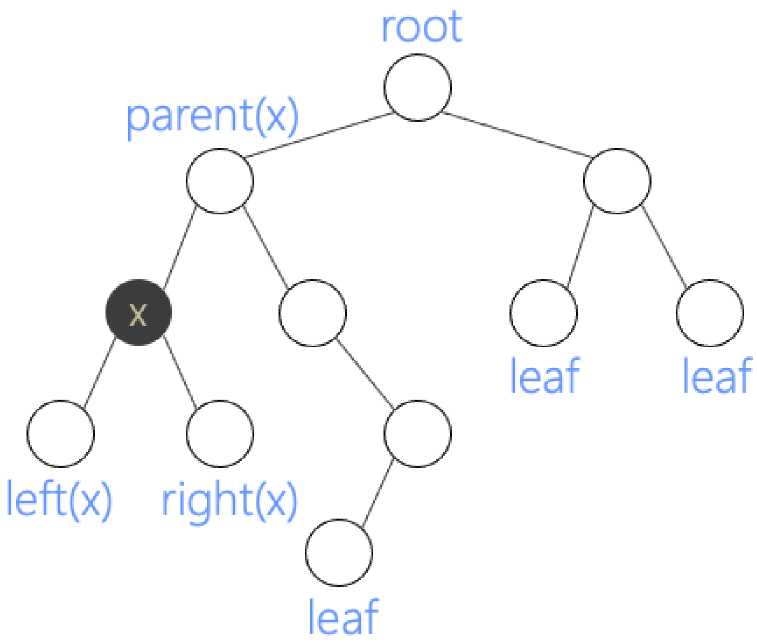


堆排序 HeapSort

在用 Linked List 或 Array 进行排序的方法中，插入的复杂度是 $O(1)$，而用于排序的复杂度是 $O(n)$，有没有一种方法能达到插入与排序的复杂度都控制在 $O(log(n))$ 呢?

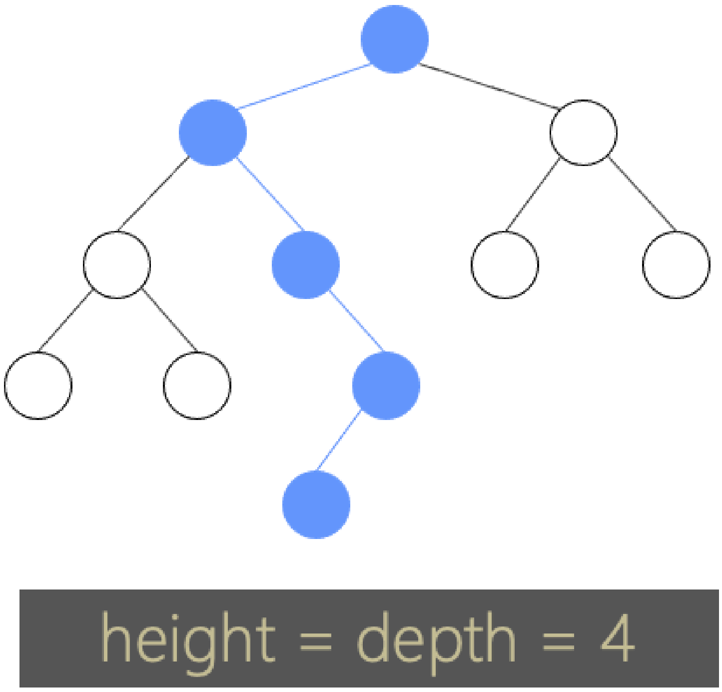
二叉树 Binary Tree

如同树木一样，**二叉树在最顶层有一个根节点，每个节点都有 0 个、1 个或 2 个子节点，没有子节点的节点称为叶节点**。对于节点 x，我们分别用 $left(x)$、$right(x)$ 和 $parent(x)$ 表示 x 的左子节点、右子节点和父节点。如图即为一个二叉树。



二叉树

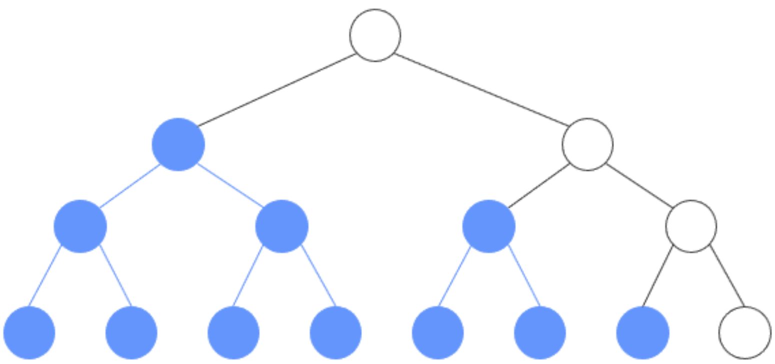
在二叉树中，树的高度 Tree Height（深度 Depth）就是从**树根到树叶的最长路径上的边数**。如图所示，这棵二叉树的深度是 4。



二叉树的深度

完美二叉树 Perfect Binary Tree

完美二叉树是指**其中一个节点可以有 0 个或 2 个子节点，并且所有叶子的深度相同，每一层高度及以上一共有** $2^{d+1}-1$ **个节点**。如图所示即为一个完美二叉树。



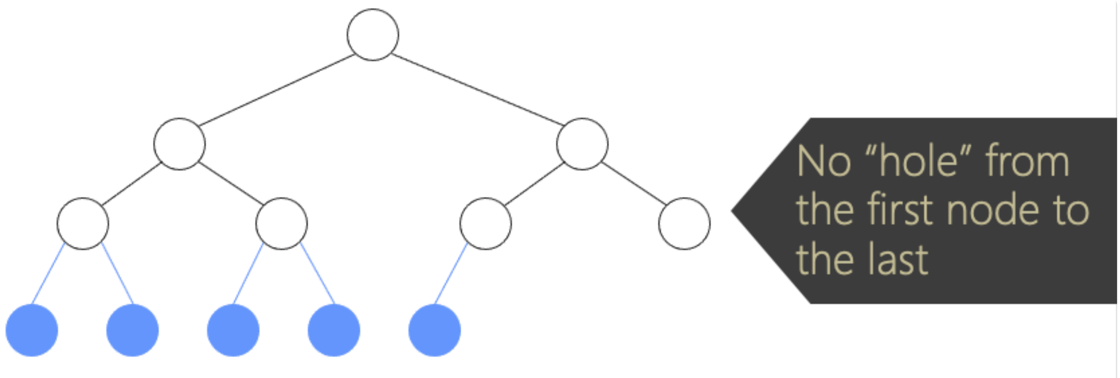
height	# of nodes
0	1
1	3
2	7
3	15
d	$2^{d+1}-1$

完美二叉树

一棵有 n **个节点**的完美二叉树的**高度为** $O(log(n))$，这意味着，如果一个算法的节点访问次数以树高为界，那么它的复杂度就是 $O(log(n))$。

二分堆 Binary Heap

堆是 "几乎完美的二叉树"，这代表着除最低层外，所有层都是满的，如果最低层不是满的，那么节点必须向左堆积。

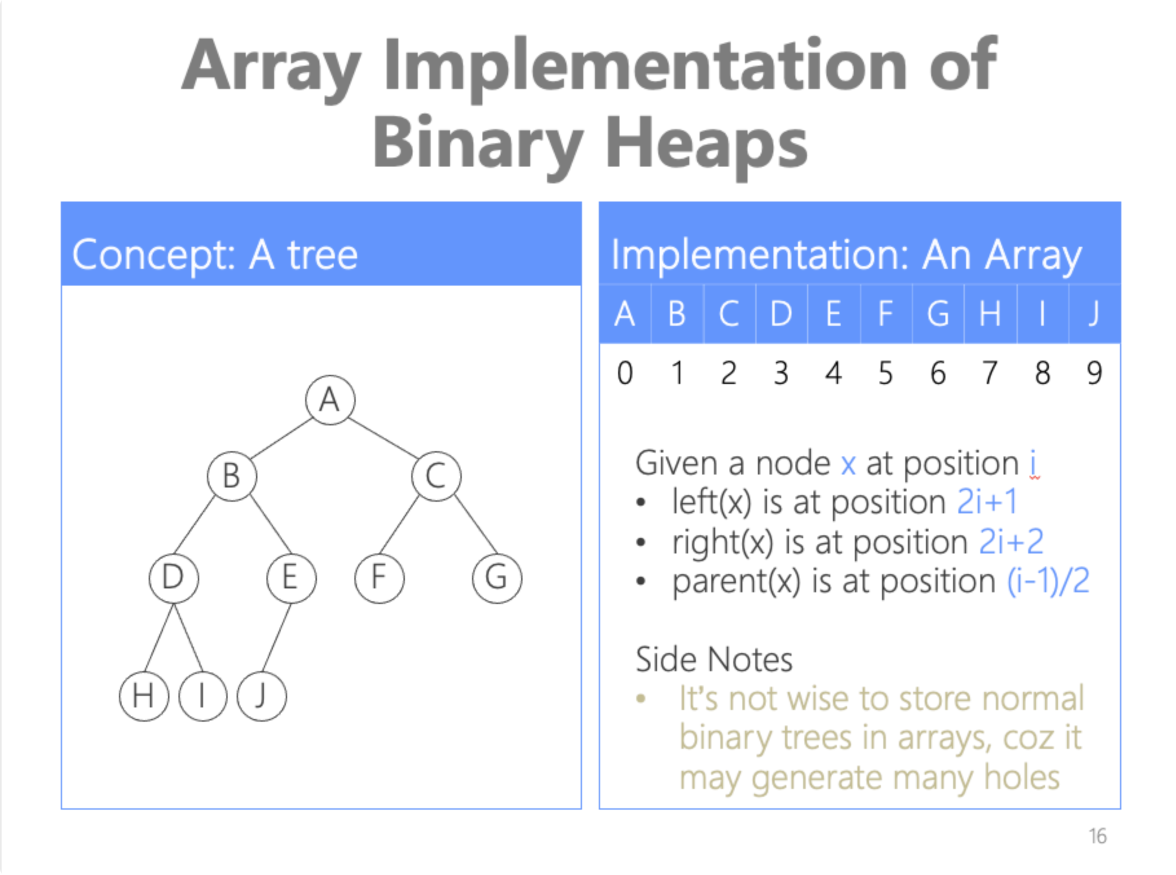


二分堆

给定一个节点数为 n、高度为 h 的二进制堆，可推出得：

- n **在** $[2^h，2^{h+1}-1]$ **范围内**
- **高度** $h = O(log(n))$

不难看出该结构非常规则，可以用数组表示，且无需任何链接。如图为一个二分堆的构建过程及每一个节点的索引。



二分堆的构建

给定在位置 i 的节点 x:

- $left(x)$ 位于位置 $2i+1$
- $right(x)$ 位于位置 $2i+2$
- $parent(x)$ 位于位置 $(i-1)/2$

堆排序

在了解完堆的特性后，很容易可以想出利用这种结构来完成排序。

最小优先级队列的属性

- 二进制堆结构
- 堆顺序属性

- 每个节点的值小于或等于其两个后代节点的值
- 最小的节点总是在最上面

二进制堆的使用在优先级队列的实现中非常普遍，因此堆一词通常被认为是数据结构的实现方式。

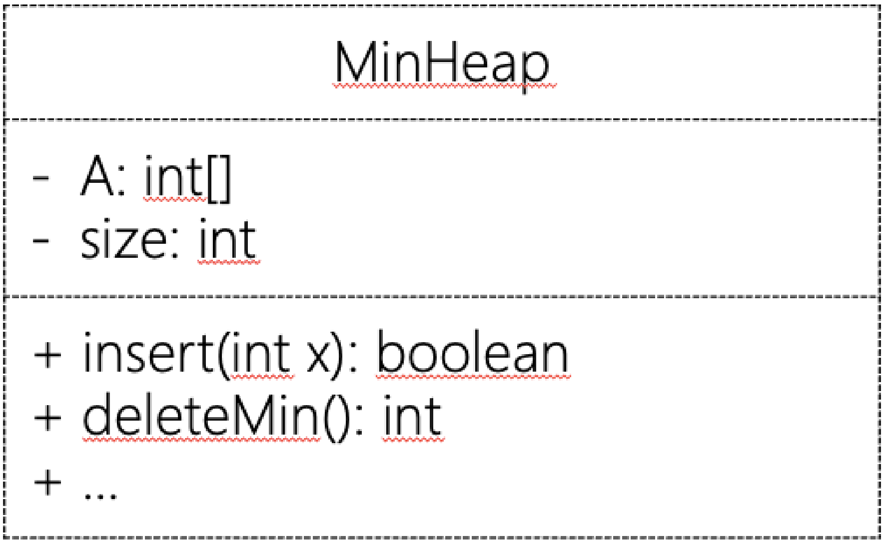
堆的属性

堆可高效支持以下操作：

- 以 $O(logN)$ 时间完成插入
- 在 $O(1)$ 时间内定位当前最小值
- 在 $O(logN)$ 时间内删除当前最小值

注意：每次插入/删除操作后，堆必须保持堆的状态

以下是堆的基本结构。



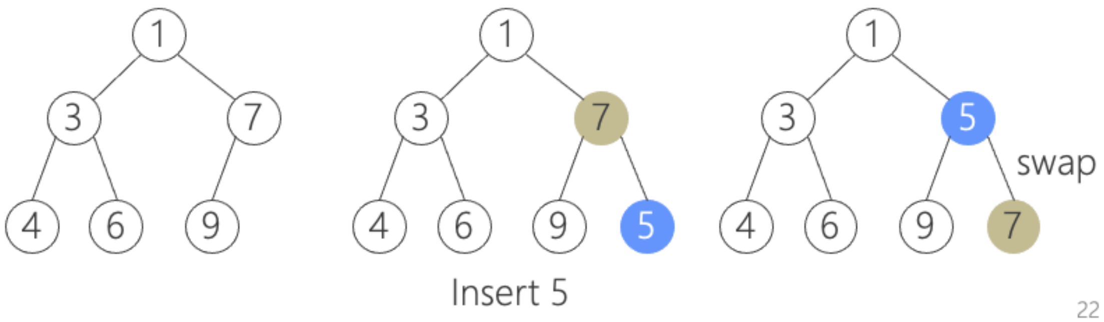
堆的结构

堆的插入

1. 将新元素添加到最低层的下一个可用位置

2. 如果违反了最小堆属性，则恢复最小堆属性

> 一般策略是向上渗透：如果元素的父元素大于子元素，则交换父元素和子元素。

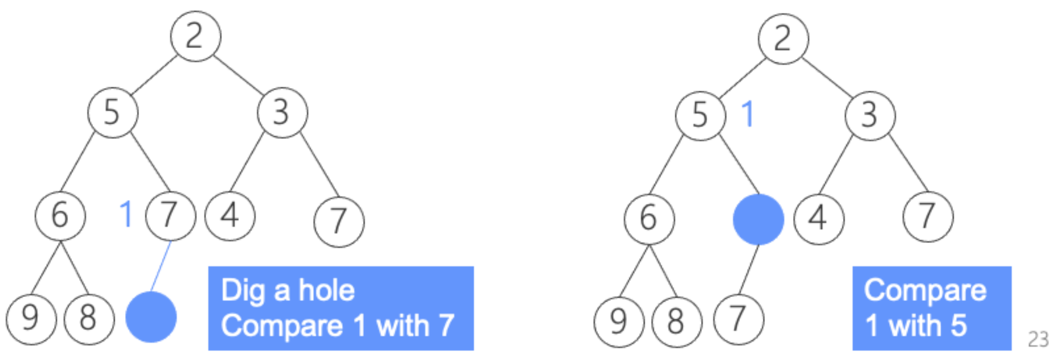


堆的插入

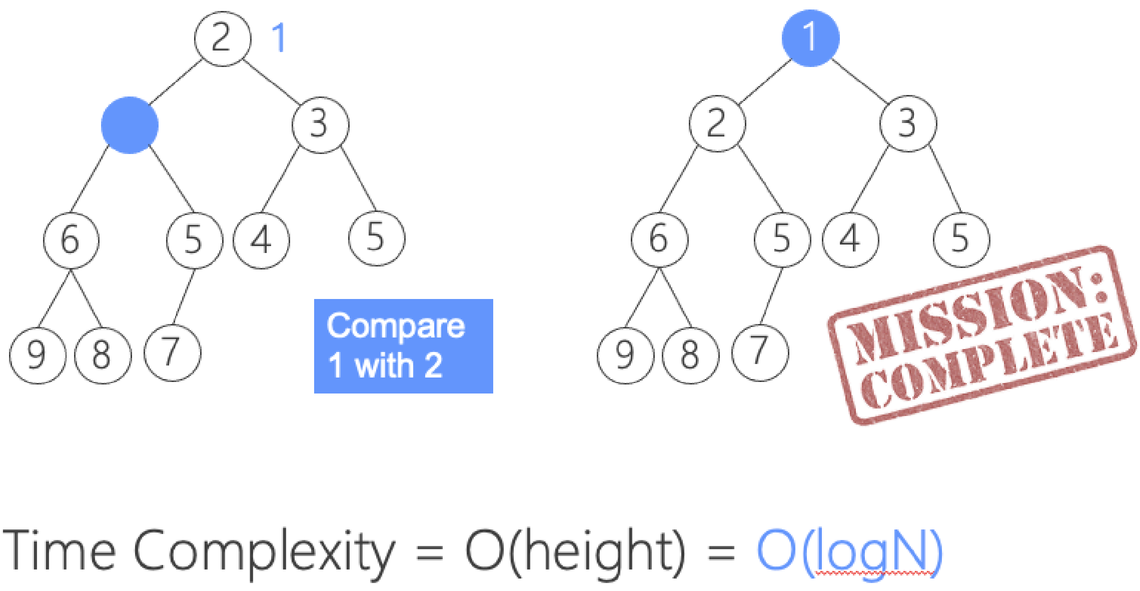
简化

如果只是进行简单的比较交换的话，那么一次交换就会有 3 个赋值语句（设立 $temp$，交换父节点，交换子节点）。这意味着如果一个元素向上渗透了 d 层，则进行 $3d$ 次赋值

可以想到的改进方案是只使用 $d+1$ 个赋值进行挖洞。如图所示，当插入 1 时，流程如下：



简化渗透



简化渗透

在这个过程中，一共只进行了 d 次父节点的交换和 1 次插入数据的赋值，因此只需要 $d+1$ 次操作。

代码实现

```
//伪代码实现
insert(x)
    IF ISFULL(A)
        return False
    // percolate up
    hole = size ++
    WHILE hole>0 AND x<A[(hole-1)/2]
        A[hole] =A[(hole-1)/2]
        hole =(hole-1)/2
    A[hole] = x
    return True
```

```java
// 代码实现
public boolean insert(int x) {
    if(size == A.length)
        return false;
    int hole = size ++;
    while(hole > 0 && x < A[(hole-1)/2]){
        A[hole] = A[(hole-1)/2];
        hole = (hole-1)/2;
    }
    A[hole] = x;
    return true;
}
```

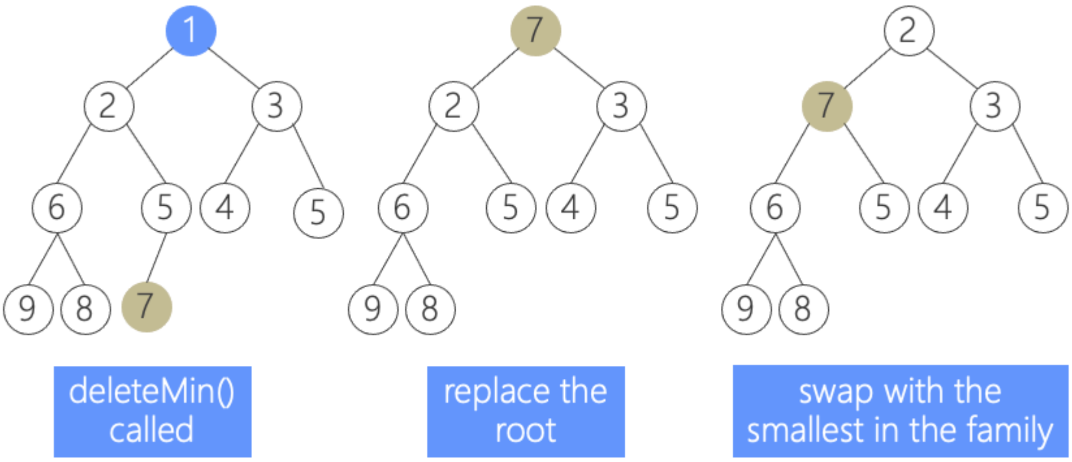
堆的排序

当插入所有的值之后，堆即处于一个有序的状态，但要让堆每一个节点的值在线性表中呈现一个有序堆状态，则需要我们每一次都从堆中依次访问目前最小的值并抽离出来，才能形成线性有序排列。

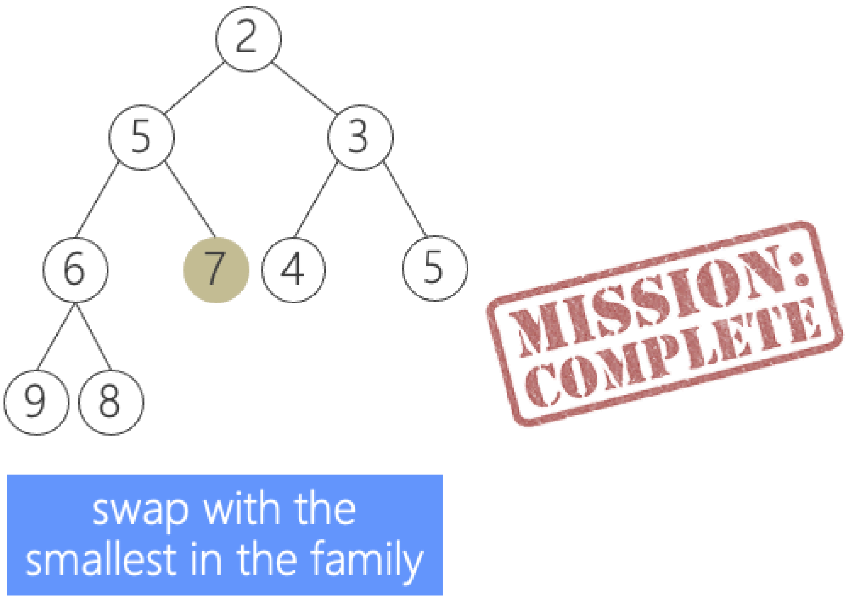
思路

1. 首先，维护二进制堆结构，即用最后一个节点的值替换根节点
2. 然后，保持堆顺序属性并向下渗透

以下是详细演示：



堆的排序



堆的排序

代码实现

```
// 伪代码实现
deleteMin()
```

```
    IF ISEMPTY(A)
        return -1
    min = A[0], hole = 0, x=A[--size]
    //percolate down
    WHILE A[hole] has children
        sid = index of A[hole]’s smaller child
        IF x⩽A[sid]
            BREAK
        A[hole] = A[sid]
    hole = sid
    A[hole] = x
    return min
```

```java
// Java实现
public int deleteMin() {
    if(size == 0)
        return -1;
    int min = A[0];
    int x = A[--size];
    int hole = 0;
    while(2*hole+1 < size) {
        int sid = 2*hole + 1;
        if(sid +1 < size && A[sid+1] < A[sid])
            sid ++;
        if(x <= A[sid])
            break;
        A[hole] = A[sid];
        hole = sid;
    }
    A[hole] = x;
    return min;
}
```

复杂度分析

1. 建立一个包含 n 个元素的二进制堆，最小元素位于堆顶

> $O(nlog(n))$

2. 执行 n 次 $DeleteMin$ 操作，按排序提取元素

> $O(nlog(n))$

3. 在第二个数组中记录这些元素，然后将数组复制回去

> $O(n)$ time
>
> $O(n)$ storage

改进方案

在上述分析中，不难看出如果要完成最终的排序则需要我们同时在最后花费 $O(n)$ 复杂度的时间与 $O(n)$ 复杂度的空间，有没有更好的方案来节约资源?

当没有额外空间

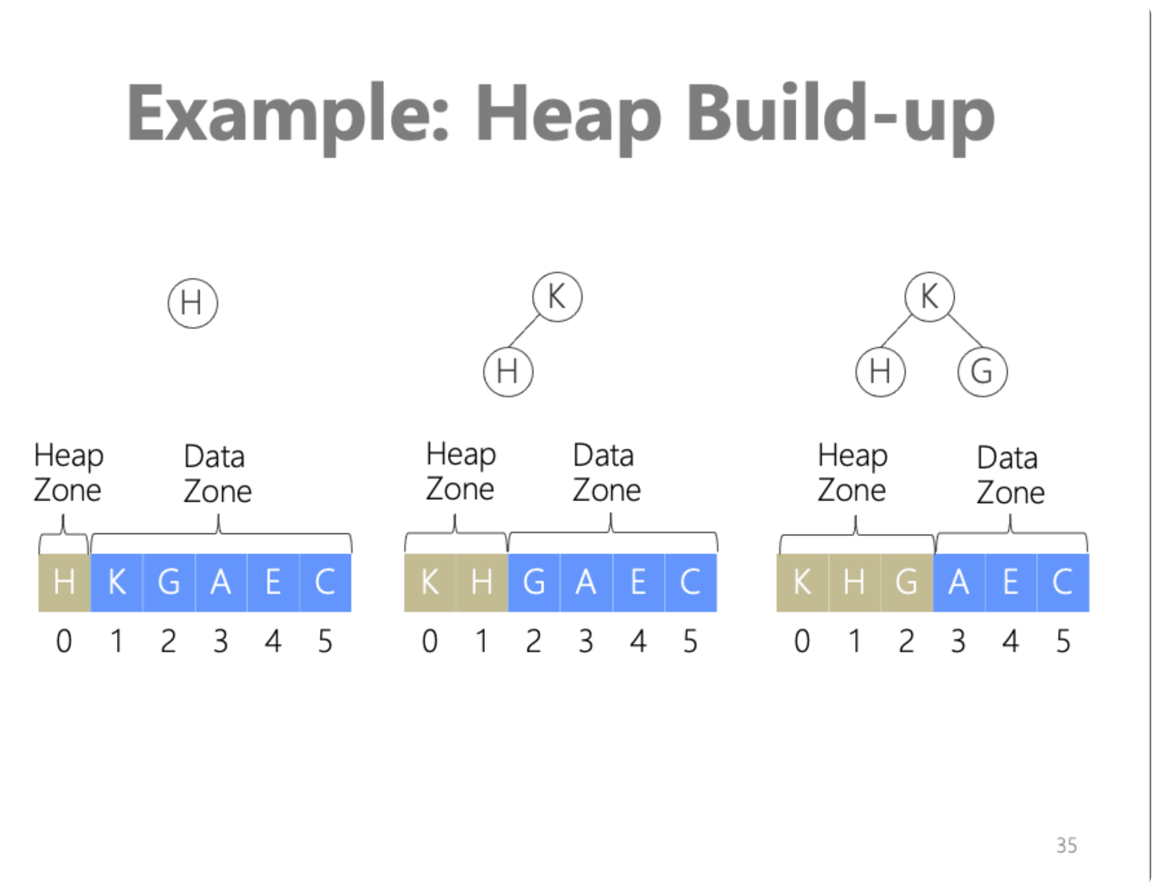
- 观察结果：每次删除最小值后，堆的大小都会缩小 1

> 我们可以使用刚刚释放的最后一个单元格来存储刚刚删除的元素，在最后一次删除最小值后，数组中的元素将按**递减顺序**排列

- 进一步观察：

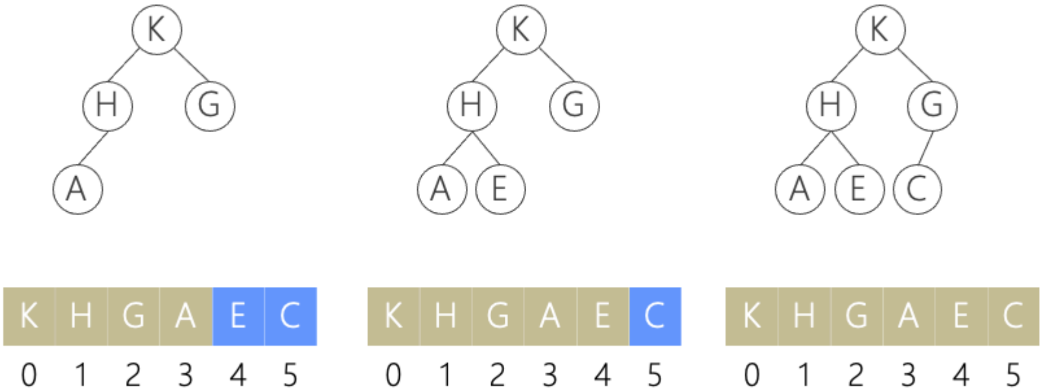
> **要按递减顺序对元素排序，使用最小堆 (min heap)**
>
> **要按递增顺序对元素排序，使用最大堆 (max heap)**
>
> **最大堆 (max heap）：父堆元素比子堆元素大**

示例如下所示：


Example: Heap Build-up
H
K
H
K
H G
Heap Zone
Data Zone
H K G A E C
0 1 2 3 4 5
Heap Zone
Data Zone
K H G A E C
0 1 2 3 4 5
Heap Zone
Data Zone
K H G A E C
0 1 2 3 4 5
35

改进堆排序

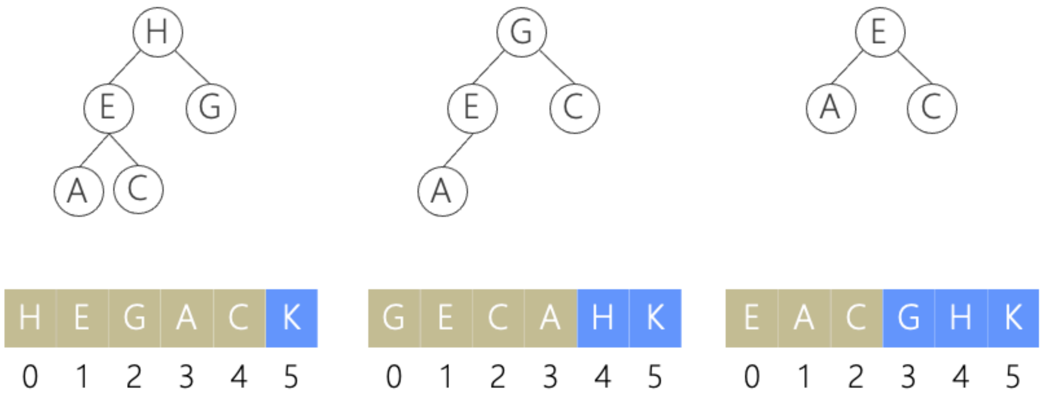
Example: Heap Build-up



36

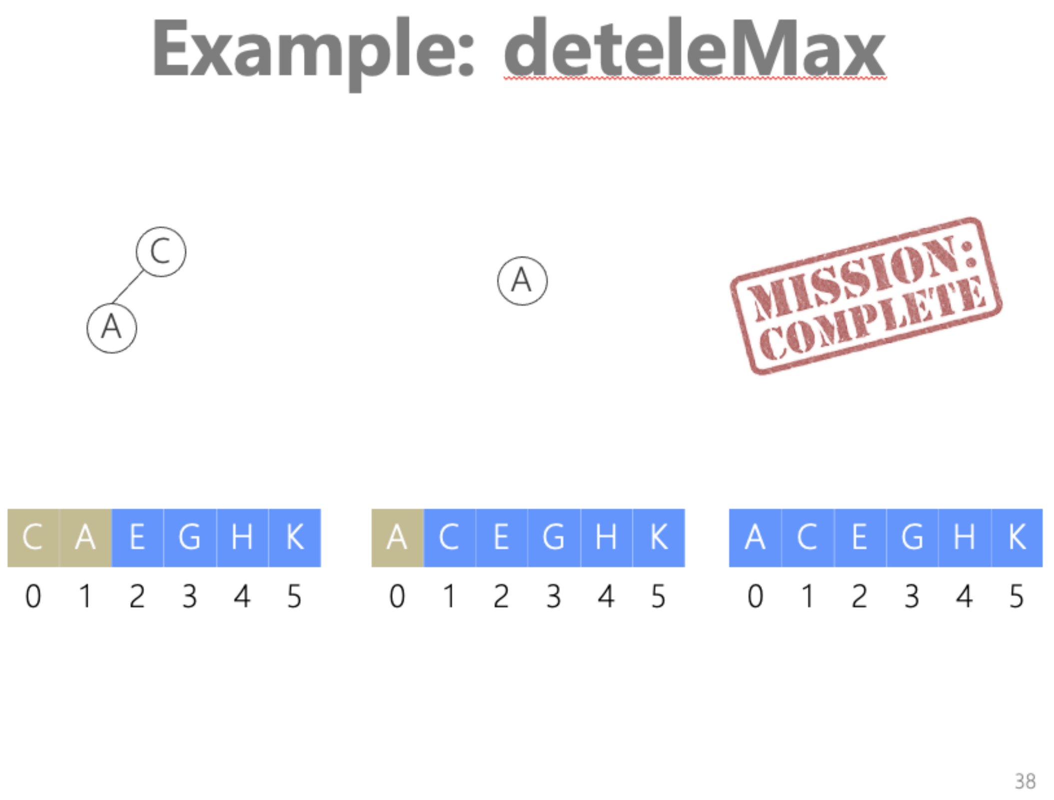
改进堆排序

Example: deteleMax



37

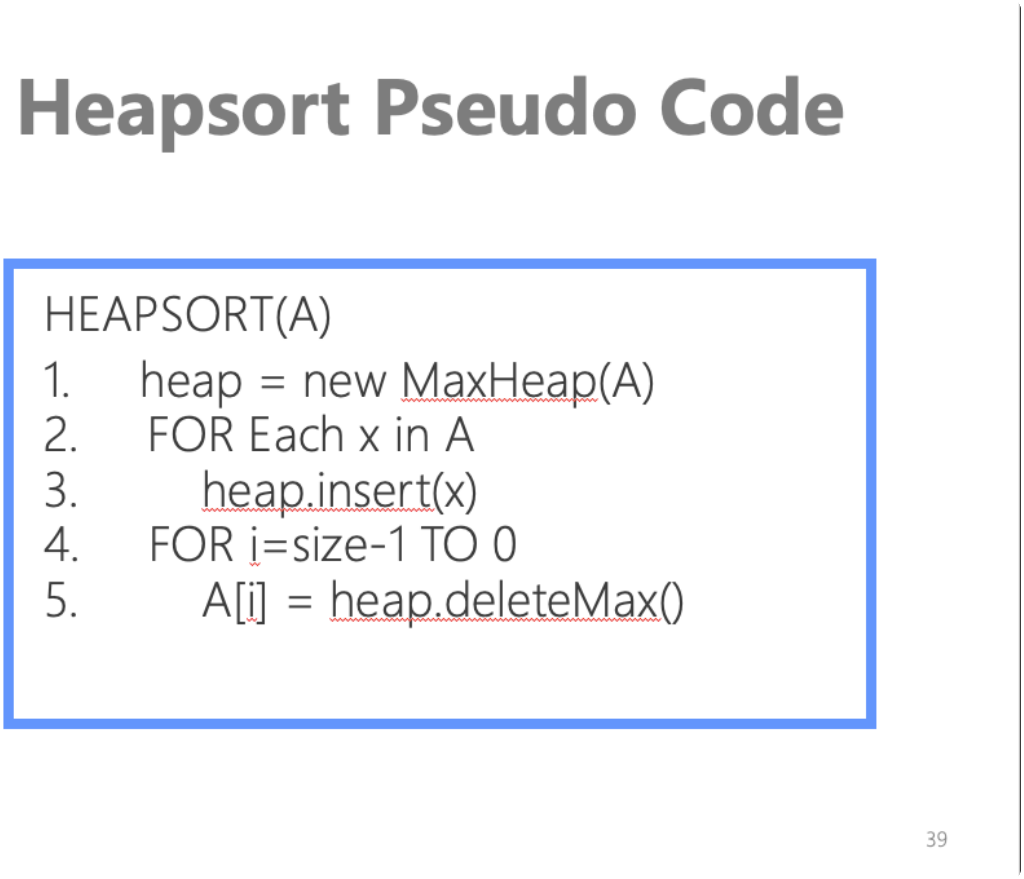
改进堆排序



改进堆排序

最大堆 Max Heap

构造结构如下：



最大堆结构

实现代码如下：

```
/*Add some comments by yourself*/
import java.util.*;

public class MaxHeap {
    int A[];
    int size;
    public MaxHeap(int A[]) {
        this.A = A;
        size = 0;
    }

    public boolean insert(int x) {
        if(size == A.length)
            return false;
```

```
        int hole = size ++;
        while(hole > 0 && x > A[(hole-1)/2]){
            A[hole] = A[(hole-1)/2];
            hole = (hole-1)/2;
        }
        A[hole] = x;
        return true;
    }

    public int deleteMax() {
        if(size == 0)
            return -1;
        int max = A[0];
        int x = A[--size];
        int hole = 0;
        while(2*hole+1 < size) {
            int sid = 2*hole + 1;
            if(sid +1 < size && A[sid+1] > A[sid])
                sid ++;
            if(x >= A[sid])
                break;
            A[hole] = A[sid];
            hole = sid;
        }
        A[hole] = x;
        return max;
    }

    public static void heapSort(int A[]) {
        MaxHeap heap = new MaxHeap(A);
        for(int x: A)
            heap.insert(x);
        for(int i=A.length-1; i>=0; i--)
            A[i] = heap.deleteMax();
    }
    public static void main(String[] args) {
        int arraySize = 100000;
        // Create a random array A1 of arraySize
```

```java
        int A[] = new int[arraySize];
        int upperbound = 100000;
        Random rand = new Random();
        for(int i=0; i<A.length; i++)
            A[i] = rand.nextInt(upperbound);
        long time1 = System.currentTimeMillis();
        heapSort(A);
        long time2 = System.currentTimeMillis();
        //System.out.println(Arrays.toString(A));
        long heapSortTime = time2 - time1;
        System.out.println("Running time for heapSort: " +
heapSortTime + "ms");
    }

}
```

至此，构建了一个不用获取额外空间且时间复杂度仅为 $O(nlog(n))$ 的排序方案。