**Binary Max Heap**

**Binary Heap Height**: If we have a Binary Heap of elements, its height will not be taller than .

-> 判断一个最大堆是否合法的条件为，判断一个内部节点的是否大于其所有（一个或者两个）子节点

-> 向最大堆中插入元素时，将元素插入到堆的末尾（即索引的最后一位），随后向上修复最大堆属性

**-> Maximum** number of **comparisons** between heap elements required to construct a max heap of N elements using the **O(n)** Build Heap:

接着对于每个节点的子节点进行递归比较，比较的次数取决于其子层数的数量，得到以下：

N=9, C=14; N=11, C=16; N=12, C=18

**-> Minimum** number of **comparisons** between heap elements required to construct a max heap of N elements using the **O(n)** Build Heap:

接着，每个需要对比的节点与其拥有的一个或两个子节点比较，不需要递归地比较。得到以下：

N=9, C=8; N=10, C=9; N=11, C=10; N=12, C=11

-> **Maximum** number of **swaps** to construct max heap of N elements using Build Heap:

接着对于每个节点和其每层的子节点进行递归交换，交换的次数取决于其子层数的数量，得到以下：

N=9, S=7; N=10, S=8; N=11, S=8; N=12, S=10

* An array A of n distinct integers that are sorted in descending order forms a valid Binary Max Heap. Assume that A[0] is not used and the array values occupy index [1:n].
* Given a Binary Max Heap, calling ShiftDown(i) ∀i > heapsize/2 will **never** change anything in the Binary Max Heap.

**HashMap**

**Load Factor:** 是一个衡量哈希表满度的指标

-> 是已经插入哈希表的元素数量。

-> 是哈希表中槽位的总数。

**Open Addressing (Linear Probing):**

-> 当发生哈希冲突时，向后寻找下一个空/已删除的槽位

-> 如果到达槽位末尾，则下一个寻找的目标为第一个槽位

-> 删除操作时，将该槽位设置为DEL，以防止在搜索时失去哈希冲突之连续性

**Primary Clustering: 主簇**是指连续的已占用槽位形成的一组，大的主簇会显著增加Hashmap操作的时间复杂度

**Closed Addressing (Separate Chaining):** 当发生碰撞时，即两个键散列到相同的索引时，冲突会通过将值存储到表外来解决(对于Separate Chaining为DLL)。

**Binary Search Tree**

-> BST(二分搜索树)中，某个节点的左侧子树中的每个节点必须小于该节点值，而右侧子树中的每个节点则必须大于该定点值

**Leaf Vertex**：没有子节点的节点。

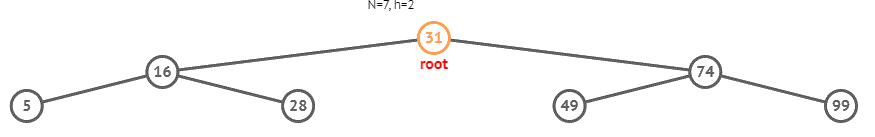
**Internal Vertex**：有子节点的节点（root节点除外）

**Insert, Search, Remove**操作都从root节点开始遍历，判断要插入/搜索/删除的值大于或小于当前节点

**Successor**: 下一个比目标节点大的值。从左往右看BST，为目标节点的右边一个节点（无论上下）

**Predecessor**: 上一个比目标节点小的值。从左往右看BST，为目标节点的左边一个节点（无论上下）

**Traversal**：分为三种方式, Inorder, Preorder, Postorder

* ****

**Inorder:**

1. 优先遍历root节点的左侧子树，在节点没有左侧和右侧字数的情况下访问该节点(5)
2. 返回**并访问**至上一个节点(16)
3. 遍历右侧的子节点，访问最底部节点(28)
4. 循环执行步骤2，3
5. 返回至root节点并访问(31)
6. 遍历root节点的右侧子树，同时优先遍历右侧中的左侧子树
7. 循环执行步骤2，3

* 访问顺序: 5 -> 16 -> 28 -> 31 -> 49 -> 74 -> 99

**Preorder:**

* 遍历顺序与inorder一致
* 但是在遍历时就访问该节点，例如从root节点开始，故root节点被第一个访问
* 访问顺序: 31 -> 16 -> 5 -> 28 -> 74 -> 49 -> 99

**Postorder:**

* 遍历顺序与inorder一致
* 但是在遍历时，只有在**当前节点没有子节点，或所有子节点都被访问过之后，才访问该节点**
* 访问顺序: 5 -> 28 -> 16 -> 49 -> 99 -> 74 -> 31
* How many structurally different BSTs can you form with **n distinct** elements?
* What is the value of the element with **rank n** in this BST?
  + 从左往右看BST，为从左往右数的第N个节点（无论上下）
* What is the **minimum** possible height of a **BST** with **N** elements?

**Sort**

**Bubble Sort:**

1. 比较相邻的元素。如果第一个比第二个大（升序排序），就交换它们。

**Selection Sort:**

1. 首先在未排序序列中找到最小（大）元素，存放到排序序列的起始位置。
2. 以此类推，直到所有元素均排序完毕。

**Insertion Sort:**

1. 从第二个元素开始向前对比，如果前一个元素比其大，则交换，对比直到第一个元素
2. 需要对比的目标指针向后移动，即第三个元素向前对比，重复第一个步骤

**Quick Sort:** , 最坏情况为

**分区操作：** 所有比基准值小的元素摆放在基准前面，所有比基准值大的元素摆在基准的后面（相同数可在任一边）。图示

描述已自动生成

**Linked List**

get(i):

search(v):

insert(i, v): 插入元素至头部，尾部，空链表的运行时间均为。插入到链表中为

remove(i): 从头部删除元素所需运行时间为。其余操作为

**Stack**

LIFO, push(), pop() are both

push(): 添加新元素至栈顶(head)

pop(): 从栈顶移除元素(head)

**Queue**

FIFO, push(), pop() are both

push(): 添加新元素至末尾(tail)

pop(): 从顶部移除元素(head)

**Postfix expression**

* push operand to stack, pop first 2 operand if an operator pushed in
* 4 1 2 9 3 / \* + 5 \* +
  + ( 9/3 \* 2 + 1 ) \* 5 + 4 = 39

**Prefix expression**

* 运算符在操作数之前，即 9/3 = /93, 9/3\*2 = \*/932
* 先算排列在操作数前的最后一个运算符
* ( 9/3 \* 2 + 1 ) \* 5 + 4 **->**  + \* + \* / 9 3 2 1 5 4

## Binary Max Heap Operations

### Insert(v)

-> 将新项 v 插入到最大二叉堆中只能在最后一个索引 N 加 1 处完成 (N+1)，以保持紧凑数组 = 完整二叉树属性。

-> 然而，最大堆属性仍然可能被违反。因此需要从插入点向上修复最大堆属性。

->-> 向上修复最大堆属性被称为ShiftUp, BubbleUp或IncreaseKey

-> 时间复杂度=

### create(A) ()

将输入数组的所有 N 个整数一一插入（即通过调用 Insert(v) 操作）到最初为空的最大二叉堆中

### create(A) ()

从数组长度的一半位置(len(A)/2)开始修复最大堆属性，递减直到第一个索引

### ExtractMax()

-> 取出最大堆中最大的数值，对于一个合法的最大堆，为root节点。

-> 将索引最后一位元素提升至root节点，并向下修复最大堆属性

-> 被称为ShiftDown, BubbleDown或Heapify，具体操作如下：

1. 将索引最后一位元素提升至root节点
2. 将root节点和其两个子节点中**较大的值**作比较，如果符合条件则替换
3. 重复第二步骤，逐步向下修复最大堆属性

-> 时间复杂度为

### UpdateKey(i, newv)

-> 如果值的索引i已知，则可以直接更新A[i]=newv

-> 然后向上和向下修复最大堆属性

-> 在知道索引的情况下，时间复杂度为

### Delete(i)

1. 将该索引的值提升至root节点的值+1，使其成为最大堆中最大的数
2. 向上修复最大堆属性ShiftUp
3. 进行ExtractMax()操作

-> 时间复杂度为