DSAA

My viewpoints

递归关键的理解是递归树和栈。

在递归函数后面的内容会被压入栈中，从return开始对代码从深度上往上一层一层向上执行。而调用递归函数的前面的部分是从上向下执行的，

总结来说，正确的理解是：

- 在递归调用时，栈顶的函数调用会“等待”递归调用返回后才会继续执行。即**继承**上一函数的状态。
- 递归函数调用前的代码是按“自上而下”的顺序执行的，递归函数调用后的代码则按“自下而上”的顺序执行（即从最深的递归返回并逐层向上执行）。

Binary Search

二分查找最关键的是确定边界条件！要想清楚遇到相同的被查找的数和相邻时应该怎么处理

```
while(left<right){  //找到数组中最接近x的最靠右的左边界。
            int mid=((left+right)+1)>>1;    //向上取整 防止无限循环
            if(a[mid]<x){
                left=mid;
            }
            else if(a[mid]==x){
                left=mid;      //因为向下取整的时候，相邻时left没有前进，会无限循环
            }
            else if(a[mid]>x){
                right=mid-1;
            }
        }
        left=1,right=N;
        while(left<right){  //最接近y的最靠左的右边界。
            int mid=(left+right)>>1;
            if(a[mid]<y){       //因为后面是+1所以不用向上取整
                left=mid+1;
            }
            else if(a[mid]==y){
                right=mid;
            }
            else if(a[mid]>y){
                right=mid;
            }
        }
//二者最关键的区别是遇到==时的处理方法，和左右那边多移动一格
```

Sort

O(n) Algorithms

Selection sort

每一次遍历未排序的数组，找出最小值放在排好的位置上

1. for integer T <- 1 to n-1 O(n)
2. k<- i
3. for integer integer j <- i+1 to n ---O(n^2)

   if A[k] > A[j] then

   k<-j
4. swap A[i] and A[k]

Insertion sort

遍历数组每一个值，将每个元素放到排好的数组合适的位置(Online algorithm)

1. for integer 1 -> n --O(n)
2. for integer j <- i to 1 (j>1) --O(n^2)

   if A[j-1]>A[j]

   swap A[j-1] and A[j]

   else break

Bubble sort

1. for integer i from 1 to n-1
2. for integer j 1 to n-i ---**O(n^2)**

   if A[j]>A[j+1]

   swap A[j+1] and A[j]

bonus 7 8 9 5 4 1 3 6 最优排序方法

O(n($\log_n$)) Algorithm

Merge Sort (归并排序)

Divade-and-comquer

归并排序的特点： 归并排序在合并两个有序数组的过程中，可以方便地统计逆序对的数量。

1. 伪代码

```
if n>1
    p <- [n/2]
    B[1…p] <- A[1…p]
    C[1…n-p] <- A[p+1…n]
-divide
-combine
    merge-sort(B,P)
    merge sort (C,n-P)
    A<- combine(B,C,p,n-p)
```

2. **Combine**?

   用 i,j,k 三个指针标记两个数组和目标数组，并根据大小关系进行移动，在一个数组完成进入后，一定要对另一个数组也进行归入

3.递归拆分

- `merge_sort(a, 1, 3);`
  - `merge_sort(a, 1, 2);`
    - `merge_sort(a, 1, 1);` (返回)
    - `merge_sort(a, 2, 2);` (返回)
    - 合并 `a[1..1]` 和 `a[2..2]`
  - `merge_sort(a, 3, 3);` (返回)
  - 合并 `a[1..2]` 和 `a[3..3]`
- `merge_sort(a, 4, 5);`
  - `merge_sort(a, 4, 4);` (返回)
  - `merge_sort(a, 5, 5);` (返回)
  - 合并 `a[4..4]` 和 `a[5..5]`
- 合并 `a[1..3]` 和 `a[4..5]`

Master Theorem(主定理)

$$T(n) = \alpha T(\lceil \frac{n}{\beta} \rceil) + O(n^r)(n \geq 2)$$

$$if(log_\beta \alpha < r), T(n) = O(n^r)$$

$$if(log_\beta \alpha = r), T(n) = O(n^r logn)$$

$$if(log_\beta \alpha > r), T(n) = O(n^{log_\beta \alpha})$$

Heap Sort(Binary tree)

Quick Sort　(A[1...n,lo=i,hi=n])

How to Achieve

1. Random pick an integer pivot in A
2. 把数组重新组合，比他小的放左边，大的放右边
3. 重复上面的操作

```
if(left>=right)return
p <- partition(A,lo,hi)
quicksort(A,lo,p-1)
quicksort(A,p+1,hi)
```

```
function partition(arr, low, high):
    randomPivotIndex = random number between low and high
    // 将随机选择的 pivot 与 high 位置的元素交换
    swap arr[randomPivotIndex] and arr[high]
    // 将 arr[high] 作为基准值
    pivot = arr[high]
    i = low - 1   // i 用于标记小于或等于 pivot 的区域末尾索引
    // 遍历数组从 low 到 high - 1
    for j = low to high - 1:
        if arr[j] <= pivot:
            i = i + 1
            swap arr[i] and arr[j]   // 将小于或等于 pivot 的元素放到左边
    swap arr[i + 1] and arr[high]
    return i + 1
```

```
QuickSort(arr, 0, 7)
|
├─ QuickSort(arr, 0, 6)
|  |
|  ├─ QuickSort(arr, 0, 3)
|  |  |
|  |  ├─ QuickSort(arr, 0, -1) // Returns
|  |  └─ QuickSort(arr, 1, 3)
|  |     ├─ QuickSort(arr, 1, 1) // Returns
|  |     └─ QuickSort(arr, 3, 3) // Returns
|  |
|  └─ QuickSort(arr, 5, 6)
|     ├─ QuickSort(arr, 5, 5) // Returns
|     └─ QuickSort(arr, 7, 6) // Returns
|
└─ QuickSort(arr, 8, 7) // Returns

```

Time complexity

时间复杂度分析的过程需要详细一些，每一部分所需要的复杂度是多少。如果出现了快速排序的思路类似的，需要证明期望是O(nlogn)。写出每一部分需要的复杂度后，再合成，取最大复杂度得到算法的时间复杂度。

Because of random, we cannot calculate the exact complexity, but expectation in statistcs.

Consider the probability we need to compare e_i and e_j . $P(e_{ij}) = \frac{2}{j-i+1}$

then We sum this probability $E(x) = 2\sum_{i=1}^{n-1}\sum_{j=i+1}^{n}\frac{1}{j-i+1}$

then we get $E(x) = O(nlogn) with (c = 2)$

计数排序（Counting Sort）

- **适用范围**：适用于**已知取值范围较小的整数**排序。
- 基本思想
  :
  1. **统计频次**：遍历待排序数组，统计每个元素出现的次数。
  2. **累加频次**：对频次数组进行累加，得到元素在排序后的位置索引。
  3. **生成排序结果**：根据累加频次，将元素放置到正确的位置上。
- 特点
  :
  - **线性时间复杂度**：时间复杂度为 O(n+k)O(n + k)O(n+k)，其中 n 是待排序元素的数量，k 是元素的取值范围。
  - **稳定性**：计数排序是稳定的排序算法，能保持相同元素的相对位置。

桶排序（Bucket Sort）

- **适用范围**：适用于**元素值分布较为均匀的数组**，且元素可以映射到桶中。
- 基本思想
  :
  1. **分配桶**：根据元素值的分布，将元素分配到不同的桶中。
  2. **桶内排序**：对每个桶内的元素进行排序，可以使用任何稳定的排序算法。
  3. **合并结果**：依次遍历每个桶，合并桶内元素得到排序结果。
- 特点
  :
  - **平均线性时间复杂度**：在理想情况下，时间复杂度为 O(n)O(n)O(n)。
  - **对数据分布敏感**：当数据分布不均匀时，桶排序的性能会下降。

Data Structure Linked list

A有两个值{pointer to the next location , now value}

链表与数组

- 链表可以动态分配空间，但每一个node用的储存空间更大
- 链表的删除和插入操作复杂度是O(1),数组是O(n)
- 数组可以进行查找和排序，链表不容易实现。但是可以**创建指针数组对链表排序**
- 链表的访问必然是O(n)

link list 可以用于适宜存储性质的数据结构，是一种容器。

也可以做成一个双向链表，即每一个node储存他的前一个和后一个的地址。

Traverse(A):

while(A.value!=null)

    print A.value

    A <- A.next

Insert Node(A,node p,i)

1. a<- 0 , node P <- A
2. while(i-1>a) —**count and traverse**
    1. p<- p.next
    2. a <- a+1
3. tmp <- p.next
4. p.next <- q
5. q.next <- tmp
6. return

Delete Node(node A,i)

1. a<- 0 , node P <- A
2. while(i - 1 > a) —**count and traverse**
    1. p<- p.next
    2. a <- a+1
3. p.next <- p.next.next
4. return A

Time complexity above is all

time: O(n) space: O(1)

Stack and queue O(1)

First In Last Out (FILO)

STD 库中Stack的实现：

当储存Stack超出数组的size时，std库会根据内存情况double或half double储存空间。

Application of Stacks

Brackets Balance Problem

Methodology

 Employ stack store checked left bracket Pop out left bracket if it is matched

Postfix expression

Infix 到 Postfix 的转换规则：

1. **操作数直接输出**：当遇到操作数时，直接将其添加到输出中。
2. 运算符按优先级处理
   - 如果运算符栈为空或栈顶是左括号 `(`，则直接将当前运算符压入栈中。
   - 如果当前运算符的优先级大于栈顶运算符的优先级，则将当前运算符压入栈中。
   - 如果当前运算符的优先级小于或等于栈顶运算符的优先级，则弹出栈顶运算符并将其添加到输出中，直到遇到优先级更低的运算符或栈为空，然后将当前运算符压入栈中。
3. **遇到左括号**：将其直接压入栈中。
4. **遇到右括号**：弹出栈顶运算符，直到遇到左括号为止（左括号不输出）。
5. **栈中剩余的运算符**：当表达式扫描完毕后，将栈中剩余的运算符依次弹出并添加到输出中。

Postfix Expression Evaluation

 5 9 3 + 4 2 * * 7 + *

 Methodology

 Read the tokens in one at a time

 If it is an operand, push it on the stack If it is a binary operator:

 pop top two elements from the stack,

 apply the operator, and push the result back on the stack

Arithmetic expression $a + b * c$

1. Prefix expression $+a * bc$ 前缀表达式
2. Infix expression $a + b * c$ 中缀表达式
3. postfix expression $abc + *$ 后缀表达式

如何做一个计算器？用栈实现

1. **infix expression -> postfix**
2. **Postfix Expression Evaluation**

Queue

Queue Operations

 enQueue: add an item at the rear of the queue deQueue: remove the item at the front of the queue

 front: get the item at the front of the queue, but do not remove it

 isEmpty: test if the queue is empty isFull: test the queue is full

 clear: clear the queue, set it as empty queue size: return the current size of the queue

Array based Queue

 MAX_SIZE = n // the max size of stack

 front = 0 // the current front rear = 0 // the current rear

 Array S with n elements

单调栈

单调栈是一种特殊的栈数据结构，其元素按一定的顺序（单调递增或单调递减）排列，用于解决一类具有“下一个更大/小元素”或“子数组最大/小值”的问题。单调栈的实现和使用简单但功能强大，特别适用于具有顺序关系的问题中。

作用

单调栈的主要作用是快速找到元素的“下一个更大/小元素”或“前一个更大/小元素”。这种需求在处理滑动窗口、柱状图最大矩形面积、股票价格波动等问题时非常常见。通过单调栈，可以有效地降低时间复杂度，通常可将暴力求解的 (O(n^2)) 时间复杂度降到 (O(n))。单调栈常见于！**对于每一个元素，他只在乎之前（后）元素的最大值或最小值**的情况

实现思想

单调栈的基本实现是依靠栈结构的“后进先出”特性来保持元素的单调性。当加入一个新的元素时，将栈中不满足单调性的元素弹出，以维持栈内的单调顺序。通过这种方式，栈内始终保留当前元素之前的“合适的”值，能够实现常数时间复杂度的查找操作。

单调栈有两种主要类型：

- **单调递增栈**：从栈底到栈顶元素逐渐增大。
- **单调递减栈**：从栈底到栈顶元素逐渐减小。

代码实现

以下是一个单调递增栈的实现示例，解决“下一个更大元素”问题。对于每个元素，找到其右边第一个比它大的元素，若没有则返回 -1。

代码解析

- `stack<int> s;` 使用栈来存储元素的索引，而不是值，便于在 `result` 数组中直接更新对应位置。
- `while` 循环：栈顶元素若小于当前元素，表示找到了栈顶元素的下一个更大值。
- `s.push(i);` 将当前元素的索引入栈，便于后续判断其下一个更大值。

时间复杂度

由于每个元素最多入栈和出栈各一次，时间复杂度为 (O(n))，适合大规模数据的高效处理。

String

Brute Force

 Check for pattern starting at every text position

```
BruteForce(T, P):
 n ← len(T), m ← len(P)
 for i ← 0 to n-m-1
     for j ← 0 to m-1
         if P[j] != T[i+j] then
             break;
    if j = m-1
        pattern occurs with shift i
```

 Time complexity

Worst case: m*n comparisions

 Too slow when m and n are large

Rabin-Karp Algorithm

1. 将字符串转换为数字

字母表和基数

假设字符串的字符来自一个字母表 Σ，例如字母表可以是 `{a, …, z, A, …, Z}`，也就是包含大小写英文字母，总共 $d = |\ \Sigma\ | = 52$个字符。

为了将一个字符串转换成一个数字，可以使用 **基数** d（即字母表的大小）和一个哈希规则 **Horner's Rule**（霍纳规则），按以下方式对字符串的每一位字符进行加权计算，使得不同的字符序列产生不同的整数值。

Horner's Rule

对于一个包含 m个字符的子串$P[0], P[1], \ldots, P[m-1]$，可以将其转换成一个整数哈希值 p：

$$p = P[m-1] + d \cdot (P[m-2] + d \cdot (P[m-3] + \ldots + d \cdot (P[1] + d \cdot P[0]) \ldots))$$

其中 P[i] 是子串中的第 i个字符，对应的整数编码。这样我们得到的 p 就是子串的哈希值。

2. 模运算以防哈希溢出

当 m 很大时，直接计算 p 可能导致整数溢出（即数字过大）。为了解决这个问题，通常将结果对一个质数 q取模，以保证计算结果不超出整数范围：

$$p = (P[m-1] + d \cdot (P[m-2] + d \cdot (P[m-3] + \ldots + d \cdot (P[1] + d \cdot P[0]) \ldots)))mod\ q$$

3. 滚动哈希

滚动哈希（Rolling Hash）是一种技术，可以在 O(1)时间内计算下一个子串的哈希值。

- **初始哈希值**：先计算第一个子串（长度为 m）的哈希值 t[0]。
- **递推计算**：假设我们已经知道 t[i]，可以在 O(1) 时间内通过以下公式计算 t[i+1]：

$$t[i+1] = (d \cdot (t[i] - h \cdot T[i]) + T[i+m])mod\ q$$

其中：

- T[i] 是文本中的第 i个字符。
- T[i+m] 是新加入窗口的字符。
- $h \equiv d^{m-1} \mod q$是一个常数，用于抵消窗口最左边字符的贡献。

这样我们就可以通过前一个哈希值 $t[i]$快速计算出下一个窗口的哈希值 $t[i+1]$，从而在 O(n−m) 时间内完成整个字符串的哈希匹配。

Rabin-Karp算法简介

Rabin-Karp是一种字符串匹配算法，用于在文本 `T` 中查找模式 `P` 的位置。它通过哈希值快速比较，提升匹配效率。

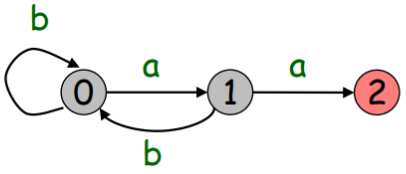
Algorithm: Rabin-Karp(T, P, d, q):

```
1. n ← len(T), m ← len(P)
2. h ← d^(m-1) (mod q), p ← 0, t0 ← 0
3. for j ← 0 to m-1
4.       p ← (dp + P[j]) mod q,
5.       t0 ← (dt0 + T[j]) mod q,
6. for i ← 0 to n-m
7.       if p != ti then
8.             ti+1 ← (d(ti-T[i]h)+T[i+m]) mod q
9.       else
10.            If P[0,..m-1]=T[i,i+m-1]
11.                  pattern occurs with shift i
12.            Else
13.                  ti+1 ← (d(ti-T[i]h)+T[i+m]) mod q
```

Finite State Automata

- A finite State automaton is defined by:
  - Q, *a set of states*
  - $q_0 \in Q$, *the start state*
  - $A \subseteq Q$, *the accepting states*
  - Σ, *the input alphabet*
  - δ, *the transition function, from* $Q \times \Sigma$ *to* Q

	0	1
a	1	2
b	0	0

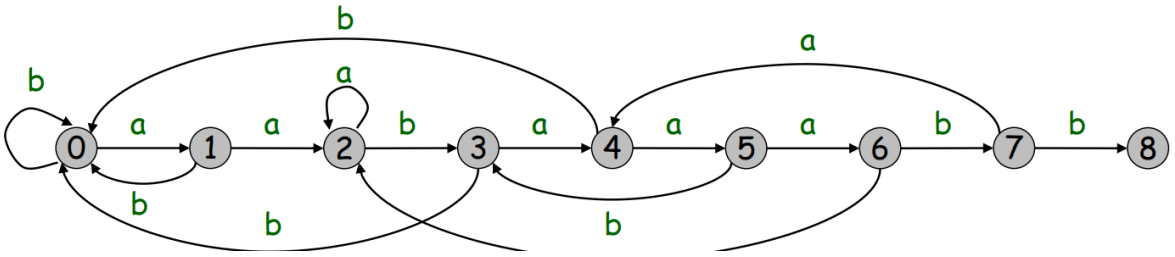


FSA construction

Search Pattern							
a	a	b	a	a	a	b	b

	0	1	2	3	4	5	6	7
a	1	2	2	4	5	6	2	4
b	0	0	3	0	0	3	7	8

j	pattern[1..j]							X
0								0
1	a							1
2	a	b						0
3	a	b	a					1
4	a	b	a	a				2
5	a	b	a	a	a			2
6	a	b	a	a	a	b		3
7	a	b	a	a	a	b	b	0



算法分析

1. **输入**：
   - `P`：模式串，长度为 `m`。
   - `Σ`：字符集，表示所有可能的字符。
2. **输出**：
   - `δ`：转移函数，表示从每个状态在读取某个字符后，转移到哪个状态。

步骤 1：初始化

```
1. m ← len(P)
2. X ← 0
3. Initialize δ(0,a) for each a ∈ Σ
```

- `m ← len(P)`：计算模式串 `P` 的长度。
- `X ← 0`：这是一个辅助变量，初始化为 `0`，用于存储当前状态的转移信息。
- `Initialize δ(0,a) for each a ∈ Σ`：对于初始状态（`0`），我们需要为字符集 `Σ` 中的每个字符初始化转移。这里的 `δ(0,a)` 表示从状态 `0` 在读取字符 `a` 时应该转移到哪个状态。初始时，可能都没有匹配，所以下面的转移会根据匹配情况填充。

步骤 2：填充转移表

```
for j ← 1 to m-1
 for each character a ∈ Σ
 if P[j+1] = a then // char match
 δ(j,a) ← j + 1
else // char mismatch
 δ(j,a) ← δ(X,a)
```

- `for j ← 1 to m-1`：遍历模式串 `P` 的每个位置，`j` 从 `1` 到 `m-1`，即我们跳过 `P[0]`（假设状态 `0` 代表空字符匹配）。
- `for each character a ∈ Σ:`
  - **如果字符匹配**（`P[j+1] = a`）：即模式串 `P` 的当前位置字符与当前字符 `a` 匹配，那么我们会将转移表 `δ(j,a)` 设置为 `j + 1`，意味着如果我们在状态 `j` 时接收到字符 `a`，我们会转移到状态 `j + 1`。
  - **如果字符不匹配**：即 `P[j+1] != a`，则我们根据上一次的匹配状态 `X` 来设置转移，即 `δ(j,a) = δ(X, a)`。这个回退操作利用了上一步计算的转移函数，从而避免重新匹配已经匹配过的部分。

步骤 3：更新辅助变量 `X`

```
 X ← δ(X,P[j])
```

- 这里，`X ← δ(X, P[j+1])` 表示更新 `X` 为当前状态 `X` 在接收到模式串 `P[j]` 时的转移状态。这是为了在下一次处理字符时，能够继续参考之前已经匹配的部分。**(这里一定注意X是延后更新的，差行复制)**

步骤 4：返回转移函数

`return δ`

- 最后返回填充完成的转移函数 δ。

转移函数的工作原理

构造出来的 δ 函数描述了从每个状态出发，在接收到某个字符时应该转移到哪个状态。它的核心思想是：通过动态更新模式串匹配状态，避免重复匹配已经处理过的部分，从而加速匹配过程。

KMP 算法：字符串匹配

KMP (Knuth-Morris-Pratt) 算法通过使用 **部分匹配表（`π` 数组）** 来高效地查找文本 `T` 中是否包含模式串 `P`，并且避免了暴力匹配中重复比较的计算，从而将时间复杂度从 `O(n * m)` 降低到 `O(n + m)`。

自己的理解： 之所以看前后缀，对于第i个主串是因为子串的前半部分与主串的后半部分已经匹配过等价于主串的前后缀匹配。

KMP 算法的核心步骤：

1. **初始化**：
   - `n ← len(T)`：获取文本串 `T` 的长度。
   - `m ← len(P)`：获取模式串 `P` 的长度。
   - `π ← NextArray(P)`：调用 `NextArray` 函数，构建模式串 `P` 的 **部分匹配表**（或 `π` 数组）。
   - `q ← 0`：初始化模式串当前匹配的位置 `q` 为 `0`，表示模式串的第一个字符还没有匹配。
2. **遍历文本串**：
   - 遍历文本串 `T` 中的每个字符（`i` 从 `1` 到 `n`），尝试与模式串 `P` 进行匹配。
3. **匹配过程中**：
   - `while q > 0 and P[q+1] != T[i]`
     :
     - 当当前匹配的字符不相等时，通过 `π` 数组回退模式串的位置 `q`，寻找上一个可能的匹配点。`π[q]` 表示当前匹配失败时模式串的下一次匹配位置。
   - `q ← π[q]`：回退到 `π[q]`，即利用部分匹配表跳过已经匹配的部分。
4. **字符匹配**：
   - `if P[q+1] = T[i]`
     :
     - 如果当前文本字符 `T[i]` 与模式串的字符 `P[q+1]` 匹配，更新 `q`，表示模式串成功匹配了一个字符。
   - `q ← q + 1`：增加 `q`，表示模式串向后推进一个字符，准备匹配下一个字符。
5. **模式串匹配成功**：
   - `if q == m`
     :
     - 当 `q == m` 时，表示模式串 `P` 已经完全匹配成功，打印匹配的位置。
     - 打印结果：`print "Pattern occurs with shift" i - m`，即模式串的起始位置为 `i - m`，表示模式串 `P` 在文本 `T` 中的匹配位置。

6. **回退继续搜索**：
    - `q ← π[q]`：继续根据部分匹配表回退，寻找可能的下一个匹配位置。

部分匹配表构建

`next` 数组是 KMP 算法中用来加速字符串匹配的核心部分。它记录了模式串（`P`）中每个位置之前的最长相等的前缀和后缀的长度。通过该数组，在匹配过程中遇到不匹配时，可以跳过已匹配部分，避免重复匹配，提高效率。

初始化：

- `m ← len(P)`：获取模式串 `P` 的长度。
- 创建一个数组 `π`，长度为 `m`，用于记录模式串每个位置的部分匹配值（即前缀和后缀的最大匹配长度）。
- 初始条件：`π[0] = 0`，表示第一个字符没有前后缀匹配。
- 变量 `k ← 0`：表示当前匹配的前缀长度。

1. **遍历模式串**： 从 `q = 2` 开始，遍历模式串的每一个字符 `P[q]`。注意这里的 `q` 是从 2 开始的，因为 `π[1]` 已经初始化为 0。
2. **回退机制**（通过 `while`）：
    - 当 `P[k+1]` 和 `P[q]` 不匹配时，`k` 需要根据 `π[k]` 回退，直到找到一个匹配或者回退到 `k = 0` 为止。
    - **回退的原因**：模式串的前缀已经部分匹配，所以可以跳过已经匹配的部分，避免重复计算。
3. **匹配判断**：
    - 如果 `P[k+1] == P[q]`，说明当前字符匹配成功，`k` 增加 `1`，并将 `π[q] = k`。
    - 如果不匹配，通过 `π[k]` 继续回退，直到找到匹配或者回退到 `k = 0`。
4. **结束**：
    - 当遍历完成时，`π` 数组即为完整的部分匹配表。

```
m ← len(P)                // 模式串长度
Let π[1,…,m] be a new array  // 创建数组 π
π[1] = 0, k ← 0            // 初始化第一个位置为 0，k 为 0
for q = 2 to m              // 从第二个字符开始遍历模式串
    while k > 0 and P[k+1] != P[q]  // 如果不匹配，回退
        k ← π[k]            // 回退到 π[k]
    if P[k+1] = P[q]        // 如果匹配
        k ← k + 1           // 增加匹配长度
    π[q] ← k                // 记录 π[q]
return π                   // 返回构建好的 π 数组
```

- **部分匹配表**：`π[q]` 表示模式串 `P[1...q]` 中，前缀和后缀的最长匹配长度。
- **回退机制**：通过 `π[k]` 回退，避免重复比较已经匹配的部分，提高效率。
- **时间复杂度**：构建 `π` 数组的时间复杂度是 **O(m)**，其中 `m` 是模式串的长度

Tree

Tree Property I: A tree with n nodes has n-1 edge.

Def: node `u` is a parent of node `v` if `v` is the node directly below u.

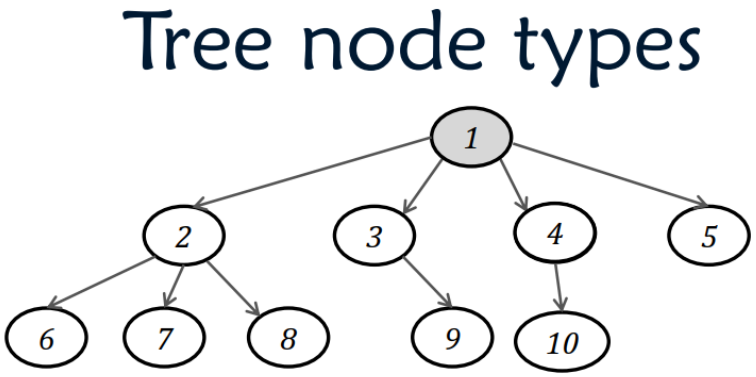
We say node `v` is a child of `u`

Sibling: 同一个父节点的子节点

一个节点只能有一个父，可以有多个子节点。

Consider a tree `T` . Let `u` and `v` be two nodes in T. let u and v be two nodes in `T` , we sat that u is an **ancestor** of v if one of the following holds:

1. **u=v** (not a proper ancestor)
2. u is a parent of v
3. u is a parent of an ancestor of v.



- A *leaf* node is a node without children
- An internal node is a node with one or more children
- E.g.,
  - Leaf nodes: 5, 6, 7, 8, 9, 10
  - Internal nodes: 1, 2, 3, 4

中间节点：拥有至少一个子节点的节点

计算内部节点数:

$$内部节点数 = \left\lceil \frac{N-1}{K} \right\rceil = \frac{N-1+K-1}{K}$$

使用整数运算实现向上取整。

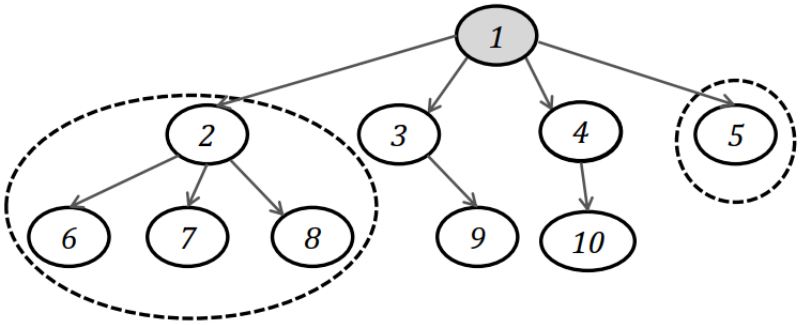
计算叶子节点数:

$$叶子节点数 = N - 内部节点数$$

Tree Property II

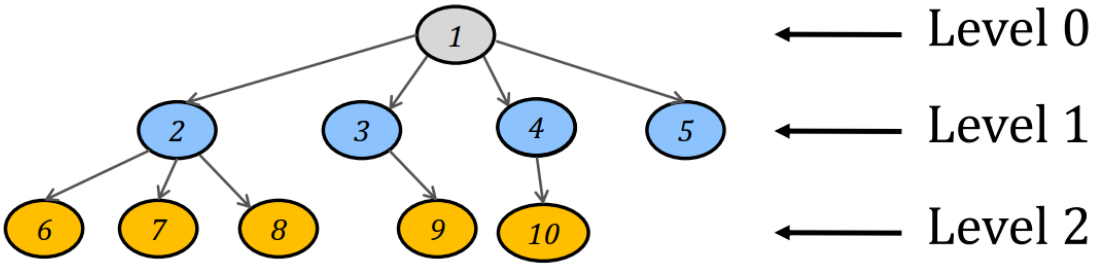
Let T be a tree where every internal node has at least 2 child nodes. If m is the number of leaf nodes, then the number of internal nodes is at most m-1.

Tree: a Recursive Data Structure



- Each node in the tree is the root of a smaller tree!
  - Refer to such trees as subtrees to distinguish them from the tree as a whole
  - Example: node 2 is the root of the subtree circled above
  - Example: node 5 is the root of a subtree with only one node.
- We will see that tree algorithms often lend themselves to recursive implementations

Path, Depth, Level, and Height



- There is exactly one path (one sequence of edges) connecting each node to the root.
- depth of a node = # of edges on the path from it to the root.
- Nodes with the same depth form a level of the tree
- The height of a tree is the maximum depth of its nodes: the tree above has a height of 2.

K-ary tree

A K-ary tree is a roted where every internal node has at most k child nodes(as tied as possible)

Binary Tree

 Binary tree **recursive definition**:

 A binary tree is either:

1. empty or
2. a node (the root of the tree) that has
    1. one or more pieces of data (the key, and possibly others)
    2. a **left subtree**, which is itself a binary tree
    3. a **right subtree**, which is itself a binary tree

 A binary tree implies an ordering among the nodes at the same level.

Binary Tree: Full Level

Consider a binary tree with height h, its level l ($0 \le l \le h$) is full if it contains 2 l nodes.

Binary Tree: Complete Binary Tree

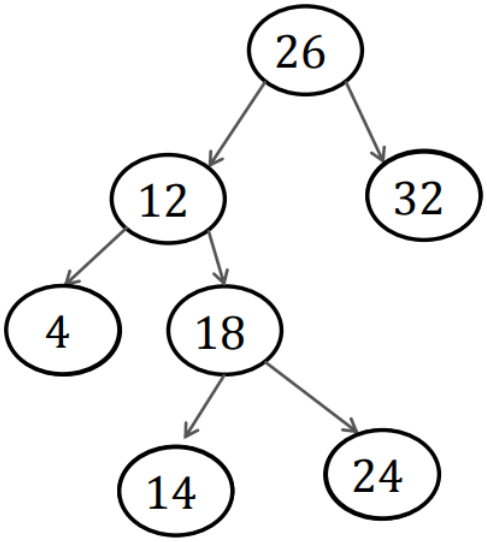
 A binary tree of height h is complete if:

 Level 0, 1, ..., h-1 are all full

 At level h, the leaf nodes are "**as far left as possible**"

 This means that if you want to add a leaf node v at level h, v would need to be on the right of all the existing leaf nodes.

Traversal



- level traversal: 广度优先搜索

  top to bottom left to right

26 12 32 4 18 14 24 (use queue to utilize)

- Preorder traversal

  root , left , right

  26 12 4 18 14 24 32

- Inorder traversal

  left root right

  4 12 14 18 24 26 32

- Postorder traversal

  left right root

  4 14 24 18 12 32 26

We can rebuild this tree by these traversal.

Preorder Traversal

 Implementation:

 Recursive Implementation? So easy?

```
preorderprint(treeNode root):
 print(root)
 if(root->left!=null)
    preorderprint(root->left)
 if(root->right!=null)
    preorderprint(root->right)
```

```
preorderiterative(treeNode root):
 treeNode stack s
 s.push(root)
 while(s!=empty)
 treeNode node= s.top()
 print(node)
 s.pop()
 if(node->right!=null)
 s.push(node->right)
 if(node->left!=null)
 s.push(node->left)
//切换pop，左右的位置关系可以实现不同traversal!!
```

Postorder traversal

逻辑:

- 第一个栈用于按 "根 -> 右 -> 左" 顺序遍历节点。
- 第二个栈用于存储逆序的遍历结果，最终直接逆序输出即为后序结果。

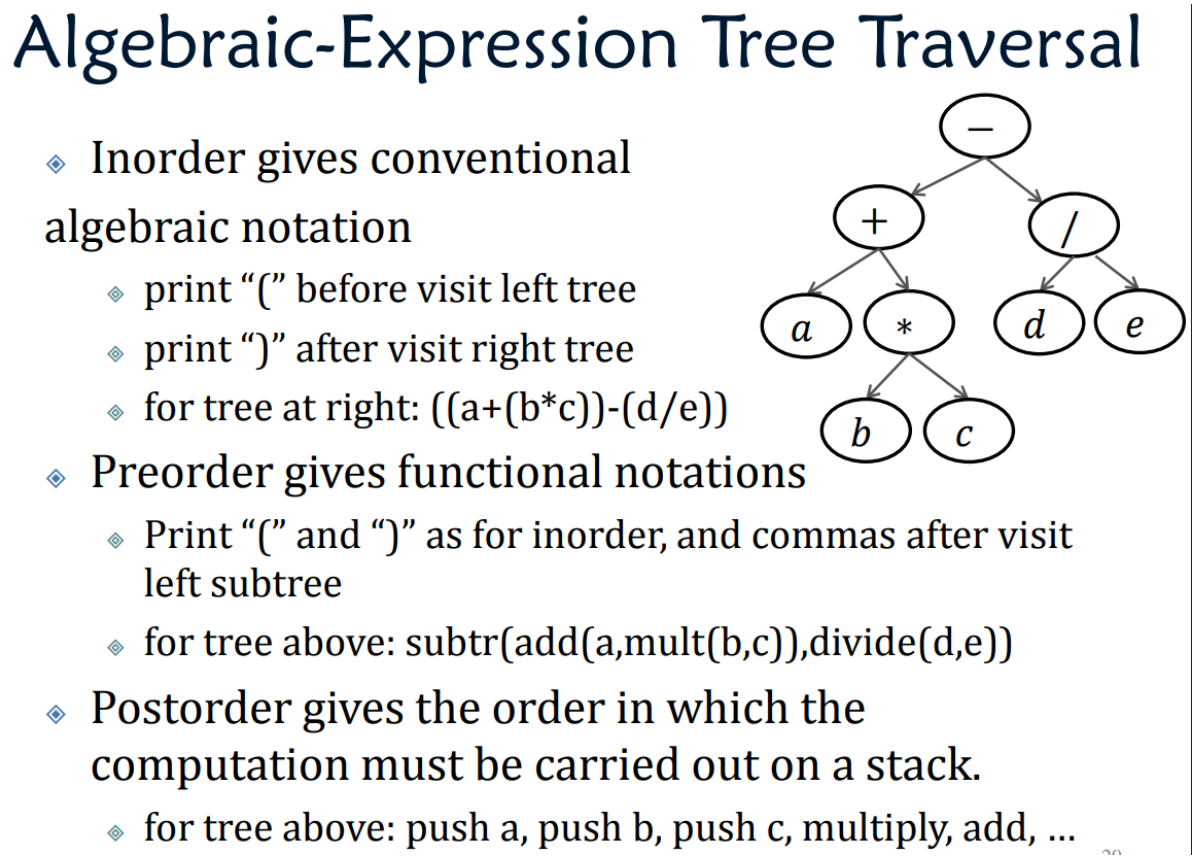
```
postorderTraversalTwoStack(root):
    if root is null:
        return []

    stack1 = []  # 用于遍历
    stack2 = []  # 用于存储结果
```

```
    result = []
    stack1.push(root)
    while stack1 is not empty:
        node = stack1.pop()
        stack2.push(node)
        # 左子树先入栈，保证右子树先处理
        if node.left is not null:
            stack1.push(node.left)
        if node.right is not null:
            stack1.push(node.right)
    # 从stack2依次输出后序结果
    while stack2 is not empty:
        result.append(stack2.pop().value)

    return result
```

Caculater



Postorder 的实现类似于栈实现计算器，把左边叶子push进去，父节点运算后把结果压入栈。

Character Encoding

Fixed encoding

 A character encoding maps each character to a number

 Computers usually use fixed-length character encodings

 ASCII uses 8 bits per character Unicode uses 16 bits per character

 All character encodings have the same length

 A given character always has the same encoding

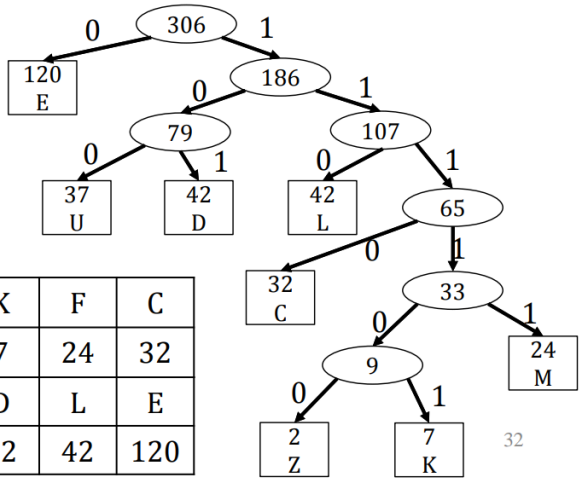
 Problem: fixed length encoding waste space Solution: a variable-length encoding

Huffman tree

1. use encodings of different lengths for different characters.
2. Assign shorter encodings to frequently occurring characters
3. **Requirement**: no character's encoding can be the **prefix** of another character's encoding (e.g., couldn't have 00 and 001)　防止区分不清

Huffman Encoding

- Huffman encoding is a type of variable-length encoding that is based on the actual character frequencies in a given document.
- Huffman encoding uses a binary tree:
  - to determine the encoding of each character
  - to decode an encoded file
- Example:
  - Leaf nodes are characters
  - 101 = 'D'
  - "000110" = "EEEL"



Z	K	F	C
2	7	24	32
U	D	L	E
37	42	42	120

32

哈夫曼树的构建步骤

1. **统计字符频率**:
   - 从文本中读取字符并统计每个字符的出现频率。
2. **创建初始节点列表**:
   - 为每个字符创建一个节点，节点中包含该字符及其频率。
   - 这些节点将用于构建哈夫曼树。
3. **合并最低频率的两个节点**:
   - 从节点列表中找出**频率最低的两个节点**。
   - 将它们合并为一个新节点，该节点的频率是两个子节点频率之和。
   - 新节点的左右子节点分别是刚刚合并的两个节点。
4. **将新节点加入节点列表**:
   - 将新合并的父节点添加回节点列表中，继续构建。
5. **重复合并直到只剩一个节点**:

- 不断重复**步骤3**和**步骤4**，直到列表中只剩一个节点。
- 最终剩下的节点即为哈夫曼树的根节点。

示例：构建哈夫曼树

给定一组字符和其对应频率：

字符	Z	K	F	C	U	D	L	E
频率	2	7	10	12	27	30	43	65

步骤示例：

1. **初始状态**：
   - 将所有字符和频率作为独立节点加入节点列表。
2. **第一次合并**：
   - 频率最小的两个节点为 `Z (2)` 和 `K (7)`。
   - 合并为新节点，频率为 `2 + 7 = 9`。
   - 新节点加入列表中，剩余节点频率为：`9, 10, 12, 27, 30, 43, 65`。
3. **第二次合并**：
   - 频率最小的两个节点为 `9` 和 `F (10)`。
   - 合并为新节点，频率为 `9 + 10 = 19`。
   - 新节点加入列表中，剩余节点频率为：`19, 12, 27, 30, 43, 65`。
4. **重复以上步骤**：
   - 继续合并频率最小的两个节点，直到最后剩下一个节点为根节点。

构建哈夫曼树的关键点

- 每次合并都保证选择频率最小的两个节点。
- 合并过程可以用优先队列（最小堆）优化，以高效选择最小频率的节点。
- 哈夫曼树用于编码字符时，**左分支表示 0，右分支表示 1**。

Priority Queue Implementation

- We will implement a priority queue using a data structure called the “binary heap” to achieve the following guarantees:
  - O(n) space consumption
  - O(log n) insertion time
  - O(log n) delete-min time

- The binary heap data structure is an array object that we can view as a complete binary tree.
  - Level 0 to h-1 are full
  - Leaf nodes in level h are “as far left as possible”

Binary Heap

Binary Heap

- Let S be a set of n integers. A binary heap on S is a binary tree T satisfying:
  - (1) T is complete
  - (2) Every node u in T corresponds to a distinct integer in S, the integer is called the key of u (and is stored at u)
  - (3) If u is an internal node, the key of u is smaller than those of its child nodes
- Note that:
  - Condition 2 implies that T has n nodes
  - Condition 3 implies that the key of u is the smallest in the subtree of u

完全树，节点有唯一值，每一个中间节点都比他的孩子小

Insert Node O(log n)

插入操作主要包括两部分：

1. **在树的底部按顺序插入新节点**（保持完全二叉树的性质）。
2. **调整堆序**（通过上浮操作修复堆序性质）。

伪代码如下：

步骤 1：在树的底部插入节点

- 创建一个新节点 `z`，值为 `e`。
- 将 `z` 插入到二叉堆中唯一可能的位置，确保树仍然是**完全二叉树。**

步骤 2：上浮操作（Heapify-Up）

1. 初始化指针 `u` 指向新插入的节点 `z`。
2. 如果 `u` 是根节点，停止操作（堆序性质已满足）。
3. 否则，比较

   ```
   u
   ```

   的值与其父节点

   ```
   p
   ```

   的值：
   - 如果 `u.key >= p.key`（最小堆）或 `u.key <= p.key`（最大堆），或者已经到了顶堆。
   - 停止操作，堆序已满足。
   - 否则，交换 `u` 和 `p` 的值。
4. 将 `u` 更新为 `p`，重复上述步骤，直到堆序恢复或到达根节点。

删除最小值的操作步骤

1. 删除堆中最小的元素（根节点）。
2. 用堆中最后一个叶节点的值替换根节点。
3. 自上而下（从根到叶）调整堆序（下沉操作，Heapify-Down），使堆重新满足最小堆的性质。

详细步骤：

1. **报告根节点的值**：
   - 因为根节点是堆中的最小值，直接将其输出。
2. **找到底部的最后一个节点**：
   - 找到堆中**最后一个节点**（右下角叶子），它是底层最右侧的节点。
3. **删除最后一个节点，将其值放到根节点**：
   - 删除最后一个节点 `z`，将其值替换到根节点的位置。
4. **开始调整堆序（下沉操作，Heapify-Down）：**
   - 如果当前节点 `u` 是叶节点，停止调整（堆已满足性质）。
   - 否则，检查

```
1 u
```

的两个子节点：

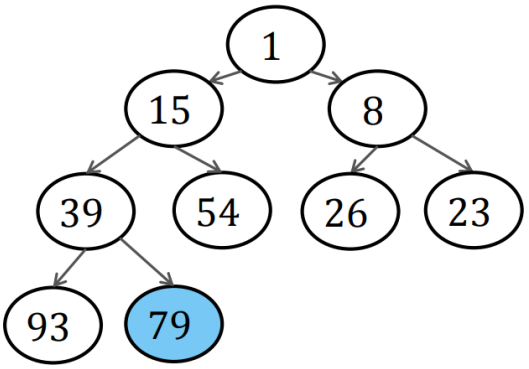
- 如果 `u` 的值小于两个子节点，堆序已满足，停止操作。
- 否则，与值较小的子节点交换位置。

- 将指针 `u` 更新为交换后的子节点，重复调整，直到堆序满足.

如何找第n大的数，把n转换为二进制数，然后根据0左1右就可以找到这个数

How to find rightmost leaf?

- Before we analyzing the time complexity of insert and delete-min, let us first consider a sub-problem:
- Given a complete binary tree T with n nodes, how to identify quickly the rightmost leaf node at the bottom level of T (i.e., colored node in below tree).
  - It is Step 1 in insert algorithm, and Step 2 in delete-min algorithm



Time Complexity Analysis

- We are now ready to prove that our insertion and delete-min algorithms finish in O(log n) time.
- It suffices to point out the key facts:
  - Step 1 of the insertion algorithm (page 8) and Step 2 of the delete-min algorithm (page 13) can be performed in O(log n) time, using our solution to previous sub-problem
  - The rest of insertion ascends a root-to-leaf path, while that of delete-min descends a root-to-leaf path. The time is O(log n) in both cases.
- Thus, we guarantee: (1) O(n) space consumption, (2) O(log n) insertion / delete-min operations.

- **完全二叉树的数组表示**：
  - 树的根节点 `u` 存储在 `A[1]` 位置。
  - 对于任何存储在 `A[i]` 的节点 `u`，它的 **左子节点** 存储在 `A[2i]` 位置。
  - 它的 **右子节点** 存储在 `A[2i + 1]` 位置。
- **为什么左子节点在 `A[2i]`**：
  - 完全二叉树的节点是按层次顺序存储的，也就是说，节点按从上到下、从左到右的顺序排列。
  - 如果节点 `u` 存储在位置 `i`，那么在 `u` 前面已经有 `i-1` 个节点。
  - `u` 的左子节点是 `u` 的左子树的根节点，并且它一定位于 `u` 之后的第一个位置。由于完全二叉树的结构，左子节点的索引应该是 `2i`。
- **为什么右子节点在 `A[2i+1]`**：
  - 同理，右子节点在左子节点之后，也就是存储在 `A[2i+1]`。
  - 这样，数组中的节点可以按层次顺序，依次存储每个节点及其子树。

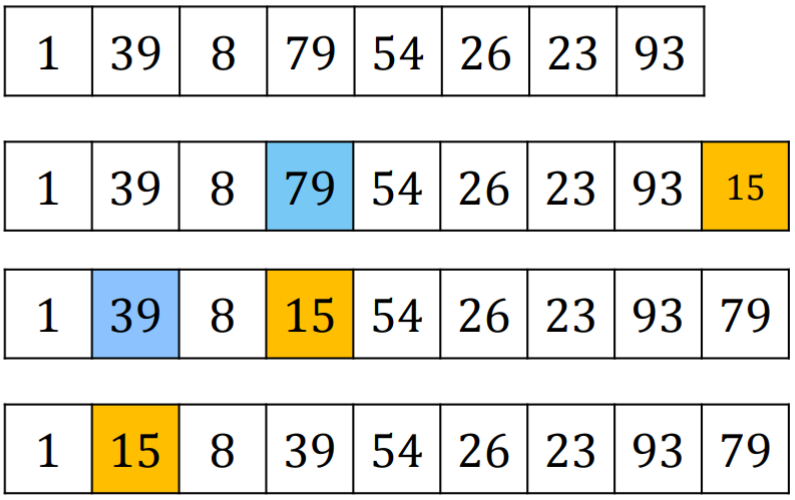
The following is an immediate corollary of the previous lemma:

推论：假设二叉树 `T` 中的节点 `u` 存储在数组 `A[i]` 位置，那么节点 `u` 的**父节点**存储在数组 `A[i//2]` 位置。(向下取整)

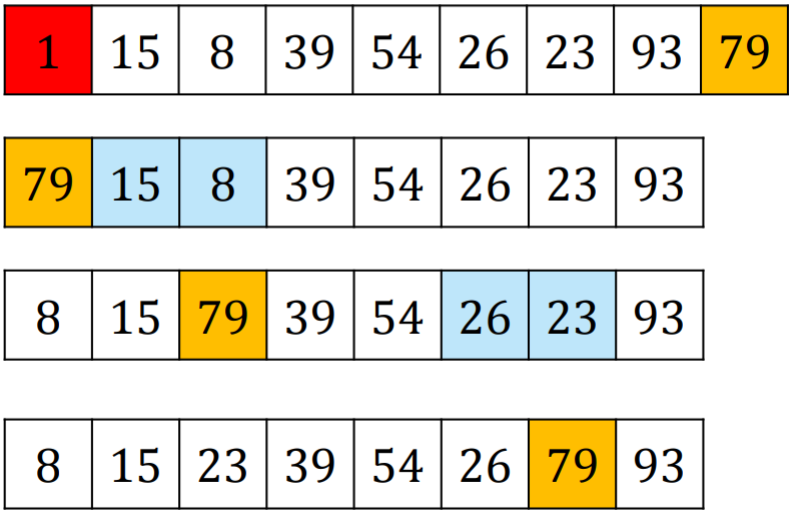
引理：在完全二叉树中，最底层最右边的叶子节点存储在数组 `A[n]` 位置，其中 `n` 是树中的总节点数。

由这几个定理，运用数组进行建堆

Insert 15



Delete-min



Create an array A that stores a set S of n integers, we can turn A into a **binary heap** on S using the following simple algorithm, which view A as a complete binary tree T:

```
For each i=n downto 1:
    Perform root-fix on the subtree of T rooted at A[i]
```

Building a Heap example

54	26	15	93	8	1	23	39
54	26	15	93	8	1	23	39
54	26	15	93	8	1	23	39
54	26	15	93	8	1	23	39
54	26	15	93	8	1	23	39
54	26	15	39	8	1	23	93
54	26	1	39	8	15	23	93
54	8	1	39	26	15	23	93

Root-fix

54	26	15	39	8	1	23	93
54	26	1	39	8	15	23	93
56	8	1	39	26	15	23	93
1	8	15	39	26	54	23	93

Complexity Analysis

- Lemma: The time complexity of turn array A into a binary heap on S is O(n).

- Proof as follows:
  - view A as a complete binary tree
  - The height of T is h.
  - Without loss of generality, assume that all the levels of T are full, i.e., $n=2^{h+1}-1$.
    - Why no generality is lost?
  - Analyze the total running time of Build heap algorithm
  - Proof that $\sum_{i=0}^{h} O\left(i * 2^{h-i}\right) = O(n)$ with $n=2^{h+1}-1$.

错位相减法，可以证明复杂度是线性。

BST

Binary Search Tree (BST)

A BST on a set S of n integers in a binary tree T satisfying all the following requirements:

- T has n nodes
- Each node u in T stores a distinct integer in S, which is called the key of u
- For every internal u, it holds that:
  - The key of u is larger than all the keys in the left subtree of u.
  - The key of u is smaller than all the keys in the right subtree of u.

Predecessor Query , Successor Query time complexity: **O(h)** h be the height of this tree

Insertion and delete: **O(n)**

前驱查询（Predecessor Query）

在一个二叉搜索树（BST）上，给定一个整数集 `S` 和一个查询值 `q`，我们要找到 `q` 的 **前驱节点**（predecessor）。前驱节点是树中 **小于 `q` 的最大节点**。

例如，如果二叉搜索树的节点是 `[20, 10, 30, 5, 15, 25, 35]`，那么对于查询 `q = 15`，前驱节点应该是 `10`，因为 `10` 是小于 `15` 的最大值。

算法步骤

1. **初始化前驱节点**：
   - 设定 `p = -∞`，即初始的前驱值为一个无穷小的值，表示还没有找到前驱。
2. **从根节点开始遍历**：
   - 设定 `u` 为树的根节点。
3. **判断当前节点**：
   - **如果 `u` 为 `nil`**（即当前节点为空），则表示查询结束，返回当前的前驱 `p`。
   - **如果 `u` 的值等于 `q`**，则直接将前驱 `p` 设置为 `q`，并返回 `p`。
   - **如果 `u` 的值大于 `q`**，那么说明 `q` 应该在左子树中。将 `u` 设置为 `u` 的左子节点，继续向左子树遍历。
   - **否则，`u` 的值小于 `q`**，此时我们可以更新前驱节点 `p` 为 `u` 的值，因为此时 `u` 小于 `q`，并且是目前为止最接近 `q` 的一个节点。然后将 `u` 设置为 `u` 的右子节点，继续向右子树遍历。
4. **终止条件**：

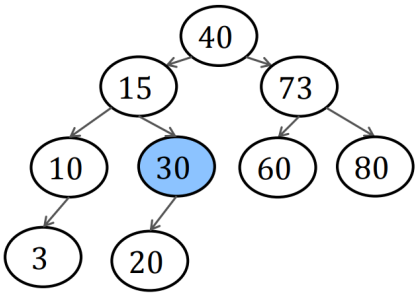
- 当遍历到空节点（`nil`）时，返回当前的前驱值 `p`。

```
function predecessorQuery(T, q):
    p ← -∞  // 初始前驱节点值
    u ← root of T  // 从根节点开始遍历

    while u ≠ nil:
        if u.key = q:
            p ← q  // 如果当前节点的值等于 q，则前驱就是 q
            return p
        else if u.key > q:
            u ← left child of u  // 如果当前节点的值大于 q，则去左子树查找
        else:
            p ← u.key  // 当前节点值小于 q，更新前驱为当前节点
            u ← right child of u  // 继续向右子树查找

    return p  // 最终返回前驱
```

- Suppose that we want to find the predecessor of 35



- (3), (4) and (5) are not true, go to (6)
- Since 30 < 35, p ← 30, since this is the predecessor of 35 so far.
- The predecessor will be in the right subtree of 30, but 30 does not have a right child. So algorithm terminates here with p = 30 as the final answer.

47

在二叉搜索树（BST）中，**后继（Successor）** 和 **前驱（Predecessor）** 是相反的概念：

- **后继节点**：给定一个整数 `q`，它的后继是 BST 中 **最小的** 大于或等于 `q` 的节点。如果 `q` 的值存在于树中，则后继就是 `q` 本身，否则是大于 `q` 的最小值。
- 如果没有比 `q` 更大的元素，则后继不存在。

后继查询（Successor）

假设集合 `S = {3, 10, 15, 20, 30, 40, 60, 73, 80}`，对于查询 `q` 的后继，返回如下：

- `q = 23` 的后继是 `30`，因为 30 是大于 23 的最小值。
- `q = 15` 的后继是 `15`，因为 15 本身就是集合中等于 `15` 的最小值。
- `q = 81` 的后继不存在，因为没有比 81 更大的元素。
-

我们可以通过在二叉搜索树（BST）中遍历来找到后继。后继查询的算法与前驱查询的算法是对称的。即根据 BST 的性质，后继查询可以按照以下规则进行：

1. **初始化**：我们从根节点开始遍历 BST，使用一个变量 `p` 来保存当前找到的后继节点。

2. 遍历:
    - 如果当前节点 `u` 的值等于 `q`，那么 `q` 的后继就是 `u`（即当前节点）。
    - 如果当前节点 `u` 的值大于 `q`，那么 `u` 可能是 `q` 的后继，但我们需要继续向左子树遍历，看看是否能找到更小的后继值。
    - 如果当前节点 `u` 的值小于 `q`，那么我们需要继续向右子树遍历，因为后继节点一定在右子树中。

Insert

BST Insertion

Suppose that we need to insert a new integer e. First create a new leaf z storing the key e. This can be done by descending a root-to-leaf path:

- 1. Set u ← the root of T
- 2. If e < the key of u
    - 2.1 If u has a left child, then set u to the left child
    - 2.2 Otherwise, make z the left child of u, and done
- 3. Otherwise:
    - 3.1 If u has a right child, then set u to the right child
    - 3.2 Otherwise, make z the right child of u, and done.
- Repeat from Step 2.

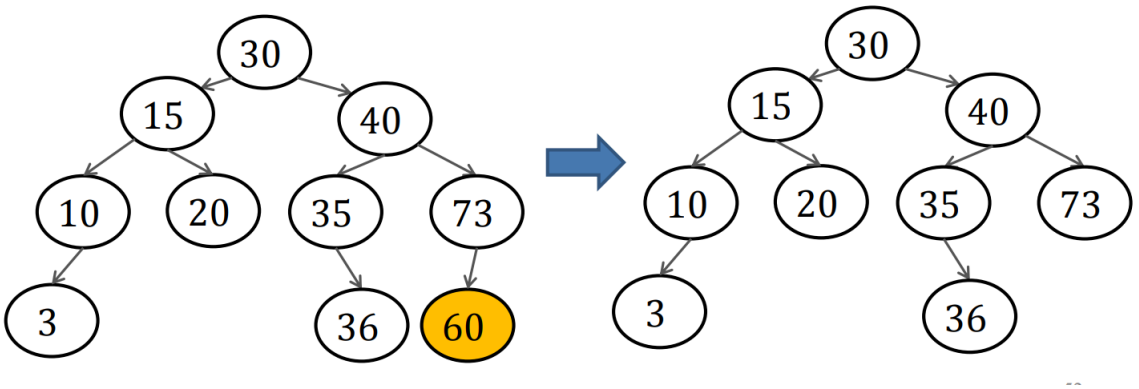
The total cost is proportional to the height of T, i.e., O(h)

找到合适的位置插入

Deletion

BST Deletion

- Suppose that we want to delete an integer e. Frist, find the node u whose key equals to e in O(h) time (through a predecessor query).
- Case 1: if u is a leaf node, simply remove it from T.
- Example: remove 60



- What happens if node u is not a leaf node?
- Case 2: if u has a right subtree:
  - Find the node v storing the successor s of e.
  - Set the key of u to s
  - Case 2.1: if v is a leaf node, them remove it from T
  - Case 2.2: otherwise, it must hold that v has a right child w, but not left child. Replace node v by subtree which rooted at w.
- Case 3: if u has no right subtree:
  - It must hold that u has a left child v, Replace node u by the subtree rooted at v.

2.2：因为如果有左孩子，successor就是他，所以一定只有右孩子。

把右子树的根节点直接覆盖掉交换后的v（被赋值为要删除的successor node)

3: 他一定是一个中间节点，有左子树，直接把做子树的根覆盖掉v。即直接用左子树替换它

复杂度分析：case2 是O(n). case1 and case3