数据结构与算法 2024 概念

2024 Spring，Compiled by Chaos Tam

---

focus on here：

本部分主要关注概念，所以涉及的题比较少。

排序算法

排序算法用于根据元素上的比较运算符重新排列给定的数组或列表元素。比较运算符用于决定各个数据结构中元素的新顺序。

有很多不同类型的排序算法。一些广泛使用的算法包括：

冒泡排序bubble

选择排序selection

插入排序insertion

快排quick sort

归并排序merge

希尔排序shell

其他排序算法可见Sorting algorithms.

冒泡排序bubble sort

就像冒泡泡一样，每一次都是使前最大（最小）的泡泡先冒出来。

因为是原地交换，因此空间复杂度是`O(1)`。

又因为每次都要遍历，一次遍历需要比较待排列部分长度减一次，因此需要比较：

$$\sum_{i=0}^{n-1} m = (1 + 2 + \ldots + n - 1) = n(n-1)/2$$

即时间复杂度为`O(n2)`（最好/最坏/平均）。

稳定性：每次都是相邻元素之间的交换，是稳定的。

```
def bubbleSort(arr):
    n = len(arr)
    for i in range(n):
        for j in range(0, n - i - 1):
            if arr[j] > arr[j + 1]: # 这里保证了稳定性
                arr[j], arr[j + 1] = arr[j + 1], arr[j]
```

改进冒泡排序：使用一个变量存储一次排序中是否有交换，这样提高效率。

```
def bubble_sort_improved(arr):
    n = len(arr)
    swapped = True
    while swapped:
        swapped = False
        for i in range(n-1):
            if arr[i] > arr[i+1]:
                arr[i], arr[i+1] = arr[i+1], arr[i]
                swapped = True
        n -= 1  # 因为每次遍历后，最大的数都会移到正确的位置，所以下次遍历不需要再考虑它

# 示例
arr = [64, 34, 25, 12, 22, 11, 90]
bubble_sort_improved(arr)
print("排序后的数组:", arr)
```

选择排序selection sort

每次选择最小的元素放到已排序部分后面。

因为是原地交换，因此空间复杂度是`O(1)`。

又因为每次都要遍历，一次遍历需要比较待排列部分长度减一次，因此需要比较：

$$\sum_{i=0}^{n-1} m = (1+2+\ldots+n-1) = n(n-1)/2$$

即时间复杂度为`O(n2)`（最好/最坏/平均）。

稳定性：可能涉及非相邻元素之间的交换，不稳定！

原始：**3** 5 4 3 1 2

第一步：1 | 5 4 3 **3** 2

第二步：1 2 | 4 3 **3** 5

第三步：1 2 3 | 4 **3** 5

第四步：1 2 3 **3** | 4 5

第五步：1 2 3 **3** 4 | 5

第六步：1 2 3 **3** 4 5（完成）

可以发现：**3**和3的相对位置发生了改变。

```
for i in range(len(A)):
    min_idx = i
    for j in range(i + 1, len(A)):
        if A[min_idx] > A[j]:
            min_idx = j
    A[i], A[min_idx] = A[min_idx], A[i]
```

快排Quick sort

采用**分治思想。每次选择一个pivot**，比pivot小的放一边，比pivot大的放另一边，两边分别进行快排。因此，快排的时间复杂度与pivot的选择有关。

最简单的方法是将列表中的每个值依次与pivot比较，创建小于和大于的列表，所以一般使用原地交换的方式。

```
# 非原地交换
def quicksort_noninplace(arr):
    if len(arr) <= 1:
        return arr

    pivot = arr[-1]  # 选择最右侧的元素作为基准值
    less = [x for x in arr[:-1] if x <= pivot]
    greater = [x for x in arr[:-1] if x > pivot]

    return quicksort_noninplace(less) + [pivot] + quicksort_noninplace(greater)
```

使用原地交换可以减少空间开销。空间复杂度：`O(logn)`。

```
def quicksort_inplace(arr,left,right):
    if left >= right:
        return

    low,high = left,right
    # 选择第一个元素作为pivot
    pivot = arr[low]

    while left < right:
        while left < right and arr[right] > pivot:
            right -= 1
        arr[left] = arr[right]
        while left < right and arr[left] <= pivot:
            left += 1
        arr[right] = arr[left]
    arr[right] = pivot

    quicksort_inplace(arr,low,left-1)
    quicksort_inplace(arr,left+1,high)
```

举例：

```
5 3 5 7 2 66 8 1  #pivot = 5

left            right # 此时left指向pivot
  |               |
 ( ) 3 5 7 2 66 8 1

left            right # 此时right指向pivot
```

```
  |               |
  1 3 5 7 2 66 8 ( )

       left     right # 此时right指向pivot
        |         |
  1 3 5 7 2 66 8 ( )

       left      right # 此时left指向pivot
         |         |
  1 3 5 ( ) 2 66 8 7

       left right # 此时left指向pivot
         |  |
  1 3 5 ( ) 2 66 8 7

      left right # 此时right指向pivot
        |  |
  1 3 5 2 ( ) 66 8 7

      left right # 结束循环
         \  /
  1 3 5 2 ( ) 66 8 7

  (1 3 5 2) | 5 | (66 8 7)
```

快排的最好和平均时间复杂度为`O(nlogn)`。证明可见：https://b23.tv/YE0WGGI，最坏情况为`O(n2)`。

由上面的例子也可以看出快排是不稳定的。

《算法导论》中则给出了下面的代码

```python
def quicksort_inplace(arr, low, high):
    if low < high:
        # pi 是分区索引，arr[low:pi] 都小于 arr[pi]，arr[pi+1:high] 都大于等于 arr[pi]

        pi = partition(arr, low, high)

        # 对分区左右两侧的元素递归进行快速排序
        quicksort_inplace(arr, low, pi-1)
        quicksort_inplace(arr, pi+1, high)

def partition(arr, low, high):
    # 选择最右侧的元素作为基准值（pivot）
    pivot = arr[high]
    i = low - 1

    for j in range(low, high):
        # 如果当前元素小于或等于基准值
        if arr[j] <= pivot:
            # 增加 i
            i += 1
            # 交换 arr[i] 和 arr[j]
            arr[i], arr[j] = arr[j], arr[i]
```

```
    # 将基准值放到正确的位置
    arr[i+1], arr[high] = arr[high], arr[i+1]
    return i+1

# 示例
arr = [10, 7, 8, 9, 1, 5]
n = len(arr)
quicksort_inplace(arr, 0, n-1)
print("排序后的数组:", arr)
```

举例:

```
1 6 4 2 5 3 # pivot = 3,i=-1

i j           # i = i+1 = 0
  |
  1 6 4 2 5 3

i j           # i = i+1 = 0
 \|
  1 6 4 2 5 3

i j         # i=0
| |
1 6 4 2 5 3

i   j       # i=0
|   |
1 6 4 2 5 3

  i   j     # i=i+1=1
  |   |
1 6 4 2 5 3

  i   j   # swap
  |   |
1 2 4 6 5 3

  i     j # 循环结束
  |     |
1 2 4 6 5 3

  i     j # 交换
  |     |
1 2 3 6 5 4

(1 2) | 3 | (6 5 4)
```

归并排序merge sort

同样采用**分治思想**。不过快排是从上层排列到底层，归并排序则是先分到底层再从底层排序排到顶层。稳定。

```python
def merge_sort(arr):
    if len(arr) <= 1:
        return arr
    # 分割数组
    mid = len(arr) // 2
    left = arr[:mid]
    right = arr[mid:]
    # 递归地对左半部分和右半部分进行排序
    left = merge_sort(left)
    right = merge_sort(right)
    # 合并两个已排序的半部分
    return merge(left, right)


def merge(left, right):
    merged = []
    left_index = 0
    right_index = 0
    # 逐个比较两个数组中的元素，并添加到合并后的数组中
    while left_index < len(left) and right_index < len(right):
        if left[left_index] <= right[right_index]:
            merged.append(left[left_index])
            left_index += 1
        else:
            merged.append(right[right_index])
            right_index += 1
    # 如果左半部分还有剩余元素，将它们添加到合并后的数组中
    while left_index < len(left):
        merged.append(left[left_index])
        left_index += 1
    # 如果右半部分还有剩余元素，将它们添加到合并后的数组中
    while right_index < len(right):
        merged.append(right[right_index])
        right_index += 1
    return merged

# 示例
arr = [38, 27, 43, 3, 9, 82, 10]
print("原始数组:", arr)
sorted_arr = merge_sort(arr)
print("排序后的数组:", sorted_arr)
```

时间复杂度（最好、最坏、平均）均为`O(nlogn)`。

插入排序insertion sort

可以理解为打牌。每次从列表中取出第一个元素，插入到排好的序列中。平均和最坏时间复杂度为`O(n2)`。　　稳定。

```
def insertion_sort(arr):
    for i in range(1, len(arr)):
        j = i
        # Insert arr[j] into the sorted sequence arry[0..j-1]
        while arr[j - 1] > arr[j] and j >= 0:
            arr[j - 1], arr[j] = arr[j], arr[j - 1]
            j -= 1
```

希尔排序shell sort

按照一定的步长进行排序，然后对每组进行插入排序，随着增量减少，每组的元素就越多，当增量被减至1时，元素被分为一组，算法结束。

先将整个待排序的记录序列分割成为若干子序列（由相隔某个“增量”的记录组成）分别进行直接插入排序，然后依次缩减增量再进行排序，待整个序列中的记录“基本有序”时，再对全体记录进行一次直接插入排序。

举例：

```
8 9 1 7 2 3 5 4 6 0

第一遍：gap=length/2=5 [8 3] [9 5] [1 4] [7 6] [2 0] 每个部分进行插入排序
3 5 1 6 0 8 9 4 7 2

第二遍：gap=5/2=2 [3 1 0 9 7] [5 6 8 4 2] 每个部分进行插入排序
0 2 1 4 3 5 7 6 9 8

第三遍：gap=2/2=1
0 1 2 3 4 5 6 7 8 9
```

因为希尔排序可能会跨越好几个元素交换，因此不稳定。看下面例子。

```
8 9 *3* 7 2 3 5 4 6 0   gap=length/2=5 [8 3] [9 5] [*3* 4] [7 6] [2 0]

3 5 *3* 6 0 8 9 4 7 2   gap=5/2=2 [3 *3* 0 9 7] [5 6 8 4 2]

0 2 3 4 *3* 5 7 6 9 8   gap=2/2=1

0 2 3 *3* 4 6 6 7 8 9

*3*和3的相对位置发生改变。
```

因此插入排序稳定，但希尔排序不稳定。

时间复杂度为`O(n)`。

```
def shell_sort(arr):
```

```
    n = len(arr)
    gap = n // 2  # 初始间隔，取数组长度的一半

    # 逐步缩小间隔
    while gap > 0:
        # 对每个子序列进行插入排序
        for i in range(gap, n):
            temp = arr[i]
            j = i
            # 插入排序
            while j >= gap and arr[j - gap] > temp:
                arr[j] = arr[j - gap]
                j -= gap

            arr[j] = temp

        gap //= 2  # 缩小间隔

# 示例
arr = [12, 34, 54, 2, 3]
print("原始数组:", arr)
shell_sort(arr)
print("排序后的数组:", arr)
```

堆排序heap sort

堆排序（Heap Sort）是一种基于二叉堆这种数据结构所设计的排序算法，是选择排序的一种。可以利用数组来模拟堆的结构。堆排序可以分为两个主要步骤：建堆（Build Heap）和堆调整（Heapify）。

```
def heapify(arr, n, i):
    largest = i  # 初始化最大值为根
    left = 2 * i + 1     # 左 = 2*i + 1
    right = 2 * i + 2    # 右 = 2*i + 2

    # 如果左子节点大于根
    if left < n and arr[i] < arr[left]:
        largest = left

    # 如果右子节点大于目前的最大值
    if right < n and arr[largest] < arr[right]:
        largest = right

    # 如果最大值不是根
    if largest != i:
        arr[i], arr[largest] = arr[largest], arr[i]  # 交换

        # 递归地堆化受影响的子树
        heapify(arr, n, largest)

# 堆排序函数
def heap_sort(arr):
    n = len(arr)
```

```
    # 构建一个大顶堆
    # 从最后一个非叶子节点开始向上遍历
    for i in range(n, -1, -1):
        heapify(arr, n, i)

    # 一个个从堆顶取出元素
    for i in range(n-1, 0, -1):
        arr[i], arr[0] = arr[0], arr[i]  # 交换
        heapify(arr, i, 0)

# 示例
arr = [12, 11, 13, 5, 6, 7]
print("原始数组:", arr)
heap_sort(arr)
print("排序后的数组:", arr)
```

举例：

```
     12              13           7              12             6
    /   \          /   \        /   \          /   \          /   \
  11    13  ->   11    12  ->  11    12  ->   11    7   ->   11    7   ->
 / \   /        / \   /       / \   /        / \   /        / \    |
5   6 7        5  6  7       5   6 (13)     5   6 (13)     5 (12) (13)

     11             5              7             5              6
   /   \          /   \          /   \         /   \          /   \
  6     7   ->   6     7   ->   6     5  ->   6     (7) ->   5     (7)  ->
 / \    |      /   \     |    /   \    \    /    \    \    /    \    \
5 (12) (13)  (11) (12) (13) (11) (12) (13) (11) (12) (13) (11) (12) (13)

      5             (5)
    /   \          /   \
  (6)    (7) ->  (6)    (7)
 /    \    \    /    \    \
(11) (12) (13) (11) (12) (13)
```

计数排序counting sort

是一种线性时间复杂度的排序算法，它假设输入是一组整数，并且知道整数的范围。计数排序通过统计每个元素出现的次数，来确定它们在排序后数组中的位置。稳定。

计数排序的本质是一种哈希算法，也就是通过建立映射关系来达到排序的目的。

基本思路：先开辟一个映射数组，然后遍历原数组，数组中的元素是几就在开辟的的映射数组下标为几的位置+1。得到的这个映射数组就包含了原数组中所有元素的映射关系。遍历一遍映射数组找出原数组中出现过的数即可。

时间复杂度`O(n+k)`，空间复杂度`O(n+k)`。k为数据范围。

```
def counting_sort(arr):
    # 找出数组中的最大值和最小值
    max_val = max(arr)
    min_val = min(arr)
```

```
    # 计算范围大小
    range_val = max_val - min_val + 1

    # 初始化计数数组，并全部置为0
    count_arr = [0] * range_val

    # 遍历输入数组，统计每个元素出现的次数
    for num in arr:
        count_arr[num - min_val] += 1

    # 更改count_arr，使得每个索引位置的值表示小于或等于该索引值的元素个数
    for i in range(1, len(count_arr)):
        count_arr[i] += count_arr[i - 1]

    # 创建一个输出数组，用于存放排序后的结果
    output_arr = [0] * len(arr)

    # 从后向前遍历输入数组，保证稳定性（即相等的元素在排序后保持原来的顺序）
    for i in range(len(arr) - 1, -1, -1):
        output_arr[count_arr[arr[i] - min_val] - 1] = arr[i]
        # 减小计数数组对应位置的计数，因为已经有一个元素放入了正确的位置
        count_arr[arr[i] - min_val] -= 1

    return output_arr

# 示例用法
arr = [4, 2, 2, 8, 3, 3, 1]
sorted_arr = counting_sort(arr)
print(sorted_arr)  # 输出: [1, 2, 2, 3, 3, 4, 8]
```

计数排序是具有局限性的，只适合**元素相对较为集中**的数组的排序，而且**只能排整数**，不能排其他类型的数据，如浮点数等等。因为下标只能是整数。对于相对映射是可以对负数进行排序的，因为无论如何最小的那个元素都是映射到0的位置的，不会越界，但是绝对映射不能对负数排序，不然的话下标访问负数就一定会越界访问的。所以计数排序的使用场景还是很局限的。

桶排序bucket sort

其工作原理是将数组分配到有限数量的桶中，然后对每个桶中的元素进行排序，最后再将各个桶中的数据有序的合并起来。

桶排序的时间复杂度理想情况下`O(n+k)`，极端情况为`O(n2)`，其中n是数据规模，k是桶的数量。空间复杂度同样。

桶排序适用于序列中元素的分布比较均匀的场景，这样其划分的每个桶中的数据比较均匀，排序较快，如果不均匀会导致数据被分配到一个桶里面，这会使其他的桶没用，其消耗的时间就主要靠这个桶所使用的排序算法了。

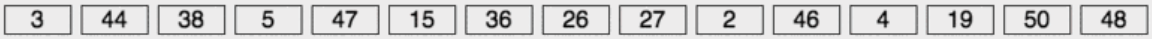
稳定主要看桶中选取的排序算法：因为相同元素会被分配到同一个桶，所以同一个桶所使用的算法不同，则其稳定性不同。

```
def bucket_sort(arr):
```

```
    # Step 1: Create n empty buckets (set the range or number of buckets)
    max_val = max(arr)
    bucket_range = max(1, max_val//len(arr))  # 根据数组长度和最大值来确定桶的大小
    bucket_list = [[] for _ in range((max_val//bucket_range) + 1)]

    # Step 2: Distribute the elements into the buckets
    for num in arr:
        index = num // bucket_range
        bucket_list[index].append(num)

    # Step 3: Sort each bucket and concatenate
    sorted_arr = []
    for bucket in bucket_list:
        # 这里可以使用任何排序算法对桶内的元素进行排序
        # 这里我们简单使用内置的排序函数
        bucket.sort()
        sorted_arr.extend(bucket)

    return sorted_arr

# 示例用法
arr = [4.2, 6.4, 2.3, 9.7, 7.0, 8.3, 1.9, 3.1]
sorted_arr = bucket_sort([int(x * 10) for x in arr])  # 假设我们处理的是小数，先乘以10转为整数
sorted_arr = [x / 10 for x in sorted_arr]  # 排序后再转回原小数形式
print(sorted_arr)  # 输出: [1.9, 2.3, 3.1, 4.2, 6.4, 7.0, 8.3, 9.7]
```

基数排序radix sort

非比较型整数排序算法，其原理是将整数按位数切割成不同的数字，然后按每个位数分别比较。这里以十进制数为例，对于每个数字，我们将其视为字符串（或者数字列表，按位拆分），然后从最低位（个位）开始，对每个位上的数字进行排序，直到最高位。稳定。



原理是将整数按位数切割成不同的数字，然后按每个位数分别比较。基数排序的方式可以采用LSD（Least significant digital）或MSD（Most significant digital），LSD的排序方式由键值的最右边开始，而MSD则相反，由键值的最左边开始。

MSD：先从高位开始进行排序，在每个关键字上，可采用计数排序。
LSD：先从低位开始进行排序，在每个关键字上，可采用桶排序。

```
def radix_sort(array):
    max_num = max(array)
    place = 1
    while max_num >= 10 ** place:
        place += 1
    for i in range(place):
        buckets = [[] for _ in range(10)]
        for num in array:
            radix = int(num/(10 ** i) % 10)
            buckets[radix].append(num)
        j= 0
        for k in range(10):
            for num in buckets[k]:
                array[j] = num
                j += 1
    return array
```

举例：

```
[54,762,85,72,78,32,344]
[762,72,32,54,344,85,78]
[32,344,54,762,72,78,85]
[32,54,72,78,85,344,762]
```

基数排序（Radix Sort）的时间复杂度和空间复杂度取决于几个因素，特别是待排序元素的范围、位数和所选择的基数（通常是10，对于十进制数）。

时间复杂度：基数排序的时间复杂度是线性的，为`O(kn)`，其中k是元素的位数，n是待排序元素的数量。这是因为基数排序需要对每个元素进行k次分配和收集操作，而每次操作的时间复杂度是`O(n)`。因此，总的时间复杂度是`O(kn)`。

空间复杂度：基数排序的空间复杂度取决于所使用的计数数组的大小。在最坏的情况下，当所有元素的某一位上的数字都相同时，计数数组需要存储与输入数组相同数量的元素。因此，空间复杂度是`O(n+k)`，其中k是计数数组的大小（通常等于基数，即10）。此外，如果使用原地（in-place）基数排序算法（这种算法较少见且实现起来更复杂），则空间复杂度可以降低到`O(1)`。

需要注意的是，虽然基数排序的时间复杂度较低，但它仅适用于非负整数，并且当元素范围较大或包含负数时，可能需要额外的预处理步骤来确保算法的正确性。

一些比较和总结

排序方法	时间复杂度（平均）	时间复杂度（最坏）	时间复杂度（最好）	空间复杂度	稳定性
插入排序	$O(n^2)$	$O(n^2)$	$O(n)$	$O(1)$	稳定
希尔排序	$O(n^{1.3})$	$O(n^2)$	$O(n)$	$O(1)$	不稳定
选择排序	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	不稳定
堆排序	$O(nlog_2n)$	$O(nlog_2n)$	$O(nlog_2n)$	$O(1)$	不稳定
冒泡排序	$O(n^2)$	$O(n^2)$	$O(n)$	$O(1)$	稳定
快速排序	$O(nlog_2n)$	$O(n^2)$	$O(nlog_2n)$	$O(nlog_2n)$	不稳定
归并排序	$O(nlog_2n)$	$O(nlog_2n)$	$O(nlog_2n)$	$O(n)$	稳定
计数排序	$O(n+k)$	$O(n+k)$	$O(n+k)$	$O(n+k)$	稳定
桶排序	$O(n+k)$	$O(n^2)$	$O(n)$	$O(n+k)$	稳定
基数排序	$O(n*k)$	$O(n*k)$	$O(n*k)$	$O(n+k)$	稳定

Name	Best	Average	Worst	Memory	Stable	Method	Other notes
In-place merge sort	—	—	`n2logn`	1	Yes	Merging	Can be implemented as a stable sort based on stable in-place merging.
Heap sort	`nlogn`	`nlogn`	`nlogn`	1	No	Selection	
Merge sort	`nlogn`	`nlogn`	`nlogn`	*n*	Yes	Merging	Highly parallelizable (up to *O*(log *n*) using the Three Hungarian's Algorithm)
Tim sort	*n*	`nlogn`	`nlogn`	*n*	Yes	Insertion & Merging	Makes *n-1* comparisons when the data is already sorted or reverse sorted.
Quick sort	`nlogn`	`nlogn`	`n2`	`logn`	No	Partitioning	Quicksort is usually done in-place with *O*(log *n*) stack space.
Shell sort	`nlogn`	`n4/3`	`n3/2`	1	No	Insertion	Small code size.
Insertion sort	*n*	`n2`	`n2`	1	Yes	Insertion	*O*(n + d), in the worst case over sequences that have *d* inversions.

Name	Best	Average	Worst	Memory	Stable	Method	Other notes
Bubble sort	*n*	n2	n2	1	Yes	Exchanging	Tiny code size.
Selection sort	n2	n2	n2	1	No	Selection	Stable with O(n) extra space, when using linked lists, or when made as a variant of Insertion Sort instead of swapping the two items.

1. **直接插入排序**：在最好的情况下，即数据已经是正序时，直接插入排序只需要进行 n-1 次比较和 0 次交换，这时的时间复杂度是 `O(n)`。然而，在最坏的情况下，即数据完全逆序，它需要进行大约 n^2/2 次比较和同样数量的交换，时间复杂度为 `O(n^2)`。
2. **冒泡排序**：在最好的情况下（即数据已经是正序），冒泡排序也只需要进行 n-1 次比较和 0 次交换，时间复杂度为 `O(n)`。在最坏的情况下（即数据完全逆序），冒泡排序需要进行大约 n^2/2 次比较和交换，时间复杂度同样为 `O(n^2)`。
3. **希尔排序**：希尔排序的情况比较特殊，它是基于插入排序的一种改进。希尔排序的性能并不像直接插入排序和冒泡排序那样严重依赖于原始数据的顺序。它通过设定不同的间隔对序列进行部分排序，随着间隔的减少，最终整个列表变得有序。希尔排序的最好情况时间复杂度可以达到 `O(nlogn)`，但最坏情况和平均情况的时间复杂度较难精确计算，一般认为是介于 `O(nlogn)` 和 `O(n^2)` 之间，依赖于间隔序列的选择。

线性结构

线性表

线性表是一种逻辑结构，描述了元素按线性顺序排列的规则。常见的线性表存储方式有**数组**和**链表**，它们在不同场景下具有各自的优势和劣势。

数组是一种连续存储结构，它将线性表的元素按照一定的顺序依次存储在内存中的连续地址空间上。数组需要预先分配一定的内存空间，每个元素占用相同大小的内存空间，并可以通过索引来进行快速访问和操作元素。访问元素的时间复杂度为O(1)，因为可以直接计算元素的内存地址。然而，插入和删除元素的时间复杂度较高，平均为O(n)，因为需要移动其他元素来保持连续存储的特性。

链表是一种存储结构，它是线性表的链式存储方式。链表通过节点的相互链接来实现元素的存储。每个节点包含元素本身以及指向下一个节点的指针。**链表的插入和删除操作非常高效，时间复杂度为O(1)**，因为只需要调整节点的指针。然而，访问元素的时间复杂度较高，平均为O(n)，因为必须从头节点开始遍历链表直到找到目标元素。

选择使用数组还是链表作为存储方式取决于具体问题的需求和限制。**如果需要频繁进行随机访问操作，数组是更好的选择。如果需要频繁进行插入和删除操作，链表更适合。**通过了解它们的特点和性能，可以根据实际情况做出选择。

在Python中，list 更接近于数组的存储结构。

顺序表

python中的顺序表就是列表，元素在内存中连续存放，每个元素都有唯一序号（下标），且根据序号访问。

求元素个数、表尾添加删除元素、随机访问的时间复杂度是`O(1)`。

查找、删除、插入为`O(n)`。

优点：不需要为逻辑关系添加额外空间，可以快速访问表中元素。

缺点：插入和删除耗时大。

链表Linked List

链表是一种常见的数据结构，用于存储和组织数据。它由一系列节点组成，每个节点包含一个数据元素和一个指向下一个节点（或前一个节点）的指针。

在链表中，每个节点都包含两部分：

1. 数据元素（或数据项）：这是节点存储的实际数据。可以是任何数据类型，例如整数、字符串、对象等。
2. 指针（或引用）：该指针指向链表中的下一个节点（或前一个节点）。它们用于建立节点之间的连接关系，从而形成链表的结构。

根据指针的类型和连接方式，链表可以分为不同类型，包括：

1. **单向链表（单链表）：每个节点只有一个指针，指向下一个节点。**链表的头部指针指向第一个节点，而最后一个节点的指针为空（指向`None`）。
2. **双向链表：每个节点有两个指针，一个指向前一个节点，一个指向后一个节点。**双向链表可以从头部或尾部开始遍历，并且可以在任意位置插入或删除节点。
3. **循环链表：最后一个节点的指针指向链表的头部，形成一个环形结构。**循环链表可以从任意节点开始遍历，并且可以无限地循环下去。

链表相对于数组的一个重要特点是，**链表的大小可以动态地增长或缩小，而不需要预先定义固定的大小。**这使得链表在需要频繁插入和删除元素的场景中更加灵活。

然而，**链表的访问和搜索操作相对较慢，因为需要遍历整个链表才能找到目标节点。**与数组相比，链表的优势在于插入和删除操作的效率较高，尤其是在操作头部或尾部节点时。因此，链表在需要频繁插入和删除元素而不关心随机访问的情况下，是一种常用的数据结构。

> 在 Python 中，`list`是使用**动态数组（Dynamic Array）实现的**，而不是链表。动态数组是一种连续的、固定大小的内存块，**可以在需要时自动调整大小。**这使得`list`支持快速的随机访问和高效的尾部操作，例如附加（append）和弹出（pop）。
>
> 与链表不同，**动态数组中的元素在内存中是连续存储的。**这允许通过索引在`list`中的任何位置进行常数时间（O(1)）的访问。此外，动态数组还具有较小的内存开销，因为它们不需要为每个元素存储额外的指针。

> 当需要在 `list` 的中间进行插入或删除操作时，**动态数组需要进行元素的移动，因此这些操作的时间复杂度是线性的（O(n)）**。如果频繁地插入或删除元素，而不仅仅是在尾部进行操作，那么链表可能更适合，因为链表的插入和删除操作在平均情况下具有常数时间复杂度。

总结起来，**Python 中的 `list` 是使用动态数组实现的，具有支持快速随机访问和高效尾部操作的优点。**但是，如果需要频繁进行插入和删除操作，可能需要考虑使用链表或其他数据结构。

单向链表实现

单向链表（单链表）：每个节点只有一个指针，指向下一个节点。链表的头部指针指向第一个节点，而最后一个节点的指针为空（指向 `None`）。

```
class Node:
    def __init__(self,node_data):
        self.data = node_data
        self.next = None
```

```
class LinkedList:
    def __init__(self):
        self.head = None

    def insert(self,item):
        temp = Node(item)
        if self.head == None:
            self.head = temp
        else:
            current = self.head
            while current.next:
                current = current.next
            current.next = temp

    def delete(self,item):
        if self.head == None:
            return
        if self.head.data == item:
            self.head = self.head.next
        else:
            current = self.head
            while current.next:
                if current.next.data == data:
                    current.next = current.next.next
                    break
                current = current.next
```

双向链表实现

双向链表：每个节点有两个指针，一个指向前一个节点，一个指向后一个节点。双向链表可以从头部或尾部开始遍历，并且可以在任意位置插入或删除节点。

```
class Node:
    def __init__(self,data):
```

```
        self.data = data
        self.prev = None
        self.next = None
class DoublyLinkedList:
    def __init__(self):
        self.head = None
        self.tail = None
    def insert_before(self,node,new_node):
        if node is None: # 链表为空
            self.head = new_node
            self.tail = new_node
        else:
            new_node.next = node
            new_node.prev = node.prev
            if node.prev is not None:
                node.prev.next = new_node
            else:  # 如果在头部插入新节点，更新头部指针
                self.head = new_node
            node.prev = new_node
```

在这个示例中，定义了一个 `Node` 类表示双向链表中的节点。每个节点都有一个 `data` 值，以及一个指向前一个节点的 `prev` 指针和一个指向后一个节点的 `next` 指针。

`DoublyLinkedList` 类表示双向链表，它具有 `head` 和 `tail` 两个指针，分别指向链表的头部和尾部。可以使用 `insert_before` 方法在给定节点 `node` 的前面插入新节点 `new_node`。如果 `node` 为 `None`，表示在空链表中插入新节点，将新节点设置为头部和尾部。否则，将新节点的 `next` 指针指向 `node`，将新节点的 `prev` 指针指向 `node.prev`，并更新相邻节点的指针，把新节点插入到链表中。

颠倒列表

```
class Node:
    def __init__(self, data, next=None):
        self.data, self.next = data, next

class LinkList:
    def __init__(self, lst):
        self.head = Node(lst[0])
        p = self.head
        for i in lst[1:]:
            node = Node(i)
            p.next = node
            p = p.next
    def reverse(self):  # 填空：实现函数
        prev = None
        curr = self.head
        while curr:
            next_node = curr.next
            curr.next = prev
            prev = curr
            curr = next_node
        self.head = prev
    def print(self):
        p = self.head
```

```
        while p:
            print(p.data, end=" ")
            p = p.next
        print()

a = list(map(int, input().split()))
a = LinkList(a)
a.reverse()
a.print()
```

循环列表实现

循环链表：最后一个节点的指针指向链表的头部，形成一个环形结构。循环链表可以从任意节点开始遍历，并且可以无限地循环下去。

```
class Node:
    def __init__(self, data, next=None):
        self.data, self.next = data, next

class LinkList:  # 循环链表
    def __init__(self):
        self.tail = None
        self.size = 0

    def isEmpty(self):
        return self.size == 0

    def pushFront(self, data):
        nd = Node(data)
        if self.tail == None:
            self.tail = nd
            nd.next = self.tail
        else:
            nd.next = self.tail.next
            self.tail.next = nd
        self.size += 1

    def pushBack(self, data):
        self.pushFront(data)
        self.tail = self.tail.next

    def popFront(self):
        if self.size == 0:
            return None
        else:
            nd = self.tail.next
            self.size -= 1
            if self.size == 0:
                self.tail = None
            else:
                self.tail.next = nd.next
        return nd.data
```

```
    def printList(self):
        if self.size > 0:
            ptr = self.tail.next
            while True:
                print(ptr.data, end=" ")
                if ptr == self.tail:
                    break
                ptr = ptr.next
            print("")

    def remove(self, data):  # 填空：实现函数
        if self.size == 0:
            return None
        else:
            ptr = self.tail
            while ptr.next.data != data:
                ptr = ptr.next
                if ptr == self.tail:
                    return False
            self.size -= 1
            if ptr.next == self.tail:
                self.tail = ptr
            ptr.next = ptr.next.next
            return True
```

指定位置插入元素

```
class Node:
    def __init__(self, data, next=None):
        self.data, self.next = data, next

class LinkList:
    def __init__(self):
        self.head = None

    def initList(self, data):
        self.head = Node(data[0])
        p = self.head
        for i in data[1:]:
            node = Node(i)
            p.next = node
            p = p.next

    def insertCat(self):
        # 计算链表的长度
        length = 0
        p = self.head
        while p:
            length += 1
            p = p.next

        # 找到插入位置
        position = length // 2 if length % 2 == 0 else (length // 2) + 1
```

```
        p = self.head
        for _ in range(position - 1):
            p = p.next

        # 在插入位置处插入数字6
        node = Node(6)
        node.next = p.next
        p.next = node

    def printLk(self):
        p = self.head
        while p:
            print(p.data, end=" ")
            p = p.next
        print()

lst = list(map(int, input().split()))
lkList = LinkList()
lkList.initList(lst)
lkList.insertCat()
lkList.printLk()
```

其他线性结构

栈 stack

可以理解为一堆书，我们只能在顶端添加或从顶端移走元素。因此最后被添加的元素会最先被移除，即**LIFO**(last-in-first-out)。主要操作包括Stack()、push(item)、pop()、peek()、is_empty()、size()。

可以定义类实现，但也可以直接使用列表实现。

```
class Stack:
    def __init__(self):
        self.items = []
    def is_empty(self):
        return self.items == []
    def push(self, item):
        self.items.append(item)
    def pop(self):
        return self.items.pop()
    def peek(self):
        return self.items[len(self.items)-1]
    def size(self):
        return len(self.items)
```

```
stack = [] # =Stack()
stack.append(item)
stack.pop()
print(stack[-1])
print(not stack)
len(stack)
```

python中也可以导入库。

```
from pythonds.basic import Stack
```

单调栈

单调栈是一种数据结构，其中栈内的元素保持单调递增或递减的顺序。这种数据结构的主要用途是解决一类问题，即寻找序列中某个元素左侧或右侧第一个比它大或小的元素。单调栈通过维护栈内元素的单调性，可以在线性时间复杂度O(n)内解决这类问题。

例题

给定一个整数数组 temperatures ，表示每天的温度，返回一个数组 answer ，其中 answer[i] 是指对于第 i 天，下一个更高温度出现在几天后。如果气温在这之后都不会升高，请在该位置用 0 来代替。

示例 1:
输入: temperatures = [73,74,75,71,69,72,76,73]
输出: [1,1,4,2,1,1,0,0]
示例 2:
输入: temperatures = [30,40,50,60]
输出: [1,1,1,0]
示例 3:
输入: temperatures = [30,60,90]
输出: [1,1,0]

```
class DailyTemperature(object):
    def dailyTemperatures(self, temperatures: list) -> list:
        """
        :type temperatures: List[int]
        :rtype: List[int]
        """
        length = len(temperatures)
        if length <= 0:
            return []
        if length == 1:
            return [0]
        # res用来存放第i天的下一个更高温度出现在几天后
        res = [0] * length
        # 定义一个单调栈
        stack = []
        for index in range(length):
            current = temperatures[index]
            # 栈不为空 且 当前温度大于栈顶元素
            while stack and current > temperatures[stack[-1]]:
                # 出栈
                pre_index = stack.pop()
                # 当前索引和出栈索引差即为出栈索引结果
                res[pre_index] = index - pre_index
            stack.append(index)
        return res


if __name__ == "__main__":
```

```
    demo = DailyTemperature()
    temperatures = [73, 74, 75, 71, 69, 72, 76, 73]
    print(demo.dailyTemperatures(temperatures))
```

队列 queue

可以理解为排队，我们只能在尾部添加或从头部移走元素。因此最先被添加的元素也会最先被移除，即**FIFO**(first-in-first-out)。主要操作包括Queue()、enqueue(item)、dequeue()、is_empty()、size()。

可以定义类实现，但也可以直接使用列表实现。

```
class Queue:
    def __init__(self):
        self.items = []
    def is_empty(self):
        return self.items == []
    def enqueue(self, item):
        self.items.insert(0, item)
    def dequeue(self):
        return self.items.pop()
    def size(self):
        return len(self.items)
```

```
queue = [] # =Stack()
stack.append(item)
stack.pop(0)
print(stack[-1])
print(not stack)
print(len(stack))
```

python中也可以导入库。

```
from pythonds.basic import Queue
```

双端队列 deque

可以在任意一端添加或移除元素。所以不要求按照**LIFO**和**FIFO**。

包括操作Deque()、add_front(item)、add_rear(item)、remove_front()、remove_rear()、is_empty()、size()操作。可以定义类，也可以使用列表实现。

```
class Deque:
    def __init__(self):
        self.items = []
    def isEmpty(self):
        return self.items == []
    def addFront(self, item):
        self.items.append(item)
    def addRear(self, item):
```

```
        self.items.insert(0, item)
    def removeFront(self):
        return self.items.pop()
    def removeRear(self):
        return self.items.pop(0)
    def size(self):
        return len(self.items)
```

```
deque = []
deque.insert(0,item)
deque.append(item)
deque.pop(0)
deque.pop()
print(not deque)
print(len(deque))
```

当然也可以导入库。

```
from pythonds.basic import Deque
```

树 tree

总算进入到树。树如其名，一棵树可以有根root，也可以有叶子children。

节点 Node：节点是树的基础部分。 每个节点具有名称，或“键值”。节点还可以保存额外数据项，数据项根据不同的应用而变。

边 Edge：边是组成树的另一个基础部分。 每条边恰好连接两个节点，表示节点之间具有关联，边具有出入方向； 每个节点（除根节点）恰有一条来自另一节点的入边； 每个节点可以有零条/一条/多条连到其它节点的出边。如果加限制不能有 “多条边”，这里树结构就特殊化为线性表

根节 Root: 树中唯一没有入边的节点。

路径 Path：由边依次连接在一起的有序节点列表。比如，哺乳纲→食肉目→猫科→猫属→家猫就是一条路径。

子节点 Children：入边均来自于同一个节点的若干节点，称为这个节点的子节点。

父节点 Parent：一个节点是其所有出边连接节点的父节点。

兄弟节点 Sibling：具有同一父节点的节点之间为兄弟节点。

子树 Subtree：一个节点和其所有子孙节点，以及相关边的集合。

叶节点or终端节点 Leaf Node：没有子节点的节点称为叶节点。

层级 Level： 从根节点开始到达一个节点的路径，所包含的边的数量，称为这个节点的层级。

堂兄弟节点：父节点在同一层的节点互为堂兄弟

节点的祖先：从根到该节点所经分支上的所有节点

子孙：以某节点为根的子树中任一节点都称为该节点的子孙

森林：由m（m>=0）棵互不相交的树的集合称为森林

重要概念

节点的度：一个节点含有的子树的个数称为该节点的度

树的度：一棵树中，最大的节点的度称为树的度

节点的层次：从根开始定义起，根为第1层，根的子节点为第2层，以此类推

树的高度或深度：树中节点的最大层次

树的实现

使用列表嵌套

```
tree = [
    1, # root
    [3,4,[
        5, # root
        [6,7]
    ]] # child
]
```

实际上表示的树是

```
   1
 / | \
3  4  5
     / \
    6   7
```

使用字典嵌套

```
tree = {
    1:{
        2:{
            4:{}
            5:{}
        }
        3:{}
    }
}
```

实际上表示的树是

```
    1
   / \
  2   3
 / \
4   5
```

定义类

```
class TreeNode:
    def __init__(self,data):
        self.data = data
        self.child = []

tree = TreeNode(1)
tree.child.append(TreeNode(2))
child_0 = tree.child[0]
child_0.child.append(TreeNode(4))
child_0.child.append(TreeNode(5))
tree.child.append(TreeNode(3))
```

树的遍历

对于普通的树，主要有前序遍历（先序遍历）、后序遍历和层次遍历三种。

现在我们以下面的树为例。

```
    1
   / \
  2   3
 / \
4   5
```

前序遍历/先序遍历

即先输出根，再遍历子节点。

```
# 树的每个节点都是TreeNode类
def preorder(tree):
    pre_list = [tree.data]

    for i in tree.child:
        pre_list += preorder(i)

    return pre_list
```

则例子遍历结果为：1 2 4 5 3

后序遍历

即先遍历子节点，再遍历根。

```
# 树的每个节点都是TreeNode类
def postorder(tree):
    post_list = []

    for i in tree.child:
        post_list += postorder(i)
    post_list.append(tree.data)

    return post_list
```

则例子遍历结果为：4 5 2 3 1

层次遍历 BFS

一层一层遍历。这种遍历方法可以和BFS联系起来。

```
# 树的每个节点都是TreeNode类
def layerorder(tree):
    layer_list = []
    queue = [tree]

    while queue:
        current = queue.pop(0)
        layer_list.append(current.data)
        for node in current.child:
            queue.append(node)

    return layer_list
```

遍历结果为：1 2 3 4 5

二叉树

二叉树是指每一个节点最多有两个子节点（左孩子右孩子）。

BinaryTree (二叉树)

二叉树是有限个元素的集合，该集合或者为空、或者有一个称为根节点（root）的元素及两个互不相交的、分别被称为左子树和右子树的二叉树组成。

二叉树的每个结点至多只有二棵子树(不存在度大于2的结点)，二叉树的子树有左右之分，次序不能颠倒。

二叉树的第i层至多有2^{i-1}个结点；

深度为k的二叉树至多有2^{k-1}个结点；

对任何一棵二叉树T，如果其终端结点数为N_0，度为2的结点数为N_2，则$N_0 = N_2 + 1$。

二叉树的遍历

二叉树的遍历分为以下三种：

先序遍历：遍历顺序规则为【根左右】

中序遍历：遍历顺序规则为【左根右】

后序遍历：遍历顺序规则为【左右根】

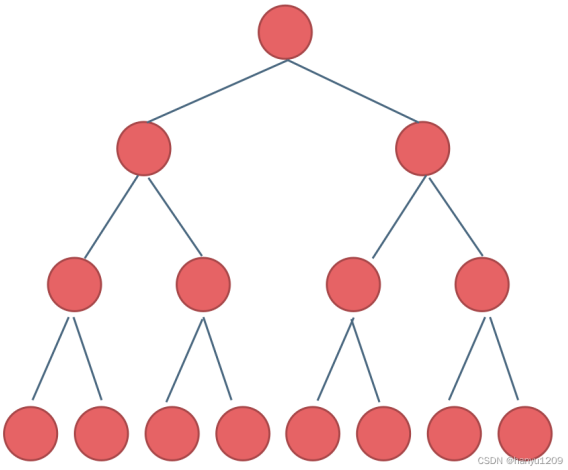
```
class TreeNode:
    def __init__(self, value):
        self.value = value
        self.left = None
        self.right = None

class Solution:
    # 先序
    def preorderTraversal(self, root: Optional[TreeNode])-> List[int]:
        result = []                         #保存结果
        def transfer(root):
            if root==None:
                return
            result.append(root.val)         #前序
            transfer(root.left)             #左
            transfer(root.right)            #右
        transfer(root)
        return result

    # 中序
    def inorderTraversal(self, root: Optional[TreeNode]) -> List[int]:
        result = []                         #存放结果
        def tranfer(root):
            if root==None:
                return
            tranfer(root.left)              #左
            result.append(root.val)         #中序
            tranfer(root.right)             #右
        tranfer(root)
        return result

    # 后序
    def postorderTraversal(self, root: Optional[TreeNode]) -> List[int]:
        result = []                        #存放结果
        def tranfer(root):
            if root==None:
                return
            tranfer(root.left)            #左
            tranfer(root.right)           #右
            result.append(root.val)       #后序
        tranfer(root)
        return result
```

“中+先”/“中+后”都可以确定二叉树的模样，“前+后”不可以

二叉树的种类

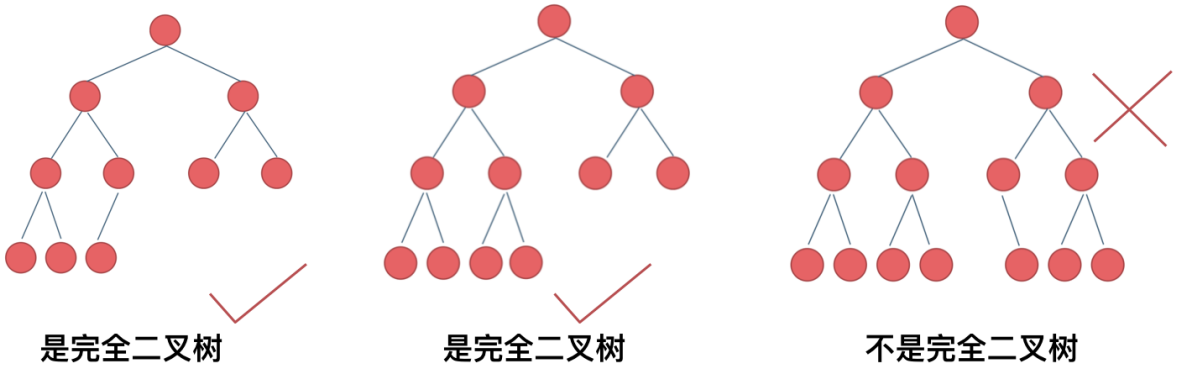
满二叉树

如果一棵二叉树**只有度为0的结点和度为2的结点**，并且**度为0的结点在同一层上**，则这棵二叉树为满二叉树。



完全二叉树

完全二叉树的定义如下：在完全二叉树中，**除了最底层节点可能没填满外**，其余每层节点数都达到最大值，并且最下面一层的节点都集中在该层最左边的若干位置。若最底层为第 h 层，则该层包含 1~ 2^{h-1} 个节点。



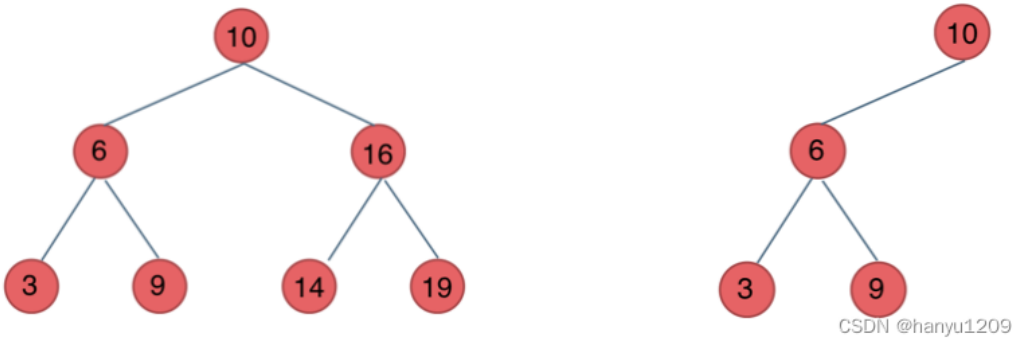
二叉搜索树

AVL平衡二叉搜索树

二叉搜索树

二叉搜索树是有数值的，**是一个有序树。**

- 若它的左子树不空，则左子树上所有结点的值均小于它的根结点的值；
- 若它的右子树不空，则右子树上所有结点的值均大于它的根结点的值；
- 它的左、右子树也分别为二叉排序树



基于二叉搜索树每个节点中的数值大小均满足：**左子节点的值 < 根节点的值 < 右子节点的值**的特点，二叉搜索树在搜索值时是依照二分来进行查找的，即先将待查找的值与每棵子树的根节点的值进行比较，当待查找的值小于每棵子树的根节点的值时，到该根节点的左子树递归查找；当待查找的值大于每棵子树的根节点的值时，到该根节点的右子树递归查找；待查找的值等于每棵子树的根节点的值时，即找到待查找的值所在的节点。除此之外，在二叉搜索树进行**删除节点，添加节点，修改节点**等操作时也会使用二分查找的思想。相比较于链表而言，二叉搜索树在搜索查找方面的效率要提高不少。

```
class Node:
    def __init__(self, val):           #定义树节点
        self.val = val
        self.lchild = None             #建立左子树
        self.rchild = None             #建立右子树

#定义二叉搜索树
class BST:
    def __init__(self, root = None):
        self.root = None
        self.size = 0

    #二叉搜索树判空
    def is_empty(self):
        return self.root == None

    #统计二叉搜索树的节点个数
    def size(self):
        return self.size

    #清空二叉搜索树
    def clear(self):
        self.root = None
```

添加节点

```
class BST:
    #添加节点
    def add_node(self, val):
        def recurse_node(node):
            if val < node.val:
                if not node.lchild:
                    node.lchild = Node(val)
                else:
                    recurse_node(node.lchild)
```

```
            elif not node.rchild:
                node.rchild = Node(val)
            else:
                recurse_node(node.rchild)
        if self.is_empty == 0:
            self.root = Node(val)
        else:
            recurse_node(self.root)
        self.size += 1
```

最坏情况：当树上的节点全都集中在根节点的某一侧时，二叉树在上就会成为一个单链结构，即添加操作的时间复杂度将会为 O(n)，其中 n 取决于二叉树上节点的个数。

一般情况：根据二叉树就结构的特征，其在添加操作上的时间复杂度和二分查找算法的时间复杂度相类似，即添加操作的平均时间复杂度为：O(log n)

查找操作

```
class BST:
    #二叉搜索树的查找
    def BST_Search(self, root, val):
        if not root:
            return None
        if val == root.val:
            return root
        elif val < root.val:
            return self.BST_Seach(root.lchild, val)
        else:
            return self.BST_Seach(root.rchild, val)
```

最坏情况：当二叉搜索树的所有节点都集中在根节点的一侧式，这棵树就会成为一个单链结构，故查找的最坏时间复杂度为：O(n) (其中 n 取决于树中节点的个数)

一般情况：二叉搜索树的查找是基于二分查找的思想实现的，故平均的时间复杂度为O(log n)

插入操作

```
class BST:
    #二叉搜索树的插入
    def BST_Insert(self, root, val):
        if root == None:
            return Node(val)
        if val < root.val:
            root.lchild = self.BST_Insert(root.lchild, val)
        if val > root.val:
            root.rchild = self.BST_Insert(root.rchild, val)
        return root
```

最坏情况：当所有的节点全部都集中在树根节点的一侧时，这棵树就会形成单链的结构，此时插入操作的时间复杂度为：O(n)，其中 n 取决于树上节点的个数。

一般情况：于上述的操作相类似，运用二分查找的思想，即二叉搜索树插入操作的一般时间复杂度为：O(log n)

创建二叉搜索树

```
class BST:
    #二叉搜索树的创建
    def BST_Build(self, nums):
        root = self.val
        for num in nums:
            self.BST_Insert(root, num)
        return root
```

删除操作

二叉搜索树的删除操作会有点复杂，这体现在，当我们找到待删除的节点时，需要对该节点的不同的情况进行讨论。

情况1：当待删除的节点没有左子节点时，则用该待删除节点的右子节点来代替删除的节点的位置

情况2：当待删除的节点没有右子节点时，则用该待删除节点的左子节点来代替删除的节点的位置

情况3：当待删除节点的左右子节点均存在时，基于二叉搜索树 左子节点的值 < 根节点的值 < 右子节点的值 的特点，我们先将一个 cur 指针指向待删除节点的右子节点，并使 cur 指针不断指向该右子节点的左子节点，当 cur 指针所指向的节点没有左子节点时，将删除节点的左子树全部添加到 cur 指针所指向的节点的左子节点。

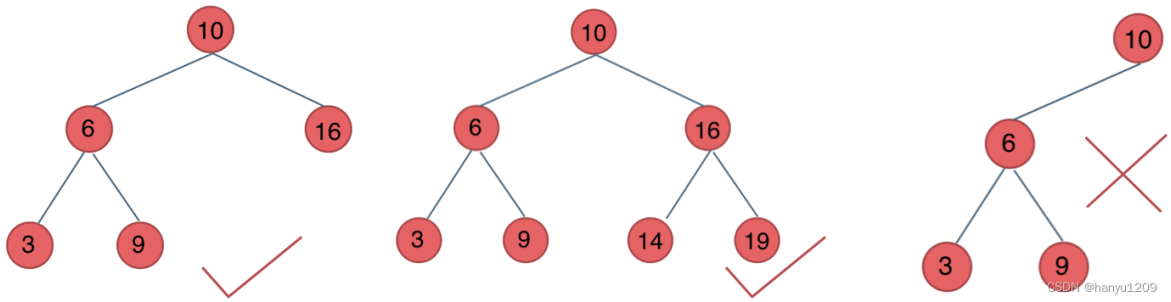
```
class BST:
    #二叉搜索树的删除
    def BST_delete(self, root, val):
        if not root:
            return root

        if root.val > val:
            root.lchild = self.BST_delete(root.lchild, val)
            return root
        elif root.val < val:
            root.rchild = self.BST_delete(root.rchild, val)
            return root
        else:
            if not root.lchild:
                return root.rchild
            elif not root.rchild:
                return root.lchild
            else:
                cur = root.rchild
                while cur.lchild:
                    cur = cur.lchild
                cur.lchild = root.lchild
                return root.rchild
```

最坏情况：当树上的节点均集中在其根节点的用一侧时，这棵树将会形成一个单链的结构，故此时删除操作的时间复杂度为：O(n)，其中 n 取决于树中节点的个数。

一般情况：在二叉搜索树进行删除操作时，会优先查找要进行删除的节点，故删除操作的时间复杂度也为 O(log n)

平衡搜索二叉树

平衡二叉搜索树：又被称为**AVL（Adelson-Velsky and Landis）树**，且具有以下性质：它是一棵空树或它的左右两个子树的高度差的绝对值不超过1，并且左右两个子树都是一棵平衡二叉树。



节点定义

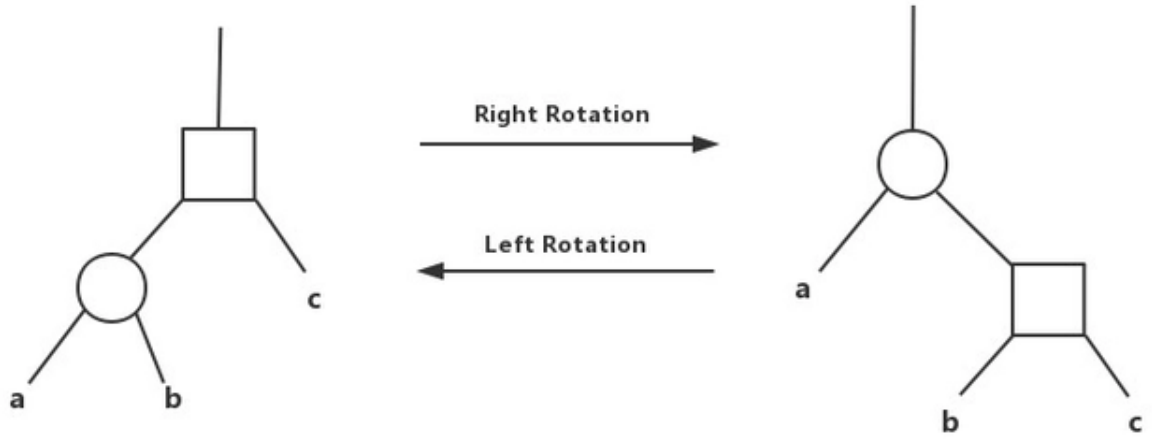
```
class AVLNode:
    def __init__(self, key):
        self.key = key
        self.height = 1
        self.left = None
        self.right = None
```

AVL树的节点除了包含值之外，还记录了节点的高度。这个高度信息是维持平衡的关键。

辅助函数

```
def get_height(node):
    if node is None:
        return 0
    return node.height

def get_balance(node):
    if node is None:
        return 0
    return get_height(node.left) - get_height(node.right)
```

几种旋转方式



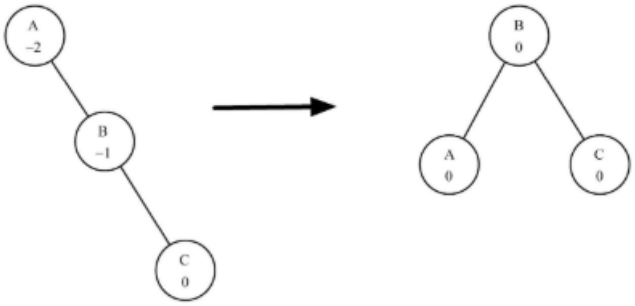


图6-28 通过左旋让失衡的树恢复平衡

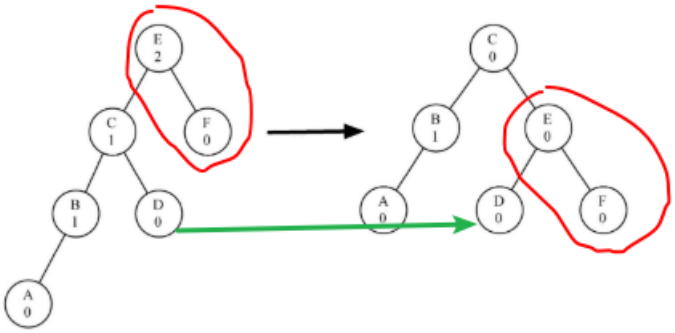


图6-29 通过右旋让失衡的树恢复平衡

RR：左旋

LL：右旋

RL：先变为RR（可认为是下半部分LL，需右旋），然后再左旋

LR：先变为LL（可认为是下半部分RR，需左旋），然后再右旋

```
def rotate_left(z): # RR
    y = z.right
    T2 = y.left

    # 执行左旋
    y.left = z
    z.right = T2

    # 更新节点的高度
    z.height = 1 + max(get_height(z.left), get_height(z.right))
    y.height = 1 + max(get_height(y.left), get_height(y.right))

    return y

def rotate_right(y): # LL
    x = y.left
    T2 = x.right

    # 执行右旋
    x.right = y
    y.left = T2

    # 更新节点的高度
    y.height = 1 + max(get_height(y.left), get_height(y.right))
    x.height = 1 + max(get_height(x.left), get_height(x.right))
```

```
    return x
```

插入操作

插入操作是在AVL树中插入新节点的过程，同时需要保持树的平衡。插入后，我们需要更新节点的高度，并进行旋转操作来恢复平衡。

```
def insert(root, key):
    if root is None:
        return AVLNode(key)

    if key < root.key:
        root.left = insert(root.left, key)
    elif key > root.key:
        root.right = insert(root.right, key)

    # 更新节点的高度
    root.height = 1 + max(get_height(root.left), get_height(root.right))

    # 获取平衡因子
    balance = get_balance(root)

    # 进行旋转操作来恢复平衡
    # 左旋
    if balance > 1 and key < root.left.key:
        return rotate_right(root)
    # 右旋
    if balance < -1 and key > root.right.key:
        return rotate_left(root)
    # 左右双旋
    if balance > 1 and key > root.left.key:
        root.left = rotate_left(root.left)
        return rotate_right(root)
    # 右左双旋
    if balance < -1 and key < root.right.key:
        root.right = rotate_right(root.right)
        return rotate_left(root)

    return root
```

删除操作

删除操作是在AVL树中删除节点的过程，同时需要保持树的平衡。删除后，我们需要更新节点的高度，并进行旋转操作来恢复平衡。

```
def delete(root, key):
    if root is None:
        return root

    if key < root.key:
```

```
            root.left = delete(root.left, key)
        elif key > root.key:
            root.right = delete(root.right, key)
        else:
            # 节点有一个或没有子节点
            if root.left is None:
                return root.right
            elif root.right is None:
                return root.left

            # 节点有两个子节点，找到右子树的最小节点
            root.key = find_min(root.right).key
            # 删除右子树的最小节点
            root.right = delete(root.right, root.key)

        # 更新节点的高度
        root.height = 1 + max(get_height(root.left), get_height(root.right))

        # 获取平衡因子
        balance = get_balance(root)

        # 进行旋转操作来恢复平衡
        # 左旋
        if balance > 1 and get_balance(root.left) >= 0:
            return rotate_right(root)
        # 右旋
        if balance < -1 and get_balance(root.right) <= 0:
            return rotate_left(root)
        # 左右双旋
        if balance > 1 and get_balance(root.left) < 0:
            root.left = rotate_left(root.left)
            return rotate_right(root)
        # 右左双旋
        if balance < -1 and get_balance(root.right) > 0:
            root.right = rotate_right(root.right)
            return rotate_left(root)

        return root
```

霍夫曼编码树

哈夫曼树（Huffman Tree）和哈夫曼编码（Huffman Coding）是一种用于数据压缩的技术，由David A. Huffman于1952年提出。哈夫曼树是一种特殊的二叉树，用于构建哈夫曼编码。哈夫曼编码是一种变长编码，用于将字符映射到不同长度的比特串，以实现数据的高效压缩。

带权路径最短的二叉树称为哈夫曼树或最优二叉树。

即权值越大的节点离根节点越近。
构建哈夫曼树的基本思想是：从所有权值为正整数的叶子节点开始，**每次选取两个权值最小的节点合并成一个新的节点**，直到最后只剩下一个根节点为止。

哈夫曼树是一棵**满二叉树**（所有非叶子节点的度都是2）。

树中的**每个叶子节点代表一个字符**，其权值即为字符出现的频率。

哈夫曼树的带权路径长度（WPL）最小，即每个字符的编码长度最短。

实现

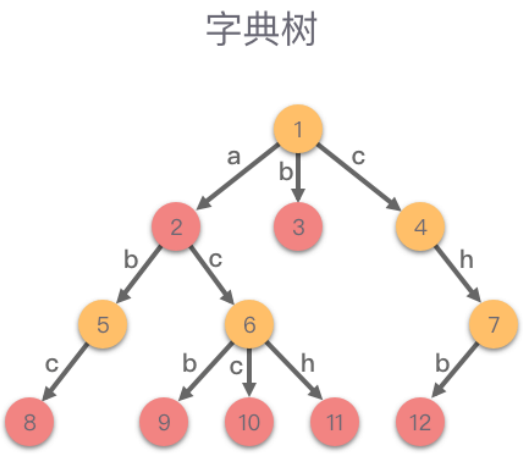
```
import heapq
from collections import defaultdict

class TreeNode:
    def __init__(self, char, freq):
        self.char = char
        self.freq = freq
        self.left = None
        self.right = None

    def __lt__(self, other):
        return self.freq < other.freq

def build_huffman_tree(text):
    # 统计字符频率
    freq_map = defaultdict(int)
    for char in text:
        freq_map[char] += 1

    # 创建优先队列（最小堆）用于构建哈夫曼树
    priority_queue = [TreeNode(char, freq) for char, freq in freq_map.items()]
    heapq.heapify(priority_queue)

    # 构建哈夫曼树
    while len(priority_queue) > 1:
        left = heapq.heappop(priority_queue)
        right = heapq.heappop(priority_queue)
        merged_node = TreeNode(None, left.freq + right.freq)
        merged_node.left = left
        merged_node.right = right
        heapq.heappush(priority_queue, merged_node)

    # 返回哈夫曼树的根节点
    return priority_queue[0]

def generate_huffman_codes(root):
    def dfs(node, code, result):
        if node:
            if node.char:
                result[node.char] = code
            dfs(node.left, code + '0', result)
            dfs(node.right, code + '1', result)

    huffman_codes = {}
    dfs(root, '', huffman_codes)
    return huffman_codes

# 测试
text = "ABRACADABRA"
huffman_tree = build_huffman_tree(text)
huffman_codes = generate_huffman_codes(huffman_tree)
```

```
print("Huffman Codes:")
for char, code in huffman_codes.items():
    print(f"{char}: {code}")
```

字典树Trie

是一种用于**处理字符串集合**的树形数据结构。它通过将字符串的每个字符存储在节点中，形成树状结构，具有高效的插入、查找和删除操作。

例如下图就是一棵字典树，其中包含有 a、abc、acb、acc、ach、b、chb 这 7 个单词。



从图中可以发现，**这棵字典树用边来表示字母，从根节点到树上某一节点的路径就代表了一个单词。**比如 1 → 2 → 6 → 10 表示的就是单词 acc。为了清楚地标记单词，**我们可以在每个单词的结束节点位置增加一个 end 标记（图中红色节点），表示从根节点到这里有一个单词。**

字典树的结构比较简单，其本质上就是一个用于字符串快速检索的多叉树，树上每个节点都包含多字符指针。将从根节点到某一节点路径上经过的字符连接起来，就是该节点对应的字符串。

字典树设计的核心思想：利用空间换时间，利用字符串的公共前缀来降低查询时间的开销，最大限度的减少无谓的字符串比较，以达到提高效率的目的。

字典树的基本操作有 **创建、插入、查找 和 删除**。这里主要介绍字典树的创建、插入和查找。

定义节点

```
class Node:                                          # 字符节点
    def __init__(self):                              # 初始化字符节点
        self.children = [None for _ in range(26)]     # 初始化子节点
        self.isEnd = False                           # isEnd 用于标记单词结束
```

定义Trie类

```
class Trie:                                          # 字典树
    # 初始化字典树
    def __init__(self):                              # 初始化字典树
        self.root = Node()                           # 初始化根节点（根节点不保存字符）
```

插入操作

1)依次遍历单词中的字符 ch，并从字典树的根节点的子节点位置开始进行插入操作（根节点不包含字符）。

2)如果当前节点的子节点中，不存在键为 ch 的节点，则建立一个节点，并将其保存到当前节点的子节点中，即 cur.children[ch] = Node()，然后令当前节点指向新建立的节点，然后继续处理下一个字符。

4)如果当前节点的子节点中，存在键为 ch 的节点，则直接令当前节点指向键为 ch 的节点，继续处理下一个字符。

5)在单词处理完成时，将当前节点标记为单词结束。

```
class Trie:
    # 向字典树中插入一个单词
    def insert(self, word: str) -> None:
        cur = self.root
        for ch in word:                             # 遍历单词中的字符
            if ch not in cur.children:              # 如果当前节点的子节点中，不存在键为 ch 的节点
                cur.children[ch] = Node()           # 建立一个节点，并将其保存到当前节点的子节点
            cur = cur.children[ch]                  # 令当前节点指向新建立的节点，继续处理下一个字符
        cur.isEnd = True                            # 单词处理完成时，将当前节点标记为单词结束
```

时间复杂度为 $O(n)$；如果使用数组，则空间复杂度为 $O(d^n)$，如果使用哈希表实现，则空间复杂度为 $O(n)$。

创建字典树

```
trie = Trie()
for word in words:
    trie.insert(word)
```

查找操作

依次遍历单词中的字符，并从字典树的根节点位置开始进行查找操作。

如果当前节点的子节点中，不存在键为 ch 的节点，则说明不存在该单词，直接返回 False。

如果当前节点的子节点中，存在键为 ch 的节点，则令当前节点指向新建立的节点，然后继续查找下一个字符。

在单词处理完成时，判断当前节点是否有单词结束标记，如果有，则说明字典树中存在该单词，返回 True。否则，则说明字典树中不存在该单词，返回 False。

```python
class Trie:
    # 查找字典树中是否存在一个单词
    def search(self, word: str) -> bool:
        cur = self.root
        for ch in word:                         # 遍历单词中的字符
            if ch not in cur.children:          # 如果当前节点的子节点中，不存在键为 ch 的节点
                return False                    # 直接返回 False
            cur = cur.children[ch]              # 令当前节点指向新建立的节点，然后继续查找下一个字符

        return cur is not None and cur.isEnd # 判断当前节点是否为空，并且是否有单词结束标记
```

时间复杂度为 $O(n)$；空间复杂度为 $O(1)$。

查找前缀

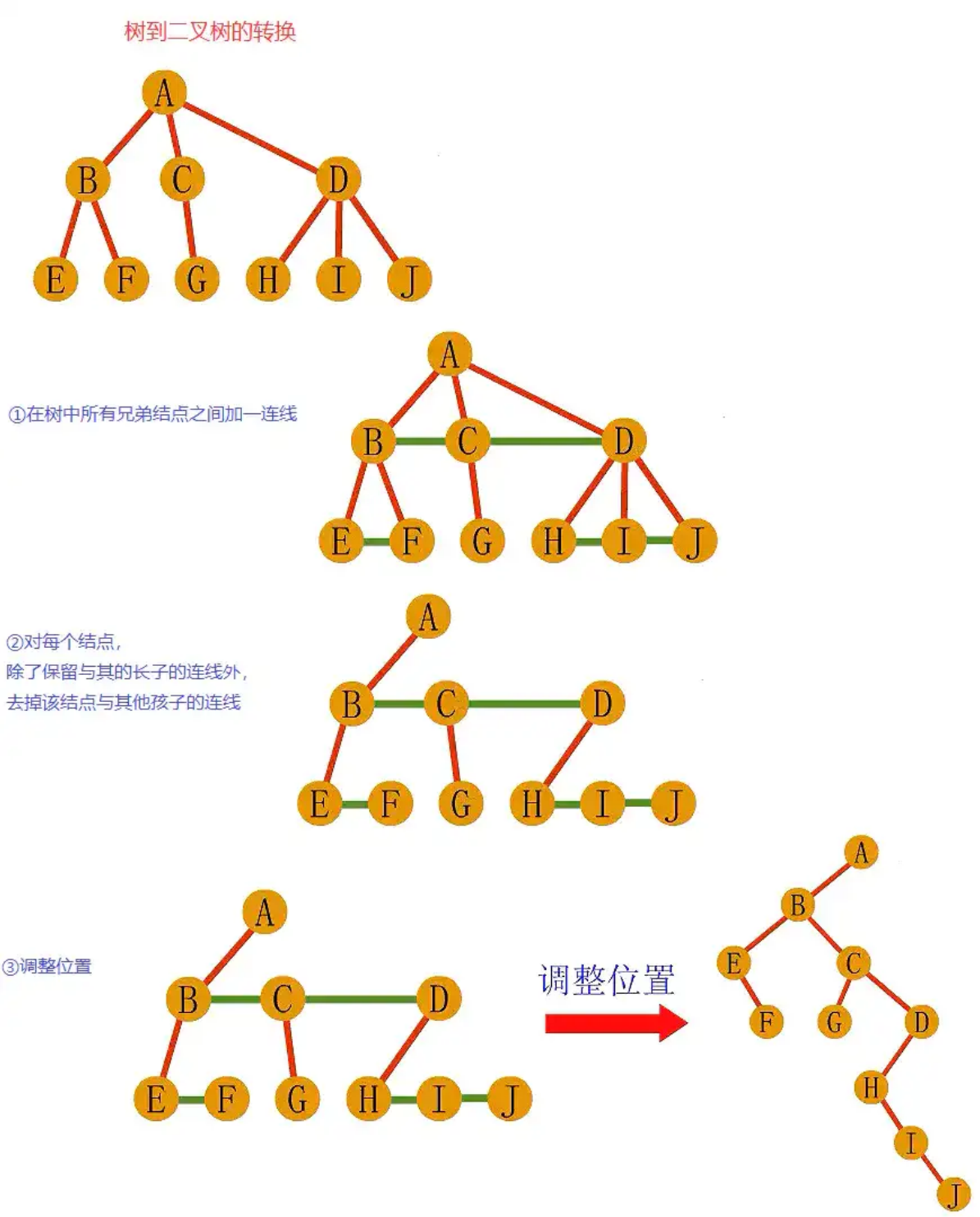
```python
class Trie:
    # 查找字典树中是否存在一个前缀
    def startsWith(self, prefix: str) -> bool:
        cur = self.root
        for ch in prefix:                       # 遍历前缀中的字符
            if ch not in cur.children:          # 如果当前节点的子节点中，不存在键为 ch 的节点
                return False                    # 直接返回 False
            cur = cur.children[ch]              # 令当前节点指向新建立的节点，然后继续查找下一个字符
        return cur is not None                  # 判断当前节点是否为空，不为空则查找成功
```

时间复杂度为 $O(m)$；空间复杂度为 $O(1)$。

树与二叉树的转换

普通树、森林->二叉树：左孩子右兄弟

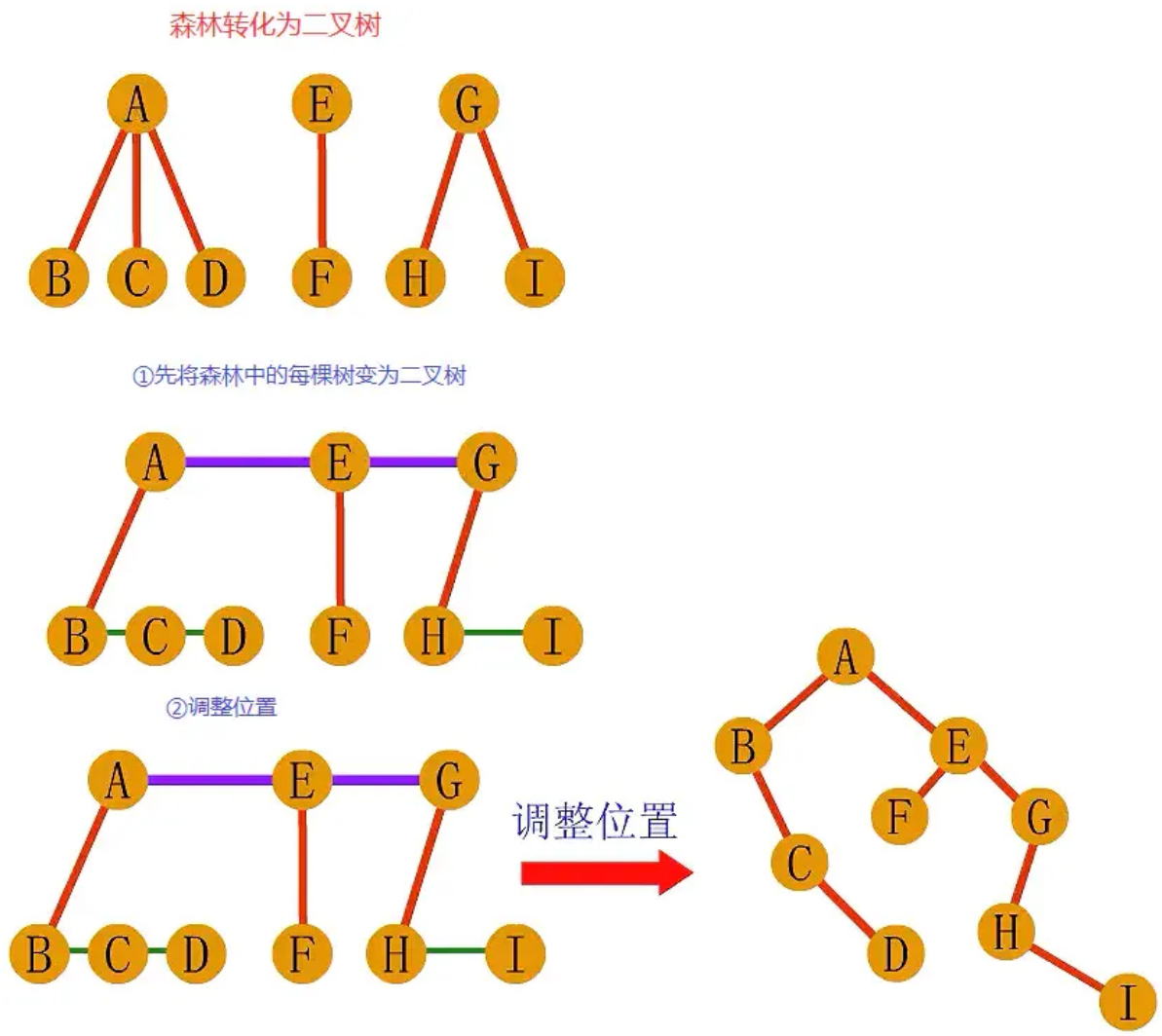
树到二叉树



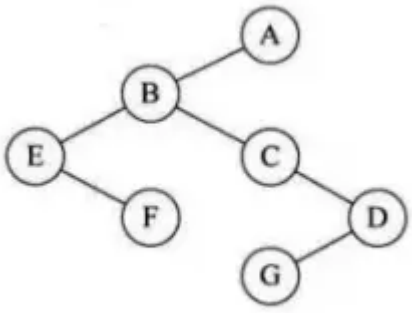

```
class Tree:
    def __init__(self,key):
        self.root = key
        self.child = []

    def get_height(self):
        height = 0
        for i in self.child:
            height = max(height,i.get_height()+1)
        return height

class BinaryTree:
```

```
    def __init__(self,key):
        self.root = key
        self.left = None
        self.right = None

    def get_height(self):
        if self.left is None:
            height1 = 0
        else:
            height1 = self.left.get_height()+1
        if self.right is None:
            height2 = 0
        else:
            height2 = self.right.get_height()+1
        return max(height1,height2)

def build_tree(s):
    stack = []
    tree1 = Tree(0)
    current = tree1
    for i in s:
        if i == 'd':
            new_node = Tree(0)
            current.child.append(new_node)
            stack.append(current)
            current = new_node
        if i == 'u':
            current = stack.pop()
    return tree1

def Tree2BinaryTree(tree:Tree):
    new_tree = BinaryTree(None)
    new_tree.root = tree.root

    if not tree.child:
        return new_tree
    elif len(tree.child) == 1:
        new_tree.left = Tree2BinaryTree(tree.child[0])
        return new_tree
    else:
        new_tree.left = Tree2BinaryTree(tree.child[0])
        current = new_tree.left
        for i in range(1,len(tree.child)):
            current.right = Tree2BinaryTree(tree.child[i])
            current = current.right
        return new_tree

s = input()
tree1 = build_tree(s)
tree2 = Tree2BinaryTree(tree1)
print(f'{tree1.get_height()} => {tree2.get_height()}')
```

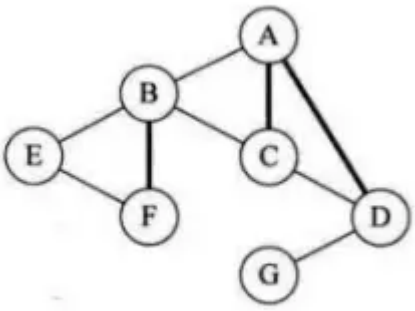
森林到二叉树


森林转化为二叉树
A
E
G
B
C
D
F
H
I
①先将森林中的每棵树变为二叉树
A
E
G
B
C
D
F
H
I
②调整位置
A
E
G
B
C
D
F
H
I
调整位置
A
B
E
F
G
C
H
D
I

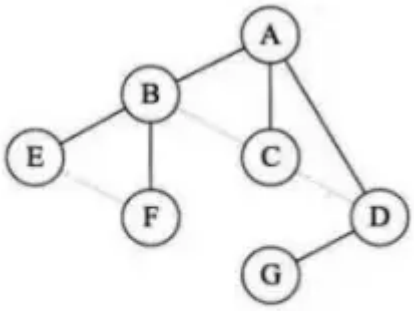
二叉树到树



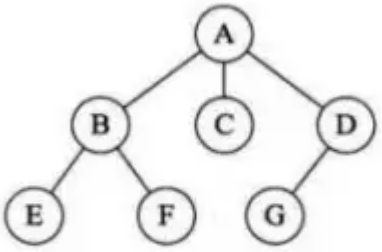
二叉树



步骤1：加线



步骤2：去线

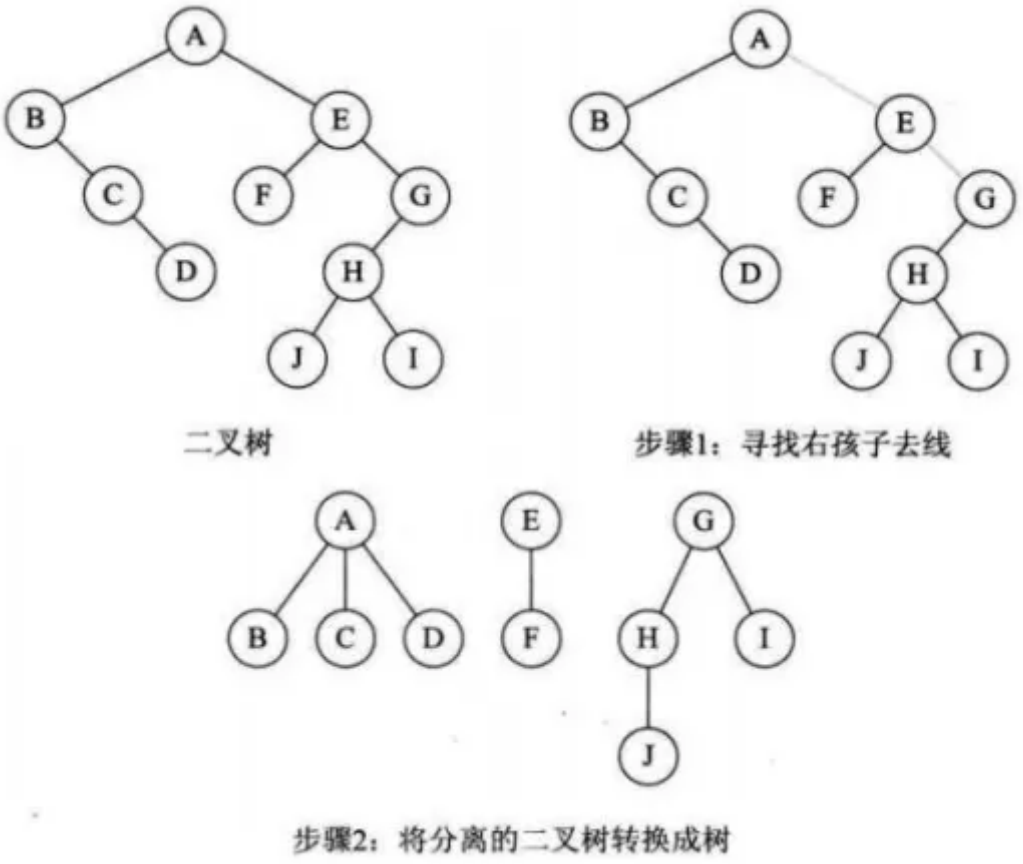


步骤3：层次调整

```
class Tree:
    def __init__(self,key):
        self.key = key
        self.children = []

def layer_overturn(tree:Tree):
    lst = []
    queue = [tree]
    while queue:
        current = queue.pop(0)
        if isinstance(current,Tree):
            lst.append(current.key)
            for i in current.children[::-1]:
                queue.append(i)
        else:
            lst.append(current)
    print(*lst)
class BinaryTree:
    def __init__(self,key):
        self.root = key
        self.left = None
        self.right = None

def build_binary(tempList):
    binary = BinaryTree(None)
    stack = []
    current = binary
    for i in tempList:
```

```
        s, s1 = i[0], i[1]
        if s1 == '0':
            if current.root is None:
                current.root = s
            else:
                while True:
                    if current.left is None:
                        current.left = BinaryTree(s)
                        stack.append(current)
                        current = current.left
                        break
                    elif current.right is None:
                        current.right = BinaryTree(s)
                        stack.append(current)
                        current = current.right
                        break
                    else:
                        current = stack.pop()

        else:
            while True:
                if current.left is None:
                    current.left = s
                    break
                elif current.right is None:
                    current.right = s
                    if stack: current = stack.pop()
                    break
                else:
                    current = stack.pop()
    return binary

def binary2tree(binary:BinaryTree):
    tree = Tree(None)
    tree.key = binary.root
    current = binary.left

    while isinstance(current,BinaryTree):
        a = current.right
        current.right = None
        tree.children.append(binary2tree(current))
        current = a

    if current != '$' and current is not None:
        tree.children.append(current)
    return tree

n = int(input())
if n == 1:
    binary = BinaryTree(input()[0])
else:
    tempList = list(input().split())
    binary = build_binary(tempList)
tree = binary2tree(binary)
layer_overturn(tree)
```

二叉树转换为森林



相关概念：**m阶B树**

B树（B-Tree）

B树，又称**多路平衡查找树**，B树中**所有结点的孩子结点数的最大值称为B树的阶**，通常用m表示。一棵m阶B树或为空树，或为满足如下特性的m叉树:

1）树中每个结点至多有m棵子树（即至多含有m-1个关键字）。

2）**若根结点不是终端结点，则至少有两棵子树。**

3）**除根结点外的所有非叶结点至少有[m/2]棵子树（即至少含有[m/2]-1个关键字）**

4）**所有叶子节点都在同一层。**

5）**每个节点中的关键字按照升序排列。**

优点:

1）减少访问磁盘的次数：B树的每个节点可以存储更多的关键字，因此树的高度相对较低，从而减少了访问磁盘的次数。

2）适应不同的数据规模：B树可以根据数据规模动态调整节点大小，适应不同的数据规模。

实现

定义B树节点类：B树的节点需要存储关键字和子节点的信息。我们可以定义一个节点类，其中包含关键字列表和子节点列表。

插入操作：B树的插入操作需要保持树的平衡性。当插入一个关键字时，需要根据B树的特性将关键字插入到合适的位置，并可能进行节点的分裂和合并操作，以维持B树的平衡性。

删除操作：B树的删除操作也需要保持树的平衡性。当删除一个关键字时，需要根据B树的特性对节点进行合并和移动操作，以维持B树的平衡性。

```
class BTreeNode:
    def __init__(self, leaf=False):
        self.keys = []
        self.children = []
        self.leaf = leaf
    # BTreeNode类：定义了B树的节点类，包含关键字列表 keys 和子节点列表 children，以及一个标志位 leaf 表示是否为叶子节点。

class BTree:
    def __init__(self, t):
        self.root = BTreeNode()
        self.t = t

    def insert(self, key):
        if len(self.root.keys) == (2 * self.t) - 1:
            new_root = BTreeNode()
            new_root.children.append(self.root)
            self.split_child(new_root, 0)
            self.root = new_root
        self._insert(self.root, key)

    def _insert(self, node, key):
        if node.leaf:
            i = 0
            while i < len(node.keys) and key > node.keys[i]:
                i += 1
            node.keys.insert(i, key)
        else:
            i = 0
            while i < len(node.keys) and key > node.keys[i]:
                i += 1
            if len(node.children[i].keys) == (2 * self.t) - 1:
                self.split_child(node, i)
                if key > node.keys[i]:
                    i += 1
            self._insert(node.children[i], key)

    def split_child(self, parent, index):
        t = self.t
        child = parent.children[index]
        new_child = BTreeNode(leaf=child.leaf)
        parent.keys.insert(index, child.keys[t - 1])
        parent.children.insert(index + 1, new_child)
        new_child.keys = child.keys[t:]
        child.keys = child.keys[:t - 1]
```

```
        if not child.leaf:
            new_child.children = child.children[t:]
            child.children = child.children[:t]

    def __str__(self):
        return self.print_tree(self.root)

    def print_tree(self, node, level=0):
        ret = ""
        if node:
            ret += self.print_tree(node.children[-1], level + 1)
            for i in range(len(node.keys) - 1, -1, -1):
                ret += "\n" + ("    " * level) + str(node.keys[i])
                ret += self.print_tree(node.children[i], level + 1)
        return ret
    # BTree类：定义了B树类，包含了B树的插入操作 insert、节点分裂操作 split_child，以及辅助方法 _insert 和打印方法 print_tree。
    # insert方法：首先判断根节点是否已满，如果是则分裂根节点；然后调用辅助方法 _insert 插入关键字。
    # _insert方法：递归地在合适的位置插入关键字，并在需要时进行节点分裂。
    # split_child方法：分裂节点，将中间的关键字提升到父节点，并将节点分裂成两个节点。

# 测试
btree = BTree(2)
keys = [3, 7, 1, 4, 9, 2, 6, 5, 8]
for key in keys:
    btree.insert(key)
print(btree)
```

B+树（B-Plus Tree）

B+树是在B树的基础上进行改进的一种树结构，它与B树的区别在于：

所有关键字都出现在叶子节点中，而非内部节点。
内部节点仅用于索引，不存储数据，叶子节点包含了所有数据项。

B+树的特点包括：
叶子节点形成了有序链表，可以支持范围查找和范围查询。
内部节点不存储数据，只存储索引，因此可以存储更多的关键字。
由于关键字只出现在叶子节点中，因此B+树的查找性能更加稳定。

实现

定义B+树节点类：B+树的节点需要存储索引信息和叶子节点指针。我们可以定义一个节点类，其中包含关键字列表、子节点列表和叶子节点指针。

插入操作：B+树的插入操作与B树类似，但是需要额外处理叶子节点之间的连接关系，以保持叶子节点形成的有序链表。

删除操作：B+树的删除操作也与B树类似，但是同样需要额外处理叶子节点之间的连接关系。

```
class BPlusTreeNode:
    def __init__(self, leaf=False):
        self.keys = []
```

```
        self.children = []
        self.next_leaf = None  # 指向下一个叶子节点
        self.leaf = leaf
    # BPlusTreeNode类：定义了B+树的节点类，与B树节点类相似，但是多了一个指向下一个叶子节点的指针 next_leaf。

class BPlusTree:
    def __init__(self, t):
        self.root = BPlusTreeNode(leaf=True)
        self.t = t

    def insert(self, key):
        if len(self.root.keys) == (2 * self.t) - 1:
            new_root = BPlusTreeNode()
            new_root.children.append(self.root)
            self.split_child(new_root, 0)
            self.root = new_root
        self._insert(self.root, key)

    def _insert(self, node, key):
        if node.leaf:
            i = 0
            while i < len(node.keys) and key > node.keys[i]:
                i += 1
            node.keys.insert(i, key)
        else:
            i = 0
            while i < len(node.keys) and key > node.keys[i]:
                i += 1
            if len(node.children[i].keys) == (2 * self.t) - 1:
                self.split_child(node, i)
                if key > node.keys[i]:
                    i += 1
            self._insert(node.children[i], key)

    def split_child(self, parent, index):
        t = self.t
        child = parent.children[index]
        new_child = BPlusTreeNode(leaf=child.leaf)
        parent.keys.insert(index, child.keys[t - 1])
        parent.children.insert(index + 1, new_child)
        new_child.keys = child.keys[t:]
        child.keys = child.keys[:t - 1]
        if not child.leaf:
            new_child.children = child.children[t:]
            child.children = child.children[:t]

    def __str__(self):
        return self.print_tree(self.root)

    def print_tree(self, node, level=0):
        ret = ""
        if node:
            ret += self.print_tree(node.children[0], level + 1)
            for i in range(len(node.keys)):
                ret += "\n" + ("    " * level) + str(node.keys[i])
```

```
                ret += self.print_tree(node.children[i + 1], level + 1)
        return ret
    # BPlusTree类：定义了B+树类，与B树类相似，但是插入和分裂操作需要额外处理叶子节点之间的连接关系。
    # insert方法：与B树的插入操作类似，但是需要在插入关键字时维护叶子节点之间的连接关系。
    # split_child方法：与B树的节点分裂操作类似，但是需要额外维护叶子节点之间的连接关系。

# 测试
bplustree = BPlusTree(2)
keys = [3, 7, 1, 4, 9, 2, 6, 5, 8]
for key in keys:
    bplustree.insert(key)
print(bplustree)
```

图 graph

写了好久的树，可算是到图了。在开始讲解之前，我们需要考虑一个事情，即，什么是图?

简单地说，图可以表示不同节点之间的关系。图有顶点和边组成，顶点用有穷非空集合`V(G)={v1,v2,...,vn}`表示，顶点之间的边用集合`E(G)={(u,v)|u∈V,v∈V}`表示，图可以表示为：`G=(V,E)`。其中G表示图，V表示顶点，E表示边。**|V|表示顶点的个数，也称图的阶，|E|表示边的条数。**

比较：

在线性表中，**数据元素之间是被串起来的，仅有线性关系**，每个数据元素最多只有一个直接前驱和一个直接后继。

在树形结构中，**数据元素之间有着明显的层次关系**，并且每一层上的数据元素可能和下一层中多个元素相关，但只能和上一层中一个元素相关。

图是一种较线性表和树更加复杂的数据结构。**在图形结构中，结点之间的关系可以是任意的，图中任意两个数据元素之间都可能相关，且相互之间没有前后上下之分都是同级关系。**

图可以根据是否有向，边是否带有权值，是否有重边，是否有环分类。

图的表示

邻接矩阵

邻接矩阵是一个二维数组，其中的元素`matrix[i][j]`表示节点`i`和节点`j`之间是否存在边。对于有权图，**矩阵的元素可以表示边的权重。**

```
adj_matrix = [[0] * vertices for _ in range(vertices)]
adj_matrix[0][1] = 1
adj_matrix[0][2] = 1
adj_matrix[1][3] = 1
adj_matrix[2][4] = 1
```

邻接表

邻接表使用字典或哈希表来表示图，其中每个节点对应一个链表，存储与该节点相邻的节点及边的信息。

```
from collections import defaultdict
class Graph:
    def __init__(self):
        self.adj_list = defaultdict(list)

    def add_edge(self, start, end, weight=1):
        self.adj_list[start].append((end, weight))
        self.adj_list[end].append((start, weight))  # 无向图需要考虑反向

graph = Graph()
graph.add_edge(0, 1)
graph.add_edge(0, 2)
graph.add_edge(1, 3)
graph.add_edge(2, 4)
```

图的遍历

两种：BFS和DFS。

深度优先搜索（DFS）

深度优先搜索从起始节点开始，尽可能深地访问图的分支，直到无法继续为止，然后回溯到上一个节点，继续深度优先搜索。

```
def dfs(graph, start, visited=None):
    if visited is None:
        visited = set()
    visited.add(start)
    print(start, end=" ")
    for neighbor, _ in graph.adj_list[start]:
        if neighbor not in visited:
            dfs(graph, neighbor, visited)

dfs(graph, 0)
```

广度优先搜索（BFS）

广度优先搜索从起始节点开始，首先访问其所有邻居节点，然后逐层扩展，直到图中所有节点都被访问。

```
from collections import deque

def bfs(graph, start):
    visited = set()
    queue = deque([start])
    visited.add(start)
    print(start, end=" ")

    while queue:
```

```
        current = queue.popleft()
        for neighbor, _ in graph.adj_list[current]:
            if neighbor not in visited:
                queue.append(neighbor)
                visited.add(neighbor)
                print(neighbor, end=" ")

# 示例
bfs(graph, 0)
```

相关算法

Warnsdorff算法

实际上是先走**未来可走选项少**的节点。

骑士周游

```
class Vertex:
    def __init__(self,id):
        self.id = id
        self.connectedTo = []
        self.color = 'white'
def pos_to_node_id(x, y, bdSize):
    return x * bdSize + y

def gen_legal_moves(row, col, board_size):
    new_moves = []
    move_offsets = [(-1, -2),(-1, 2),(-2, -1),(-2, 1),
                    (1, -2),(1, 2),(2, -1),(2, 1)]
    for r_off, c_off in move_offsets:
        if 0 <= row + r_off < board_size and 0 <= col + c_off < board_size:
            new_moves.append((row + r_off, col + c_off))
    return new_moves

def knight_graph(board_size):
    kt_graph = {}
    for row in range(board_size):
        for col in range(board_size):
            node_id = pos_to_node_id(row, col, board_size)
            kt_graph[node_id] = Vertex(node_id)
            new_positions = gen_legal_moves(row, col, board_size)
            for row2, col2 in new_positions:
                other_node_id = pos_to_node_id(row2, col2, board_size)
                kt_graph[node_id].connectedTo.append(other_node_id)
    return kt_graph

def ordered_by_avail(n:Vertex):
    res_list = []
    for v in n.connectedTo:
        if kt_graph[v].color == "white":
            c = 0
            for w in kt_graph[v].connectedTo:
                if kt_graph[w].color == "white":
                    c += 1
```

```
            res_list.append((c,kt_graph[v]))
    res_list.sort(key=lambda x: x[0])
    return [y[1] for y in res_list]

def knight_tour(n, path, u:Vertex, limit):
    u.color = "gray"
    path.append(u)
    if n < limit:
        neighbors = ordered_by_avail(u) #对所有的合法移动依次深入
        i = 0
        for nbr in neighbors:
            if nbr.color == "white" and knight_tour(n + 1, path, nbr, limit):
                return True
        else:
            path.pop()
            u.color = "white"
            return False
    else:
        return True

n = int(input())
kt_graph = knight_graph(n)
x,y = map(int,input().split())
start = kt_graph[x*n+y]
done = knight_tour(0, [], start, n*n-1)
if done:
    print("success")
else:
    print("fail")
```

Dijkstra

鸣人和佐助

```
import heapq
import math
def dijkstra(m,n,x1,y1,x2,y2,path:list):
    moves = [(0,-1),(0,1),(-1,0),(1,0)]
    distances = [[math.inf]*n for _ in range(m)]
    pos = []
    if path[x1][y1] == '#' or path[x2][y2] == '#':
        return 'NO'
    distances[x1][y1] = 0
    heapq.heappush(pos,(0,x1,y1))
    while pos:
        cur_d,x,y = heapq.heappop(pos)
        if x == x2 and y == y2:
            return cur_d
        h = int(path[x][y])
        for nx,ny in moves:
            new_x,new_y = nx+x,ny+y
            if 0 <= new_x < m and 0 <= new_y < n and path[new_x][new_y] != '#':
                if distances[new_x][new_y] > cur_d+abs(int(path[new_x][new_y])-
h):
```

```
                    distances[new_x][new_y] = cur_d+abs(int(path[new_x][new_y])-
h)
                    heapq.heappush(pos,(distances[new_x][new_y],new_x,new_y))
    return 'NO'

m, n, p = map(int, input().split())
path = [input().split() for _ in range(m)]

for _ in range(p):
    a, b, c, d = map(int,input().split())
    print(dijkstra(m,n,a,b,c,d,path))
```

兔子与樱花

```
import heapq
import math
def dijkstra(graph,start,end,P):
    if start == end: return []
    dist = {i:(math.inf,[]) for i in graph}
    dist[start] = (0,[start])
    pos = []
    heapq.heappush(pos,(0,start,[]))
    while pos:
        dist1,current,path = heapq.heappop(pos)
        for (next,dist2) in graph[current].items():
            if dist2+dist1 < dist[next][0]:
                dist[next] = (dist2+dist1,path+[next])
                heapq.heappush(pos,(dist1+dist2,next,path+[next]))
    return dist[end][1]

P = int(input())
graph = {input():{} for _ in range(P)}
for _ in range(int(input())):
    place1,place2,dist = input().split()
    graph[place1][place2] = graph[place2][place1] = int(dist)

for _ in range(int(input())):
    start,end = input().split()
    path = dijkstra(graph,start,end,P)
    s = start
    current = start
    for i in path:
        s += f'->({graph[current][i]})->{i}'
        current = i
    print(s)
```

Prim

Prim算法也是一种贪心算法。它从一个初始顶点开始，每次选择与当前生成树相连的权重最小的边所连接的顶点加入生成树中，直到所有顶点都被加入。

优点：

对于稠密图来说，Prim算法的效率更高，因为它按照顶点来考虑，而不是边。
实现时可以使用优先队列（最小堆）来维护候选边，使得算法的时间复杂度为O(ElogV)，其中V是顶点的数量。

缺点：

实现稍微复杂一些，需要维护候选边的数据结构。

1.将节点设置为有三个变量的节点，一个num储存节点标号，一个distance储存节点到下一个节点的权值，一个储存有到下一个节点的指针。同时建立关联表。
2.会将已经遍历的节点装入一个集合S中，同时建立一个bool数组visited来判断节点是否在集合内。先将初始节点加入S中。
3.然后探索集合外代价最小的边，来探索未纳入的节点，所以还要有一个集合外的边集合来比较最小边是哪条，将每次最小边所连接的点纳入这个集合当中，distoblock存放着它到集合最短的距离。同时每个节点初始化distoblock每个节点为99999，视为无穷。
————————————————————

兔子与星空

```
def prim(graph):
    selected = set()
    selected.add(list(graph.keys())[0])
    unselected = set(graph.keys()) - selected
    weight_sum = 0

    while unselected:
        min_weight = float('inf')
        start_vertex = None
        end_vertex = None
        for vertex in selected:
            for neighbor, weight in graph[vertex].items():
                if neighbor in unselected:
                    if min_weight > weight:
                        min_weight = weight
                        start_vertex = vertex
                        end_vertex = neighbor
        weight_sum += min_weight
        selected.add(end_vertex)
        unselected.remove(end_vertex)
    return weight_sum

n = int(input())
graph = {chr(i):{} for i in range(ord('A'),ord('A')+n)}
for _ in range(n-1):
    vertex,k,*lst = input().split()
    for i in range(int(k)):
        graph[vertex][lst[2*i]] = int(lst[2*i+1])
        graph[lst[2*i]][vertex] = int(lst[2*i+1])

print(prim(graph))
```

Krusal

每次都寻找两个节点之间的最小边，需要考虑是否会成环。

Kruskal算法是一种贪心算法。它首先**将所有的边按照权重从小到大进行排序**，然后依次考虑每条边，如果该边连接的两个顶点不在同一个连通分量中（即加入该边不会形成环），则将该边加入最小生成树中。

优点：

简单易懂，容易实现。
适用于稀疏图，因为它按照边来考虑，而不是顶点。

缺点：

实现时需要对边进行排序，时间复杂度为O(ElogE)，其中E是边的数量。
对于稠密图，算法的效率相对较低。

```
class Kruskal:
    def __init__(self, vertices):
        self.vertices = vertices
        self.parent = {vertex: vertex for vertex in vertices}
        self.rank = {vertex: 0 for vertex in vertices}
        self.minimum_spanning_tree = []

    def find(self, vertex):
        if self.parent[vertex] != vertex:
            self.parent[vertex] = self.find(self.parent[vertex])
        return self.parent[vertex]

    def union(self, u, v):
        root_u = self.find(u)
        root_v = self.find(v)

        if root_u != root_v:
            if self.rank[root_u] > self.rank[root_v]:
                self.parent[root_v] = root_u
            elif self.rank[root_u] < self.rank[root_v]:
                self.parent[root_u] = root_v
            else:
                self.parent[root_v] = root_u
                self.rank[root_u] += 1

    def kruskal(self, edges):
        sorted_edges = sorted(edges, key=lambda edge: edge[2])

        for edge in sorted_edges:
            u, v, weight = edge
            if self.find(u) != self.find(v):
                self.minimum_spanning_tree.append(edge)
                self.union(u, v)

        return self.minimum_spanning_tree

# Example Usage:
vertices = ['A', 'B', 'C', 'D', 'E']
edges = [
```

```
    ('A', 'B', 4),
    ('A', 'C', 6),
    ('B', 'C', 2),
    ('B', 'D', 9),
    ('C', 'D', 7),
    ('C', 'E', 8),
    ('D', 'E', 3)
]

kruskal = Kruskal(vertices)
minimum_spanning_tree = kruskal.kruskal(edges)
print(minimum_spanning_tree)
```

拓扑排序

可以用来判断有向图是否有环。

拓扑排序结果不唯一!

统计入度:
对图中的每个顶点，统计其入度，即指向它的边的数量。

初始化:
将入度为0的顶点加入一个队列，作为初始节点。

拓扑排序:
1.从队列中取出一个顶点，并输出。
2.将该顶点的所有邻接顶点的入度减1。
3.如果某个邻接顶点的入度减为0，则将其加入队列。
4.重复步骤3，直到队列为空。

检查:
如果输出的顶点数等于图中的顶点数，则拓扑排序成功，否则图中存在环。

```
def tuopo(graph,N):
    indegree = {i:0 for i in range(1,N+1)}
    for node in graph:
        for neighbor in graph[node]:
            indegree[neighbor] += 1

    queue = [node for node in range(1,N+1) if indegree[node] == 0]
    result = []
    while queue:
        node = queue.pop(0)
        result.append(node)
        for neighbor in graph[node]:
            indegree[neighbor] -= 1
            if indegree[neighbor] == 0:
                queue.append(neighbor)

    if len(result) == N:
        return False
    else:
        return True
```

```
for _ in range(int(input())):
    N, M = map(int, input().split())
    graph = {i: [] for i in range(1, N + 1)}
    for i in range(M):
        x, y = map(int, input().split())
        graph[x].append(y)

    print('Yes' if tuopo(graph,N) else 'No')
```

并查集

并查集是一种用于处理集合的数据结构，它主要支持两种操作：合并两个集合和查找一个元素所属的集合。包括并查集的基本概念、实现方式、路径压缩和应用场景。

```
class DisjointSet:
    def __init__(self, size):
        self.parent = [i for i in range(size)]
        self.rank = [0] * size

    def find(self, x):
        if self.parent[x] != x:
            self.parent[x] = self.find(self.parent[x])  # 路径压缩
        return self.parent[x]

    def union(self, x, y):
        root_x = self.find(x)
        root_y = self.find(y)
        if root_x != root_y:
            if self.rank[root_x] < self.rank[root_y]:
                self.parent[root_x] = root_y
            elif self.rank[root_x] > self.rank[root_y]:
                self.parent[root_y] = root_x
            else:
                self.parent[root_x] = root_y
                self.rank[root_y] += 1

# 示例
disjoint_set = DisjointSet(5)
disjoint_set.union(0, 1)
disjoint_set.union(1, 2)
disjoint_set.union(3, 4)
```

其他

懒删除

```
import heapq
stack = []
heap = []
set1 = set() # 标记已经删除的数据
while True:
    try:
        s = input()
    except EOFError:
        break

    if s == 'pop':
        if stack:
            set1.add(stack.pop()) # 标记

    elif s == 'min':
        while heap and heap[0] in set1:
            set1.remove(heapq.heappop(heap))
        if heap:
            print(heap[0])

    else:
        s1,s2 = s.split()
        s2 = int(s2)
        stack.append(s2)
        heapq.heappush(heap,s2)
```

运算符的实现

```
class A:
    def __init__(self, x):
        self.x = x

    def __lt__(self, other):
        if isinstance(other, A):
            return self.x < other.x
        elif isinstance(other, int):
            return self.x < other
        else:
            return NotImplemented

    def __ge__(self, other):
        if isinstance(other, A):
            return self.x >= other.x
        elif isinstance(other, int):
            return self.x >= other
        else:
            return NotImplemented
```

```
a, b, c = map(int, input().split())
print(isinstance(A(2), A))
print(A(a) < A(b))
print(A(a) >= A(c))
print(A(a) < c)
"""
in python:
def __lt__(self, other): <
def __ge__(self, other): >=
def __le__(self, other): <=
def __gt__(self, other): >
def __eq__(self, other): =
def __ne__(self, other): !=

def __add__(self, other): +
def __sub__(self, other): -
def __mul__(self, other): *
def __truediv__(self, other): /
def __floordiv__(self, other): //
def __mod__(self, other): %
def __pow__(self, other): **

def __iadd__(self, other): +=
def __isub__(self, other): -=
def __imul__(self, other): *=
def __itruediv__(self, other): /=
def __ifloordiv__(self, other): //=
def __imod__(self, other): %=
def __ipow__(self, other): **=

def __neg__(self): 一元负号(-)
def __pos__(self): 一元正号(+)
def __abs__(self): abs()
def __invert__(self, other): 按位取反~
def __round__(self, ndigits=None): round()
def __index__(self, other): 索引

def __getitem__(self, key): obj[key]
def __setitem__(self, key, value): obj[key] = value
def __delitem__(self, key): del obj[key]

def __len__(self): len()
def __iter__(self): 定义迭代器行为
def __getattr__(self, name): 定义获取不存在属性时的行为
def __setattr__(self, name, value): 定义设置属性的行为
def __delattr__(self, name): 定义删除属性的行为

def __repr__(self): repr()
def __str__(self): str() and print()
"""
```

复合函数

```
def combine(f,g):
    return lambda x: g(f(x))     # 补充这句代码
def inc(x):
    return x + 1
def square(x):
    return x * x
c  = int(input())
fx = combine(inc,square)
print(fx(c))
```