

# Chapter 1 算法分析 | ALGORITHM ANALYSIS

## 一、时间复杂度和空间复杂度 | Time complexity & Space complexity

### 1. $O(f(N))$ 、 $\Omega(f(N))$ 、 $\Theta(f(N))$ 和 $o(f(N))$

- (I)  $O(f(N))$  是“上界”，即不会超过某个函数  $f(N)$  的常数倍
- (II)  $\Omega(f(N))$  是“下界”，即不会小于某个函数  $f(N)$  的常数倍
- (III)  $\Theta(f(N))$  是“紧界”或“精确界”，即表示算法的时间复杂度的增长率被  $f(N)$  夹住，既不超过  $f(N)$  的常数倍，也不会低于  $f(N)$  的常数倍
- (IV)  $o(f(N))$  是“严格小于”上界，即  $T(N) \neq \Theta(f(N))$  但  $T(N) = O(f(N))$

#### Note

When analyzing the time complexity, generally  $T_{avg}(N)$  and  $T_{worst}(N)$  are analyzed.

1.  $T(N) = O(f(N))$  if there are positive constants  $c$  and  $n_0$  such that  $T(N) \leq c \times f(N)$  for all  $N \geq n_0$ .
2.  $T(N) = \Omega(g(N))$  if there are positive constants  $c$  and  $n_0$  such that  $T(N) \geq c \times g(N)$  for all  $N \geq n_0$ .
3.  $T(N) = \Theta(h(N))$  if and only if  $T(N) = O(h(N))$  and  $T(N) = \Omega(h(N))$ .
4.  $T(N) = o(h(N))$  if  $T(N) = O(p(N))$  and  $T(N) \neq \Theta(p(N))$ .

Always pick the smallest upper bound and the largest lower bound.

- ☞ If  $T_1(N) = O(f(N))$  and  $T_2(N) = O(g(N))$ , then
  - (a)  $T_1(N) + T_2(N) = \max(O(f(N)), O(g(N)))$ , 顺序
  - (b)  $T_1(N) * T_2(N) = O(f(N) * g(N))$ . 嵌套
- ☞ If  $T(N)$  is a (polynomial of degree  $k$ )  $k$ 次多项式, then  
 $T(N) = \Theta(N^k)$ .
- ☞  $\log^k N = O(N)$  for any constant  $k$ . This tells us that  
**logarithms grow very slowly.**

### 2. 递推式确定

主要的递推式：

$$T(N) = aT\left(\frac{N}{b}\right) + f(N)$$

简化一点，如果  $f(N) = N$ , 则递推式可转化为：

$$\begin{aligned} T(N) &= aT\left(\frac{N}{b}\right) + N \\ &= a\left(aT\left(\frac{N}{b^2}\right) + \frac{N}{b}\right) + N \\ &= a^k T\left(\frac{N}{b^k}\right) + N\left(\frac{a}{b}\right)^{k-1} + \dots + a\frac{N}{b} + N \end{aligned}$$

最终推出

(I) 当  $a = b$  时，

$$\begin{aligned} T(N) &= kN + NT(1) \\ &= O(N \log N) \end{aligned}$$

(II) 当  $a \neq b$  时，

$$T(N) = T(1)N^{\log_b^a} + \frac{b}{b-a}(N - N^{\log_b^a})$$

(也许没推错,也可能不太对 x anyway 掌握方法即可)

## 二、排序 | Sorting

### 1. 插入排序 | Insertion Sort

**Note**

简单理解为摸牌算法

#### 1.1) 算法流程 | Algorithm Flow



Figure 1: Insertion Sort

#### 1.2) 代码实现

```

Insertion Sort
1 void insertionSort(int nums[], int size){
2     for(int i = 0; i < size-1; i++){
3         int base = nums[i+1];
4         int j = i;
5         for(; j >= 0; j--){
6             if(base > nums[j]){
7                 break;
8             }
9             nums[j + 1] = nums[j];
10        }
11        nums[j + 1] = base;
12    }
13 }
14
15 // 折半插入排序
16 void insertBinarySort(int nums[], int size){
17     for(int i = 0; i < size-1; i++){
18         int left = 0, right = i, base = nums[i+1];
19         // int mid = (left + right)/2;
20         > 注意这里不能在循环外只是做了一次计算 傻了傻了
21         while(left <= right){
22             int mid = (left + right)/2;
23             if(nums[mid] > base){

```

```

24     }
25     else{
26         left = mid + 1;
27     }
28 }
29 // for(int j = i; j >= right; j--){
30     > 上述二分查找最后应该插入的位置是right右侧或者left左侧的位置
31     int j;
32     for(j = i; j >= left; j--){
33         nums[j + 1] = nums[j];
34     }
35     nums[j + 1] = base;
36 }

```

### 1.3) 复杂度分析 | Time Complexity

#### 1.3.1) 直接插入排序

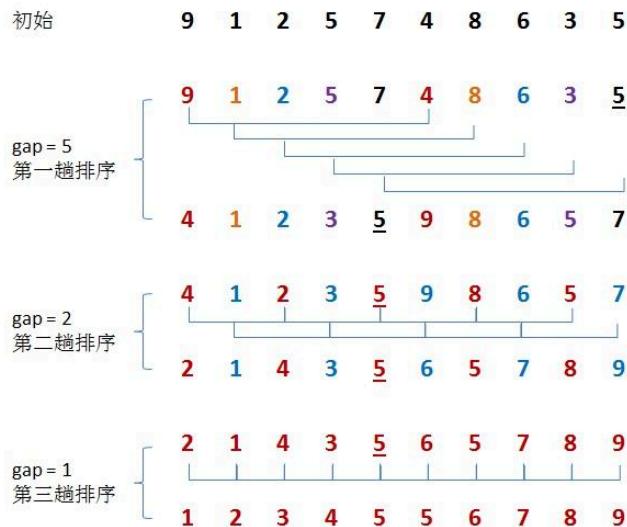
- (I) 最好时间复杂度:  $O(N)$
- (II) 最坏时间复杂度:  $O(N^2)$
- (III) 平均时间复杂度:  $O(N^2)$
- (IV) 空间复杂度:  $O(1)$
- (V) 稳定性: 稳定排序

#### 1.3.2) 折半插入排序 (二分插入排序)

- (I) 最好时间复杂度:  $O(N \log N)$
- (II) 最坏时间复杂度:  $O(N^2)$
- (III) 平均时间复杂度:  $O(N^2)$
- (IV) 空间复杂度:  $O(1)$
- (V) 稳定性: 稳定排序

## 2. 希尔排序 | Shell Sort

### 2.1) 算法流程 | Algorithm Flow



图·希尔排序示例图 知乎 @developer1024

Figure 2: Shell Sort

### Note

- (I) 希尔排序是分成  $n$  个子序列进行插入排序的，子序列的间隔是一个增量，随着排序的进行，增量逐渐减小，直到增量为 1 时，整个序列就变成了有序的。
- (II)  $n$  个子序列，Shell Sort 一定会分成  $n$  个组

## 2.2) 代码实现

### Shell Sort

```
1 void shellSortOriginal(int arr[], int n) {  
2     // 从n/2开始，每次间隔减半  
3     for (int gap = n / 2; gap > 0; gap /= 2) {  
4         // 对每个间隔进行插入排序  
5         for (int i = gap; i < n; i++) {  
6             int temp = arr[i];  
7             int j = i;  
8  
9             // 在间隔为gap的子序列中进行插入排序  
10            while (j >= gap && arr[j - gap] > temp) {  
11                arr[j] = arr[j - gap];  
12                j -= gap;  
13            }  
14            arr[j] = temp;  
15        }  
16    }  
17}  
18}
```

## 2.3) 复杂度分析 | Time Complexity

- (I) 平均时间复杂度:  $O(N^{1.3})$
- (II) 最坏时间复杂度:  $O(N^2)$
- (III) 最好时间复杂度:  $O(N)$
- (IV) 空间复杂度:  $O(1)$
- (V) 稳定性: 不稳定排序

## 3. 冒泡排序 | Bubble Sort

### 3.1) 算法流程 | Algorithm Flow



Figure 3: Bubble Sort

## 3.2) 代码实现

### Bubble Sort

```
1 /* 冒泡排序 */
2 void bubbleSort(int nums[], int size) {
3     // 外循环: 未排序区间为 [0, i]
4     for (int i = size - 1; i > 0; i--) {
5         // 内循环: 将未排序区间 [0, i] 中的最大元素交换至该区间的最右端
6         for (int j = 0; j < i; j++) {
7             if (nums[j] > nums[j + 1]) {
8                 int temp = nums[j];
9                 nums[j] = nums[j + 1];
10                nums[j + 1] = temp;
11            }
12        }
13    }
14 }
```

### 3.3 复杂度分析 | Time Complexity

- (I) 最好时间复杂度(经过优化的):  $O(N)$
- (II) 最坏时间复杂度:  $O(N^2)$
- (III) 平均时间复杂度:  $O(N^2)$
- (IV) 空间复杂度:  $O(1)$
- (V) 稳定性: 稳定排序

## 4. 选择排序 | Selection Sort

### 4.1 算法流程 | Algorithm Flow

- **时间复杂度为  $O(n^2)$ 、非自适应排序:** 外循环共  $n - 1$  轮，第一轮的未排序区间长度为  $n$ ，最后一轮的未排序区间长度为 2，即各轮外循环分别包含  $n$ 、 $n - 1$ 、 $\dots$ 、3、2 轮内循环，求和为  $\frac{(n-1)(n+2)}{2}$ 。
- **空间复杂度为  $O(1)$ 、原地排序:** 指针  $i$  和  $j$  使用常数大小的额外空间。
- **非稳定排序:** 如图 11-3 所示，元素 `nums[i]` 有可能被交换至与其相等的元素的右边，导致两者的相对顺序发生改变。



Figure 4: Selection Sort

## 4.2) 代码实现

### Selection Sort

```

1 /* 选择排序 */
2 void selectionSort(int nums[], int n) {
3     // 外循环: 未排序区间为 [i, n-1]
4     for (int i = 0; i < n - 1; i++) {
5         // 内循环: 找到未排序区间内的最小元素
6         int k = i;
7         for (int j = i + 1; j < n; j++) {
8             if (nums[j] < nums[k])
9                 k = j; // 记录最小元素的索引
10    }
11    // 将该最小元素与未排序区间的首个元素交换
12    int temp = nums[i];
13    nums[i] = nums[k];
14    nums[k] = temp;
15 }
16 }
```

## 5. 堆排序 | Heap Sort

### 5.1) 算法流程 | Algorithm Flow



Figure 5: Heap Sort

## 5.2) 代码实现

### Heap Sort

```
1 /* 堆的长度为 n , 从节点 i 开始, 从顶至底堆化 */
```

```

2 void siftDown(int nums[], int n, int i) {
3     while (1) {
4         // 判断节点 i, l, r 中值最大的节点, 记为 ma
5         int l = 2 * i + 1;
6         int r = 2 * i + 2;
7         int ma = i;
8         if (l < n && nums[l] > nums[ma])
9             ma = l;
10        if (r < n && nums[r] > nums[ma])
11            ma = r;
12        // 若节点 i 最大或索引 l, r 越界, 则无须继续堆化, 跳出
13        if (ma == i) {
14            break;
15        }
16        // 交换两节点
17        int temp = nums[i];
18        nums[i] = nums[ma];
19        nums[ma] = temp;
20        // 循环向下堆化
21        i = ma;
22    }
23 }
24
25 /* 堆排序 */
26 void heapSort(int nums[], int n) {
27     // 建堆操作: 堆化除叶节点以外的其他所有节点
28     for (int i = n / 2 - 1; i >= 0; --i) {
29         siftDown(nums, n, i);
30     }
31     // 从堆中提取最大元素, 循环 n-1 轮
32     for (int i = n - 1; i > 0; --i) {
33         // 交换根节点与最右叶节点 (交换首元素与尾元素)
34         int tmp = nums[0];
35         nums[0] = nums[i];
36         nums[i] = tmp;
37         // 以根节点为起点, 从顶到底进行堆化
38         siftDown(nums, i, 0);
39     }
40 }

```

### 5.3) 复杂度分析 | Time Complexity

- **时间复杂度为  $O(n \log n)$ 、非自适应排序:** 建堆操作使用  $O(n)$  时间。从堆中提取最大元素的时间复杂度为  $O(\log n)$ ，共循环  $n - 1$  轮。
- **空间复杂度为  $O(1)$ 、原地排序:** 几个指针变量使用  $O(1)$  空间。元素交换和堆化操作都是在原数组上进行的。
- **非稳定排序:** 在交换堆顶元素和堆底元素时，相等元素的相对位置可能发生变化。

Figure 6: Heap Sort Complexity

## 6. 归并排序 | Merge Sort

### 6.1) 算法流程 | Algorithm Flow

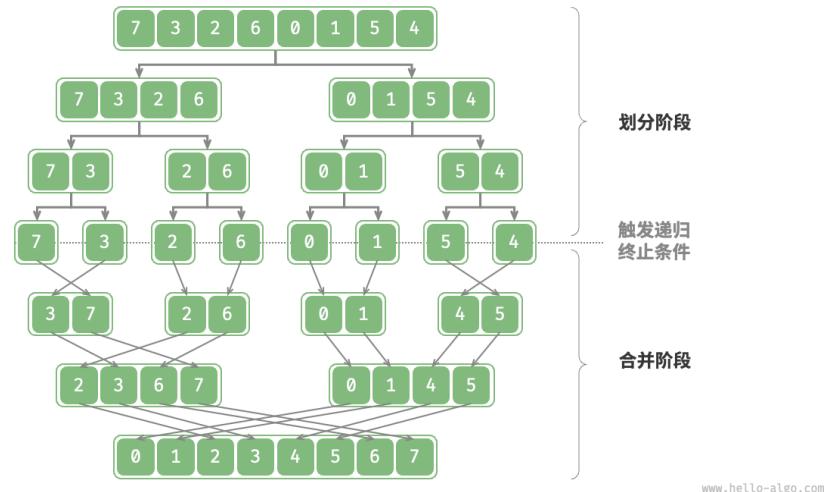


Figure 7: Merge Sort

### 6.2) 代码实现

```
Merge Sort
1 /* 合并左子数组和右子数组 */
2 void merge(int *nums, int left, int mid, int right) {
3     // 左子数组区间为 [left, mid], 右子数组区间为 [mid+1, right]
4     // 创建一个临时数组 tmp , 用于存放合并后的结果
5     int tmpSize = right - left + 1;
6     int *tmp = (int *)malloc(tmpSize * sizeof(int));
7     // 初始化左子数组和右子数组的起始索引
8     int i = left, j = mid + 1, k = 0;
9     // 当左右子数组都还有元素时, 进行比较并将较小的元素复制到临时数组中
10    while (i <= mid && j <= right) {
11        if (nums[i] <= nums[j]) {
12            tmp[k++] = nums[i++];
13        } else {
14            tmp[k++] = nums[j++];
15        }
16    }
17    // 将左子数组和右子数组的剩余元素复制到临时数组中
18    while (i <= mid) {
19        tmp[k++] = nums[i++];
20    }
21    while (j <= right) {
22        tmp[k++] = nums[j++];
23    }
24    // 将临时数组 tmp 中的元素复制回原数组 nums 的对应区间
25    for (k = 0; k < tmpSize; ++k) {
26        nums[left + k] = tmp[k];
27    }
28    // 释放内存
29    free(tmp);
30 }
31
32 /* 归并排序 */
33 void mergeSort(int *nums, int left, int right) {
34     // 终止条件
35     if (left >= right)
```

```

36     |     return; // 当子数组长度为 1 时终止递归
37     // 划分阶段
38     int mid = left + (right - left) / 2;      // 计算中点
39     mergeSort(nums, left, mid);        // 递归左子数组
40     mergeSort(nums, mid + 1, right); // 递归右子数组
41     // 合并阶段
42     merge(nums, left, mid, right);
43 }

```

### 6.3) 复杂度分析 | Time Complexity

- 时间复杂度为  $O(n \log n)$ 、非自适应排序：划分产生高度为  $\log n$  的递归树，每层合并的总操作数量为  $n$ ，因此总体时间复杂度为  $O(n \log n)$ 。
- 空间复杂度为  $O(n)$ 、非原地排序：递归深度为  $\log n$ ，使用  $O(\log n)$  大小的栈帧空间。合并操作需要借助辅助数组实现，使用  $O(n)$  大小的额外空间。
- 稳定排序：在合并过程中，相等元素的次序保持不变。

## 7. 快速排序 | Quick Sort

### 7.1) 算法流程 | Algorithm Flow

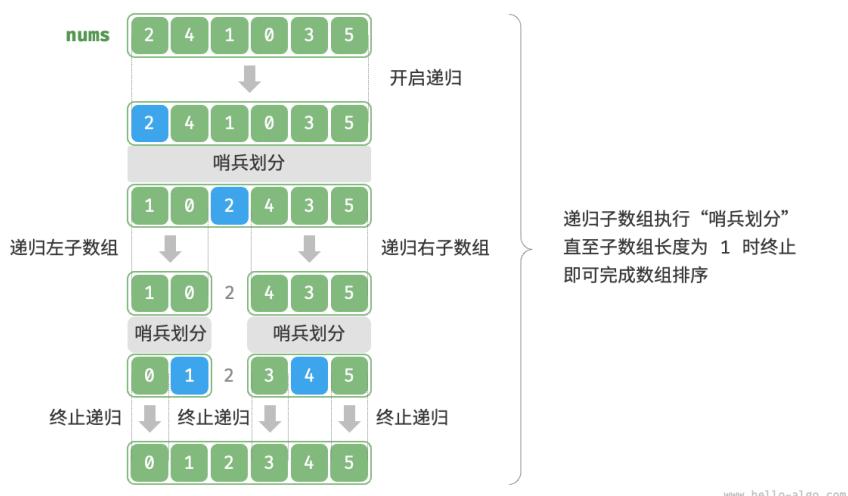


Figure 8: Quick Sort

### 7.2) 代码实现

```

Quick Sort
1 /* 快速排序 */
2 void quickSort(int nums[], int left, int right) {
3     // 子数组长度为 1 时终止递归
4     if (left >= right) {
5         return;
6     }
7     // 哨兵划分
8     int pivot = partition(nums, left, right);
9     // 递归左子数组、右子数组
10    quickSort(nums, left, pivot - 1);
11    quickSort(nums, pivot + 1, right);
12 }

```

### 7.3) 复杂度分析 | Time Complexity

- 时间复杂度为  $O(n \log n)$ 、非自适应排序：在平均情况下，哨兵划分的递归层数为  $\log n$ ，每层中的总循环数为  $n$ ，总体使用  $O(n \log n)$  时间。在最差情况下，每轮哨兵划分操作都将长度为  $n$  的数组划分为长度为 0 和  $n - 1$  的两个子数组，此时递归层数达到  $n$ ，每层中的循环数为  $n$ ，总体使用  $O(n^2)$  时间。
- 空间复杂度为  $O(n)$ 、原地排序：在输入数组完全倒序的情况下，达到最差递归深度  $n$ ，使用  $O(n)$  栈帧空间。排序操作是在原数组上进行的，未借助额外数组。
- 非稳定排序：在哨兵划分的最后一步，基准数可能会被交换至相等元素的右侧。

### 8. 排序的一般下界 | A General Lower Bound for Sorting

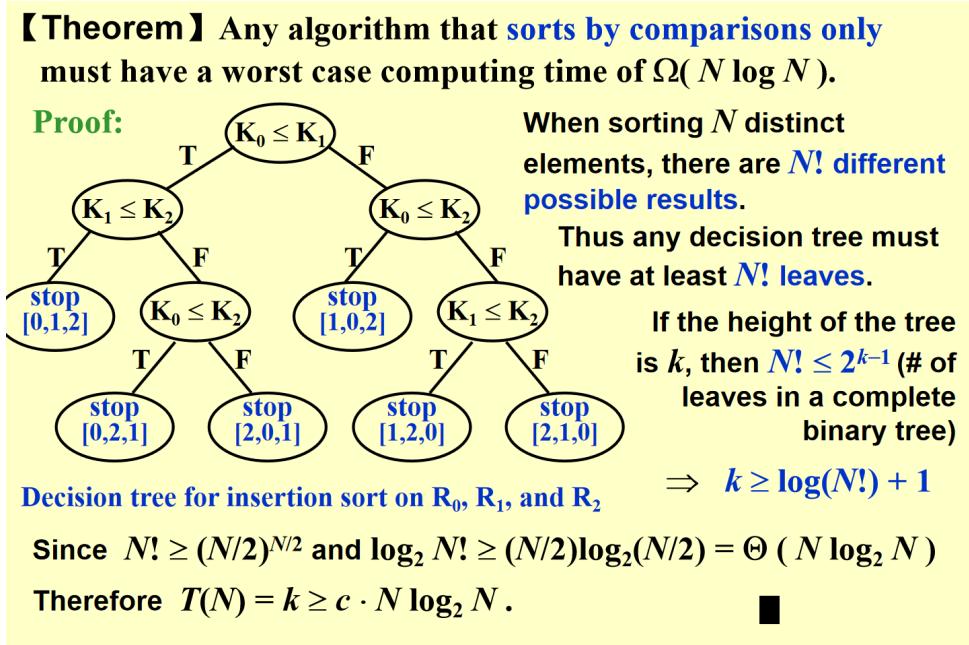


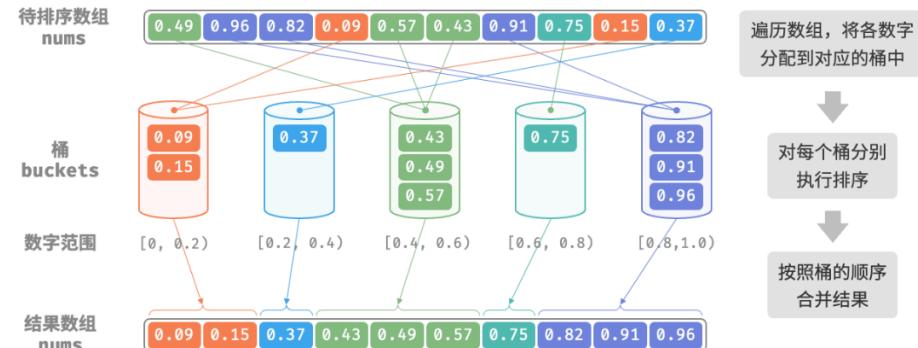
Figure 9: A General Lower Bound for Sorting

## 9. 桶排序 | Bucket Sort

### 9.1) 算法流程 | Algorithm Flow

考虑一个长度为  $n$  的数组，其元素是范围  $[0, 1)$  内的浮点数。桶排序的流程如图 11-13 所示。

1. 初始化  $k$  个桶，将  $n$  个元素分配到  $k$  个桶中。
2. 对每个桶分别执行排序（这里采用编程语言的内置排序函数）。
3. 按照桶从小到大的顺序合并结果。



www.hello-algo.com

Figure 10: Bucket Sort

## 10. 基数排序 | Radix Sort

### 10.1) 算法流程 | Algorithm Flow

以学号数据为例，假设数字的最低位是第 1 位，最高位是第 8 位，基数排序的流程如图 11-18 所示。

1. 初始化位数  $k = 1$ 。
2. 对学号的第  $k$  位执行“计数排序”。完成后，数据会根据第  $k$  位从小到大排序。
3. 将  $k$  增加 1，然后返回步骤 2. 继续迭代，直到所有位都排序完成后结束。



Figure 11: Radix Sort

# Chapter 2 数据结构 | Data Structure

## Attention

- (I) constraint 限制
- (II) Consecutive 连续的
- (III) sequential 顺序的
- (IV) query 查询
- (V) siblings 兄弟姐妹

## 一、栈 | Stack

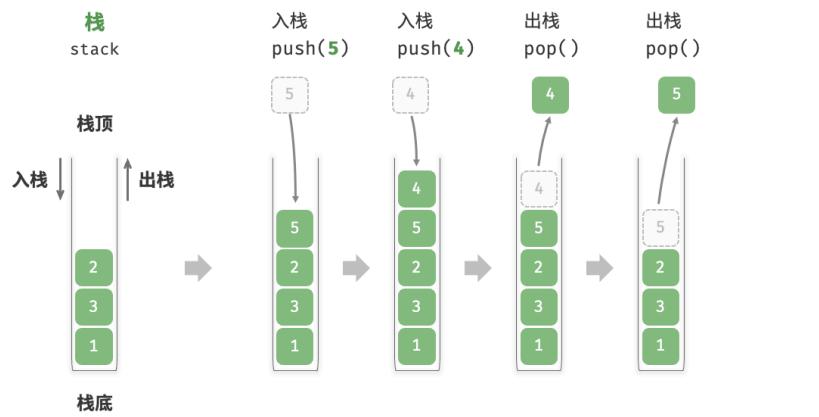


Figure 12: Stack

## 1. 基本功能 | Basic Functions

方法	描述	时间复杂度
push()	元素入栈 (添加至栈顶)	$O(1)$
pop()	栈顶元素出栈	$O(1)$
peek()	访问栈顶元素	$O(1)$

Figure 13: Basic Functions

## 2. 基本实现 | Basic Implementation

### 2.1) 基于链表实现 | Linked List Implementation

### 2.2) 基于数组实现 | Array Implementation

## 3. 栈的应用 | Applications of Stack

### 3.1) 表达式求值 | Expression Evaluation

#### Cite

- (1) 中缀表达式 | Infix: 运算符在操作数之间的表达式, 称为中缀表达式。

a + b \* c - d/e

(2) 前缀表达式 | Prefix: 运算符在操作数之前的表达式, 称为前缀表达式

$- + a * b c / d e$

(3) 后缀表达式 | Postfix: 运算符在操作数之后的表达式, 称为后缀表达式。

$a b c * + d e / -$

### 中缀表达式与后缀表达式的相互转化:

#### (1) 中缀→后缀

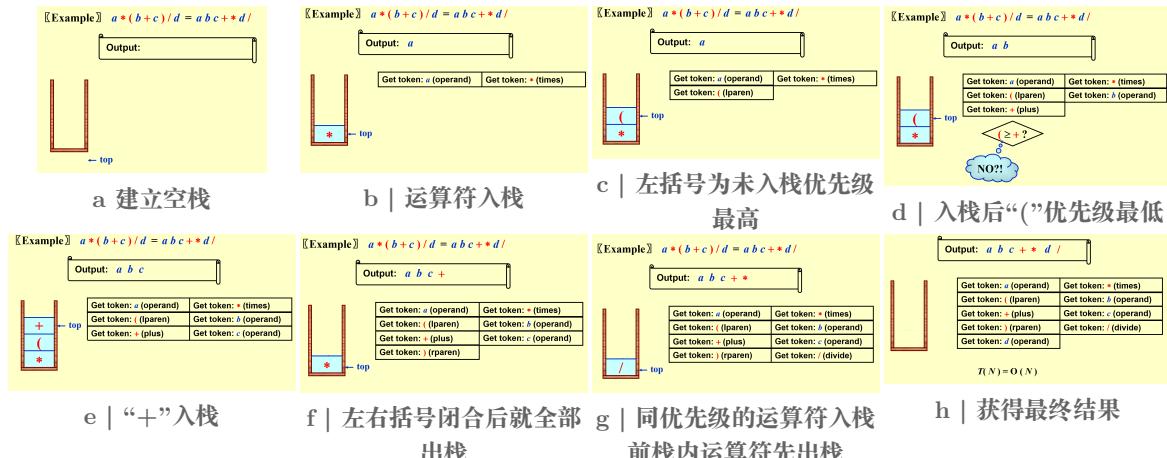


Figure 14: 转化流程

#### (2) 后缀→中缀

此过程比较简单, 只需要将后缀表达式中的操作数和运算符依次入栈, 遇到运算符时就出栈两个操作数进行计算, 然后将结果入栈, 最后栈中剩下的就是最终结果。

#### Note

#### Solutions:

① Never pop a ( from the stack except when processing a ).

② Observe that when ( is not in the stack, its precedence is the highest; but when it is in the stack, its precedence is the lowest. Define in-stack precedence and incoming precedence for symbols, and each time use the corresponding precedence for comparison.

Note:  $a - b - c$  will be converted to  $a b - c -$ . However,  $2^2 2^3 (2^{2^3})$  must be converted to  $2 2 3 ^ ^$ , not  $2 2 ^ 3 ^$  since exponentiation associates right to left. (理解为后一个“^”优先级更高)

Figure 15: 注意事项

### 3.2) 系统栈 | System Stack

While running recursive programs, Return Address, Stack Frame, Local Variables are all pushed into the stack. If too many recursions are called, the system stack will be overflowed and the system will crash.

当递归程序为尾递归程序时, 编译器会将递归调用转换为循环调用, 避免了系统栈可能出现的溢出

## 二、队列 | Queue

A queue is a First-In-First-Out (FIFO) list, that is, an ordered list in which insertions take place at one end and deletions take place at the opposite end.

## 1. 数组实现

### 1.1) 实现

使用一个变量 `front` 指向队首元素的索引，并维护一个变量 `size` 用于记录队列长度。定义 `rear = front + size - 1`，这个公式计算出的 `rear` 指向队尾元素，数组中包含元素的有效区间为 `[front, rear]`

### 1.2) 代码实现:

```
Quene
1 struct QueueRecord {
2     int Capacity; /* max size of queue */
3     int Front;      /* the front pointer */
4     int Rear;       /* the rear pointer */
5     int size; /* Optional - the current size of queue */
6     ElementType *Array; /* array for queue elements */
7 } ;
8
9 /*常见函数*/
10 void Enqueue(ElementType X,Queue Q);
11 ElementType Front(Queue Q);
12 void Dequeue(Queue Q);
```

## 2. 基本操作

- (1) Initialize | 初始化: `rear = -1, front = 0`.
  - (2) Enqueue | 入队: 将输入元素赋值给 `rear++` 索引处, 并将 `size` 增加 1
  - (3) Dequeue | 出队: 先 `free(Arr[front])`, 再将 `front` 增加 1, 并将 `size` 减少 1

### - Note

其实这里 size 是多余的

### 3. 循环队列

- (1) **Initialize | 初始化:** `rear = -1 % MAX_SIZE, front = 0.`
  - (2) **Enqueue | 入队:** 将输入元素赋值给 `(rear++) % MAX_SIZE` 索引处
  - (3) **Dequeue | 出队:** 先 `free(Arr[front])`, 再 `(front++) % MAX_SIZE`
  - (4) 当 `front - rear == 2` 的时候, 循环队列填满; 如果引入 size, 则可以多出一个队列空间

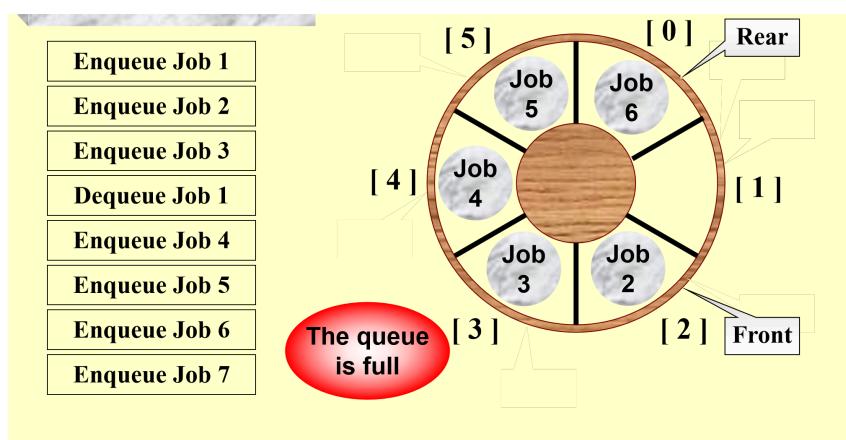


Figure 16: 循环队列

### 三、 树 | Tree

### - Attention

- (1) 对于二叉树而言给定 preorder 和 postorder，无法确定这棵树

- (2) 任何二叉树都可以用数组表示
- (3) If a general tree T is converted into a binary tree BT, then the **BT inorder traversals** gives the same sequence as that of the **post-order traversal of T**?
- (4) 二叉决策树-折半查找判定树

我们要写折半查找的判定树，首先就是要了解折半查找的步骤，然后依次将mid指向的值作为树的value。如下展示：

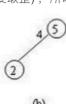
例如：长度为10的 折半查找 判定树的具体生成过程，都遵循左孩子结点<根结点<右孩子结点

**{1,2,3,4,5,6,7,8,9,10}**

在长度为10的有序表中进行折半查找，不论查找哪个记录，都必须和中间记录进行比较，而中间记录为  $(1+10)/2=5$  (注意要取整，即向下取整) 即判定数的根结点为5。

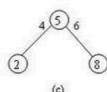
(a)

考虑判定树的左子树，即将查找区域调整到左半区，此时的查找区间为[1,4]，那么中间值为  $(1+4)/2=2$  (注意要取整)，所以做孩子根结点为2



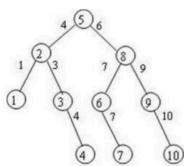
(b)

考虑判定树的右子树，即将查找区域调整到右半区，此时的查找区间为[6,10]，那么中间值为  $(6+10)/2=8$  (注意要取整)，所以做孩子根结点为8



(c)

重复以上步骤，依次去确定左右孩子



<https://blog.csdn.net/m4900515>

## 1. 二叉树 | Binary Tree

### 1.1) 基本概念

- (I) **二叉树**：是一种非线性数据结构，二叉树与链表有点类似，基本单元是**节点**，每个节点包含**值data**，左子树**left**，右子树**right**。

**Tip**

以下二叉树的概念，对于普通树也是适用的，而且普通树可以向二叉树进行转化：

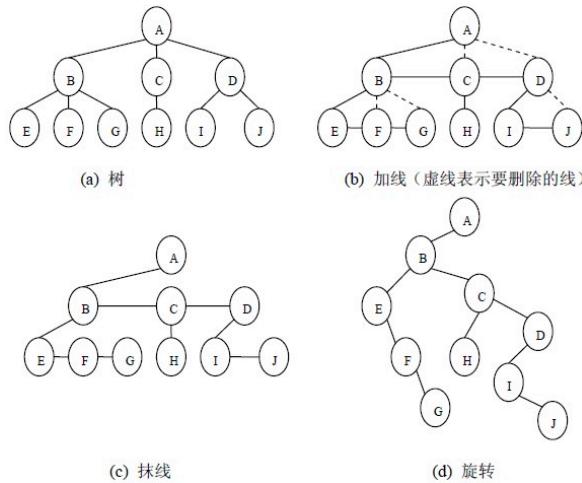


Figure 17: Transition

**转化方法：**将每个节点的第一个子节点作为其左子节点,其余子节点作为其左子节点的右子节点。这样,普通树就转换为了二叉树

代码实现方式:

```

1 #include <stdio.h>
2 //c语言实现
3 typedef struct TreeNode{
4     int data;
5     TreeNode *left;
6     TreeNode *right;
7 }TreeNode;
8
9 typedef struct TreeNode *Node;
10 //创建一个新节点
11 Node createNode(int data){
12     Node newNode = (Node)malloc(sizeof(TreeNode));
13     newNode->data = data;
14     newNode->left = NULL;
15     newNode->right = NULL;
16     return newNode;
17 }
```

(II) 树的常见术语:

**Summary**

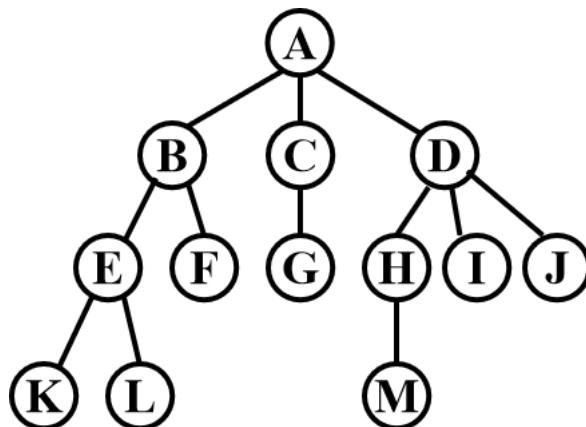


Figure 18: Example

- (1) **Degree of a node | 节点的度:** number of subtrees of the node. For example,  $\text{degree}(A) = 3$ ,  $\text{degree}(F) = 0$ .
- (2) **Degree of a tree | 树的度:** The maximum degree of any node across all the trees.

$$\text{Degree} = \max_{\text{node } \in \text{tree}} \{\text{Degree}(\text{node})\}$$

For example, degree of this tree = 3.

- (3) **Siblings | 兄弟姐妹:** Children of the same parent.
- (4) **Path from  $n_1$  to  $n_2$ :** a (unique) sequence of nodes  $n_1, n_2, \dots, n_k$  such that  $n_i$  is the parent of  $n_{i+1}$  for  $1 < i < k$ .
- (5) **Length of path:** number of edges on the path.
- (6) **Depth of  $n_i$  | 节点深度:** length of the unique path from the root to  $n_i$ ,  $\text{Depth}(\text{root}) = 0$ .
- (7) **Height of  $n_i$  | 节点高度:** length of the longest path from  $n_i$  to a leaf.  $\text{Height}(\text{leaf}) = 0$ , and  $\text{height}(D) = 2$ .
- (8) **Level of  $n_i$  | 节点层数:** 从顶至底递增, 根节点所在层为 1
- (9) **Height (Depth) of a tree | 树的高度/深度:**  $= \text{height}(\text{root}) = \text{depth}(\text{deepest leaf}) = 3$
- (10) **Ancestors of a node | 节点的祖先 (根节点):** all the nodes along the path from the node up to the root.
- (11) **Descendants of a node | 节点的后代 (子节点):** all the nodes in its subtrees.

### (III) 树的性质:

对于

## 1.2) 树的遍历 | Tree Traversals —— visit each node exactly once

### (I) Preorder Traversal | 先序遍历:

优先打印节点, 再打印节点的子节点:

```

1 void preorder ( tree_ptr tree ){
2     if(tree){
3         visit ( tree );
4         preorder ( tree->Left );
5         preorder ( tree->Right );
6     }
7 }
```

### (II) Postorder Traversal | 后序遍历:

优先打印子节点, 再打印节点的根节点:

```

1 void postorder(tree_ptr tree) {
2     if (tree) {
3         postorder(tree->Left);
4         postorder(tree->Right);
5         visit(tree);
6     }
7 }
```

### (III) Inorder Traversal | 中序遍历:

中序遍历()

```

1 void inorder ( tree_ptr tree ){
2     if(tree){
3         inorder(tree->Left);
```

```

4     visit(tree);
5     inorder(tree->Right);
6   }
7 }
```

#### (IV) Levelorder Traversal | 层序遍历:

层序遍历——顾名思义，按照层的顺序进行打印

```

1 void levelorder(tree_ptr tree) {
2   queue_ptr q = create_queue(); // create a queue
3   enqueue(q, tree); // enqueue the root node
4   while (!is_empty(q)) {
5     tree_ptr node = dequeue(q); // dequeue the front node
6     visit(node); // visit the node
7     if (node->Left) {
8       enqueue(q, node->Left); // enqueue the left child
9     }
10    if (node->Right) {
11      enqueue(q, node->Right); // enqueue the right child
12    }
13  }
14  destroy_queue(q); // destroy the queue
15 }
```

#### 前置函数

```

1 typedef struct QueueRecord *queue_ptr; // define queue pointer type
2 int is_empty(queue_ptr q) {
3   return q->front == q->rear; // check if the queue is empty
4 }
5
6
7 queue_ptr create_queue() {
8   queue_ptr q = malloc(sizeof(struct QueueRecord)); // allocate memory for the queue
9   q->capacity = MAX_SIZE; // set the capacity of the queue
10  q->front = 0; // initialize front pointer
11  q->rear = -1; // initialize rear pointer
12  q->array = malloc(q->capacity * sizeof(tree_ptr)); // allocate memory for the array
13  return q; // return the created queue
14 }
15
16
17 tree_ptr dequeue(queue_ptr q) {
18   if (is_empty(q)) {
19     printf("Queue is empty\n");
20     return NULL;
21   }
22   tree_ptr node = q->array[q->front]; // dequeue the front node
23   q->front = (q->front + 1) % q->capacity; // circular increment
24   return node; // return the dequeued node
25 }
26
27
28 void enqueue(queue_ptr q, tree_ptr node) {
29   if ((q->rear + 1) % q->capacity == q->front) {
30     printf("Queue is full\n");
31     return;
32   }
33   q->rear = (q->rear + 1) % q->capacity; // circular increment
34   q->array[q->rear] = node; // enqueue the node
35 }
36
37
38 void destroy_queue(queue_ptr q) {
```

```

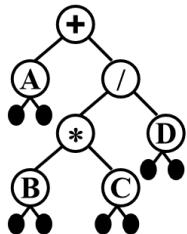
39     free(q->array); // free the array
40     free(q); // free the queue structure
41 }

```

### 1.3) 表达式树 | Expression Trees (Syntax trees)

【Example】 Given an infix expression:

$A + B * C / D$



Then **inorder traversal**  $\Rightarrow A + B * C / D$

**postorder traversal**  $\Rightarrow A B C * D / +$

**preorder traversal**  $\Rightarrow + A / * B C D$

- (I) **Infix expression | 中缀表达式:** 运算符写在两个操作数中间的表达式, 称作中缀表达式 (即数学中常用的正常的表达式)

eg.  $(A + B) * C - D$

- (II) **Prefix expression | 前缀表达式:** 可以是单个变量, 一个操作符, 后面跟两个操作数, 每个前缀表达式包括一个操作符和两个操作数。

eg.  $- * + A B C D$

- (III) **Postfix expression | 后缀表达式:** 可以是单个变量, 或者是两个操作数外跟一个操作符, 每个后缀表达式包括两个操作数后跟一个操作符

eg.  $A B + C * D -$

表达式树 —

中缀表达式->后缀表达式——转化

$(a + b)*(c*(d + e)) = a b + c d e + * *$

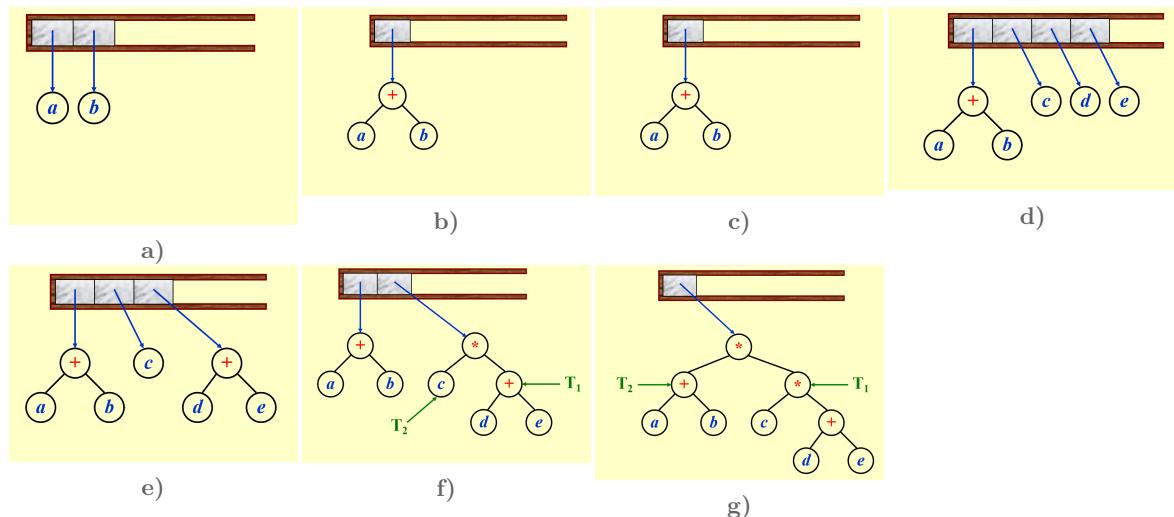


Figure 19: 转化流程

## 1.4) 线索二叉树 | Threaded Binary Trees

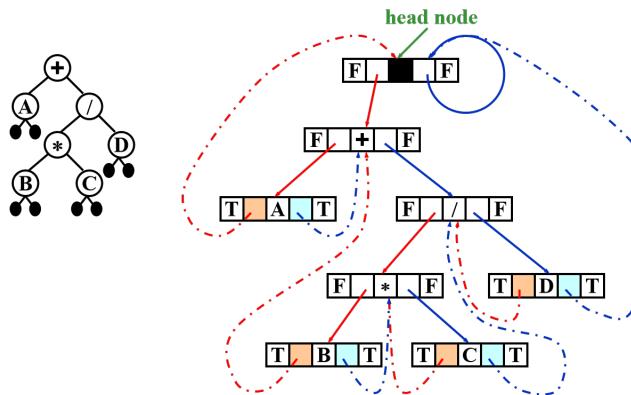
### Note

线索二叉树分为 in-order、pre-order、post-order，只需要根据 traversal 顺序来即可

- Rule 1: If Tree->Left is null, replace it with a pointer to the inorder predecessor of Tree.
- Rule 2: If Tree->Right is null, replace it with a pointer to the inorder successor of Tree.
- Rule 3: There must not be any loose threads. Therefore a threaded binary tree must have a head node of which the left child points to the first node.

举例：

【Example】 Given the syntax tree of an expression (infix)  
 $A + B * C / D$



16/16

## 2. 二叉树的分类和性质 | Classification and Properties of Binary Trees

### (I) 二叉树的分类：

- 满二叉树 | Full Binary Tree: A binary tree in which every node has either 0 or 2 children.
- 完美二叉树 | Perfect Binary Tree: A binary tree in which all the internal nodes have two children and all leaves are at the same level.
- 完全二叉树 | Complete Binary Tree:
- 斜二叉树 | Skewed Binary Tree:

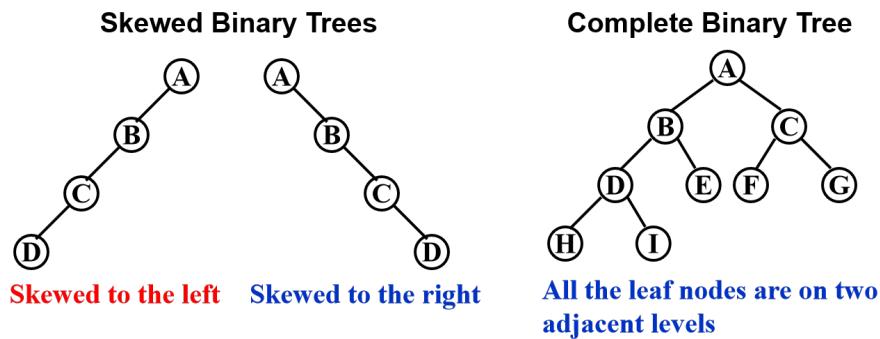
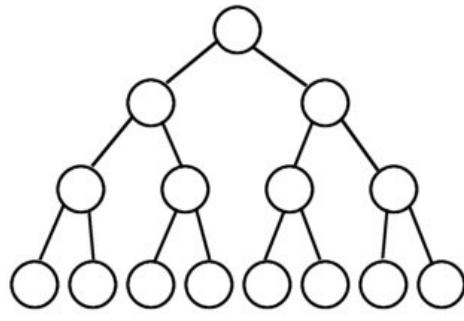
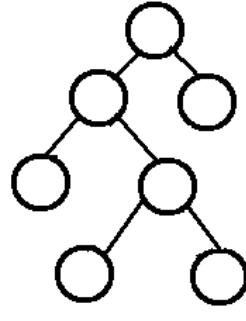


Figure 20: Skewed Binary Tree and Complete Binary Tree



a) 完美二叉树



b) 满二叉树

Figure 21: 满二叉树与完美二叉树

## (II) 二叉树的性质 | Properties of Binary Trees:

- a) 在第  $i$  层最多的节点数是  $2^{i-1}$  个
- b) 一个深度为  $k$  的二叉树中最多有  $2^k - 1$  个节点
- c) 对于任何非空二叉树都有  $n_0 = n_2 + 1$ , 其中  $n_0$  是叶节点的个数,  $n_2$  是节点度为 2 的节点数

Proof

For Question b): 在第  $j$  层,  $0 \leq j \leq k$ , 最多的节点数为  $2^{j-1}$  个, 则深度为  $k$  的二叉树最多含有的节点数  $N$  可以由下式计算:

$$N = \sum_{j=0}^k 2^{j-1} = 2^k - 1$$

命题得证.

For Question c):

1. 考虑整个二叉树的总节点数  $N = n_2 + n_1 + n_0$ , 其中  $n_1$  为节点度为 1 的节点的数目;
2. 再考虑节点总的边数  $B = 0 \times n_0 + 1 \times n_1 + 2 \times n_2 = n_1 + 2n_2$ . 而边数和节点数的关系为:  $B + 1 = N$  (设想把每条边都分配给它下面相连的节点, 这样只有根节点没有对应的边, 于是两者之间关系就出来了  $-> B + 1 = N$ )

由以上两点得出: 对于任何非空二叉树都有  $n_0 = n_2 + 1$

命题得证.

- d) 扩展:  $n_0 = n_2 + 2n_3 + 3n_4 + \dots + 1$  or  $n_0 = 1 + \sum_{k=2}^{\infty} (k-1)n_k$

## 3. 二叉查找树 | Binary Search Tree

### 3.1) 基本概念

A **binary search tree** is a binary tree. It may be empty. If it is not empty, it satisfies the following properties:

**Properties**

- (I) Every node has a key which is an integer(or any other element type that can be compared in size), and the keys are distinct.
- (II) The keys in a nonempty left subtree must be smaller than the key in the root of the subtree.
- (III) The keys in a nonempty right subtree must be larger than the key in the root of the subtree.
- (IV) The left and right subtrees are also binary search trees.

## 3.2) ADT 实现 | Implementation of ADT

### 3.2.1) ADTS

```
1 SearchTree MakeEmpty( SearchTree T );
2 Position Find( ElementType X, SearchTree T );
3 Position FindMin( SearchTree T );
4 Position FindMax( SearchTree T );
5 SearchTree Insert( ElementType X, SearchTree T );
6 SearchTree Delete( ElementType X, SearchTree T );
7 ElementType Retrieve( Position P );
```

### 3.2.2) Implementations

#### 1. 查找(伪递归和循环实现):

```
1 Position Find(ElementType X, SearchTree T)
2 {
3     if (T == NULL)                      /* not found in an empty tree */
4         return NULL;
5     if (X < T->Element)                /* if smaller than root */
6         return Find(X, T->Left);       /* search left subtree */
7     else if (X > T->Element)          /* if larger than root */
8         return Find(X, T->Right);      /* search right subtree */
9     else                                /* if X == root */
10        return T;                      /* found */
11 }
12
13 Position Iter_Find(ElementType X, SearchTree T)
14 {
15     /* iterative version of Find */
16     while (T)
17     {
18         if (X == T->Element)
19             return T; /* found */
20         if (X < T->Element)
21             T = T->Left; /* move down along left path */
22         else
23             T = T->Right; /* move down along right path */
24     } /* end while-loop */
25     return NULL; /* not found */
26 }
```

#### 2. 查找最小/最大值(伪递归和循环实现):

```
1 Position FindMin(SearchTree T)
2 {
3     if (T == NULL)                      /* not found in an empty tree */
4         return NULL;
5     else if (T->Left == NULL)
6         return T; /* found left most */
7     else
8         return FindMin(T->Left); /* keep moving to left */
9 }
10
11 Position FindMax(SearchTree T)
12 {
13     if (T != NULL)
14         while (T->Right != NULL)
15             T = T->Right; /* keep moving to find right most */
16     return T;              /* return NULL or the right most */
17 }
```

### 3. 插入 | Insert:

```

1 SearchTree Insert(ElementType X, SearchTree T)
2 {
3     if (T == NULL)
4     { /* Create and return a one-node tree */
5         T = malloc(sizeof(struct TreeNode));
6         if (T == NULL)
7             FatalError("Out of space!!!");
8         else
9         {
10             T->Element = X;
11             T->Left = T->Right = NULL;
12         }
13     } /* End creating a one-node tree */
14     else /* If there is a tree */
15     {
16         if (X < T->Element)
17             T->Left = Insert(X, T->Left);
18         else if (X > T->Element)
19             T->Right = Insert(X, T->Right);
20     /* Else X is in the tree already; we'll do nothing */
21     return T; /* Do not forget this line!! */
22 }
```

### 4. 删除 | Delete:

```

1 /**
2 * Definition for a binary tree node.
3 * struct TreeNode {
4 *     int val;
5 *     struct TreeNode *left;
6 *     struct TreeNode *right;
7 * };
8 */
9 struct TreeNode* FindMin(struct TreeNode* root){
10    while(root->left){
11        root = root->left;
12    }
13    return root;
14 }
15
16 struct TreeNode* deleteNode(struct TreeNode* root, int key) {
17    struct TreeNode* ptr = NULL;
18    if(!root){
19        printf("error\n");
20        return NULL;
21    }
22    if(key < root->val){
23        root->left = deleteNode(root->left, key);
24    }
25    else if(key > root->val){
26        root->right = deleteNode(root->right, key);
27    }
28    else{
29        if(root->left&&root->right){
30            ptr = FindMin(root->right);
31            root->val = ptr->val;
32            root->right = deleteNode(root->right, root->val);
33        }
34        else if(root->left){
35            ptr = root;
```

```

36     root = root->left;
37     free(ptr);
38 }
39 else if(root->right){
40     ptr = root;
41     root = root->right;
42     free(ptr);
43 }
44 else{
45     ptr = root;
46     root = NULL;
47     free(ptr);
48 }
49 }
50 return root;
51 }

```

## 四、堆 | Heap

### 1. 存储

堆通过完全二叉树存储在数组中

A complete binary tree of height  $h$  has between  $2^h$   
and  $2^{h+1}-1$  nodes.  $\rightarrow h = \lfloor \log N \rfloor$

❖ Array Representation : BT[n + 1] (BT[0] is not used)

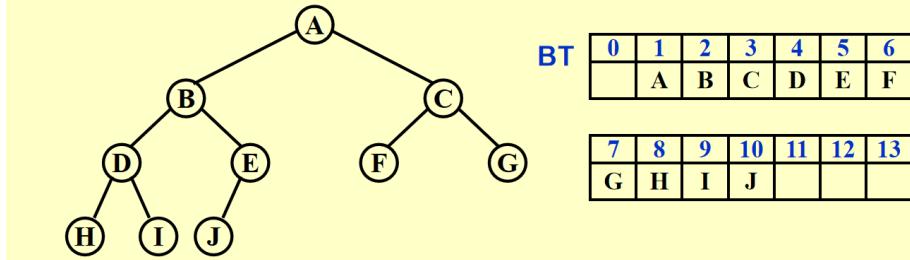


Figure 22: 数组存储

### 2. 父子节点查找

(I) 若第一个节点索引为 1, 则对于节点  $i$ , 其父节点索引为  $\frac{i}{2}$ , 左子节点索引为  $2i$ , 右子节点索引为  $2i + 1$ , 如下图所示:

$$(1) \text{ index of } parent(i) = \begin{cases} \lfloor i/2 \rfloor & \text{if } i \neq 1 \\ \text{None} & \text{if } i = 1 \end{cases}$$

$$(2) \text{ index of } left\_child(i) = \begin{cases} 2i & \text{if } 2i \leq n \\ \text{None} & \text{if } 2i > n \end{cases}$$

$$(3) \text{ index of } right\_child(i) = \begin{cases} 2i + 1 & \text{if } 2i + 1 \leq n \\ \text{None} & \text{if } 2i + 1 > n \end{cases}$$

Figure 23: 索引从 1 开始的堆

(II) 若第一个节点索引为 0, 则对于节点  $i$ , 其父节点索引为  $\frac{i-1}{2}$ , 左子节点索引为  $2i + 1$ , 右子节点索引为  $2i + 2$ , 如下图所示:

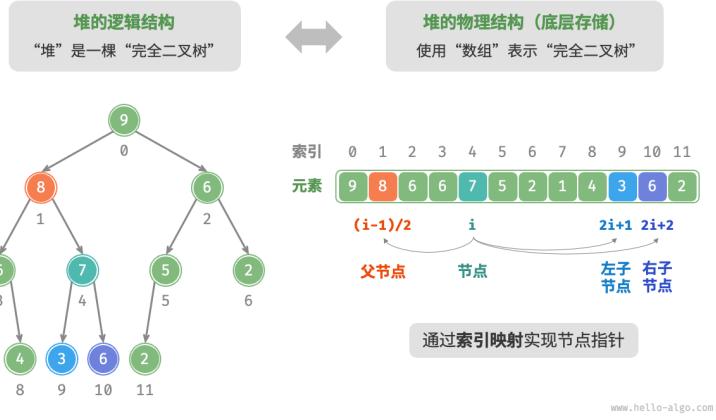


Figure 24: 索引从 0 开始的堆

### 3. 堆化 | Heapify

#### 3.1) 节点入堆 | Insert a Node into the Heap

**Note**

此情况为从底至顶的堆化

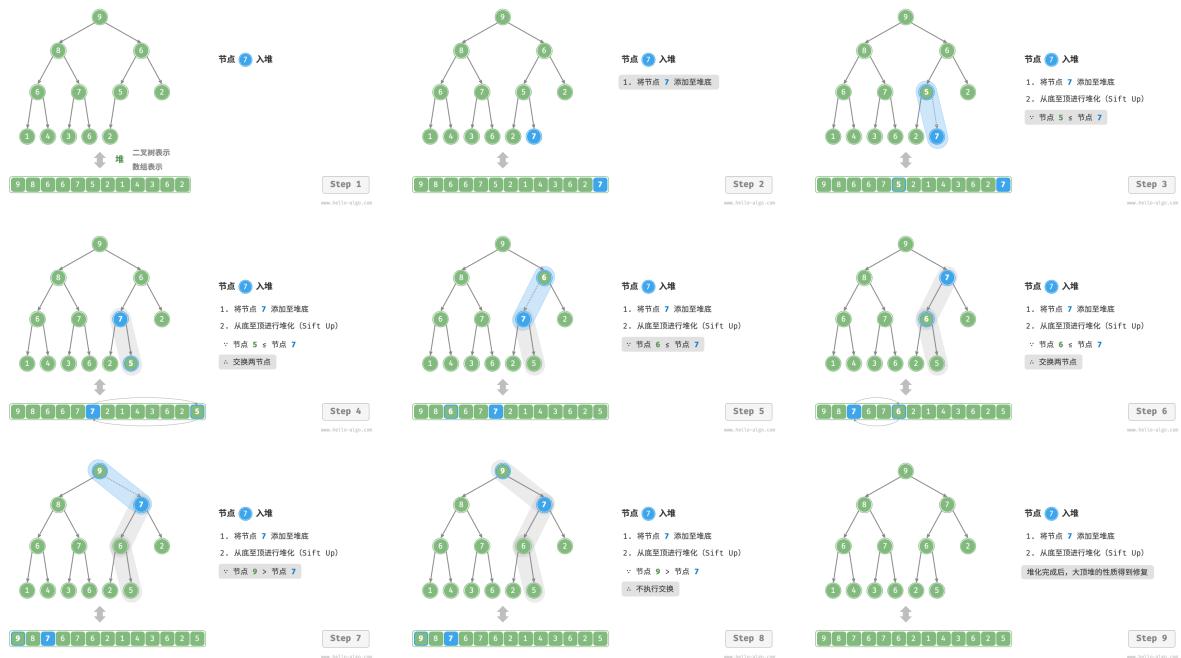
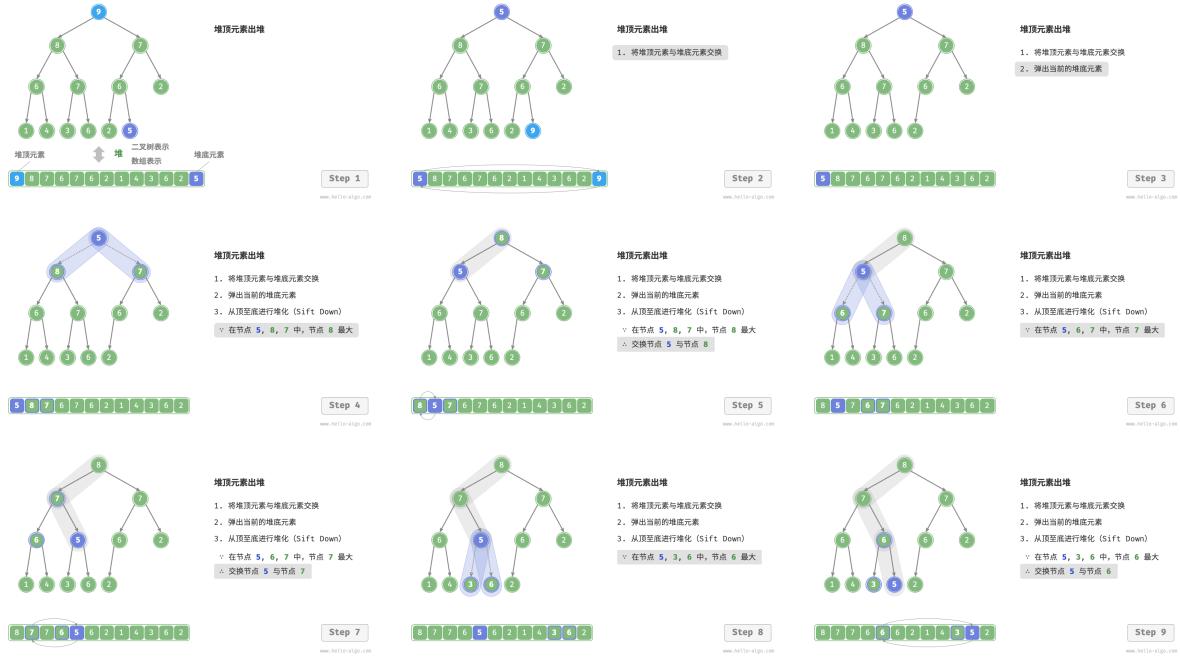


Figure 25: Insert a Node into the Heap

#### 3.2) 节点出堆 | Delete a Node from the Heap

**Note**

此情况为从顶至底的堆化——先删除根节点，然后从根节点依次向下调整堆的结构

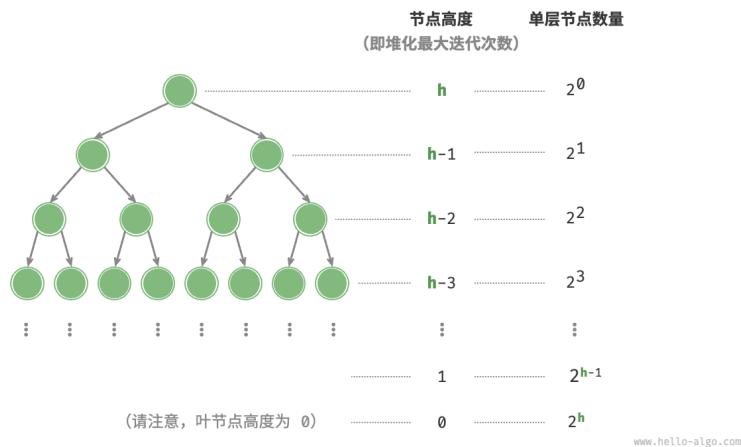


**Figure 26:** Delete a Node from the Heap

### 3.3) 建堆操作 | Build a Heap

## - Note

建堆操作是将一个无序的数组转换为一个堆的过程。可以通过 [从最后一个非叶子节点开始](#)，依次向上 [从底至顶](#) 调整每个节点，使其满足堆的性质。



**Figure 27:** Build a Heap

## 五、并查集 | The Disjoint Set

## 1. 等价关系

### (I) 关系 “R”:

A relation  $R$  is defined on a set  $S$  if for every pair of elements  $(a, b)$ ,  $a, b \in S$ ,  $a R b$  is either true or false. If  $a R b$  is true, then we say that **a is related to b**.

(II) 等价关系:

A relation  $R$  is an **equivalence relation** if it satisfies the following three properties:

- **Reflexive | 自反性:** For every element  $a$  in  $S$ ,  $a R a$  is true.
- **Symmetric | 对称性:** For every pair of elements  $(a, b)$  in  $S$ , if  $a R b$  is true, then  $b R a$  is also true.
- **Transitive | 传递性:** For every triple of elements  $(a, b, c)$  in  $S$ , if  $a R b$  and  $b R c$  are both true, then  $a R c$  is also true.

(III) Two members  $x$  and  $y$  of a set  $S$  are said to be in the same equivalence class iff  $x \sim y$ .

## 2. Union and Find

只需要注意如下 Union-Find by Size 操作即可

```
Union-Find by Size
1 SetType Find(ElementType X, DisjSet S)
2 {
3     ElementType root, trail, lead;
4     for (root = X; S[root] > 0; root = S[root])
5         ; /* find the root */
6     for (trail = X; trail != root; trail = lead)
7     {
8         lead = S[trail];
9         S[trail] = root;
10    } /* collapsing */
11   ~~~> 注意此段的Find操作会直接将节点直接指向根节点
12 }
```

## 六、图算法 | Graph Algorithms

### 1. 图的基本概念 | Basic Concepts of Graphs

图 (graph) 是一种非线性数据结构，由顶点 (vertex) 和边 (edge) 组成。我们可以将图  $G$  抽象地表示为一组顶点和一组边的集合。以下示例展示了一个包含 5 个顶点和 7 条边的图。

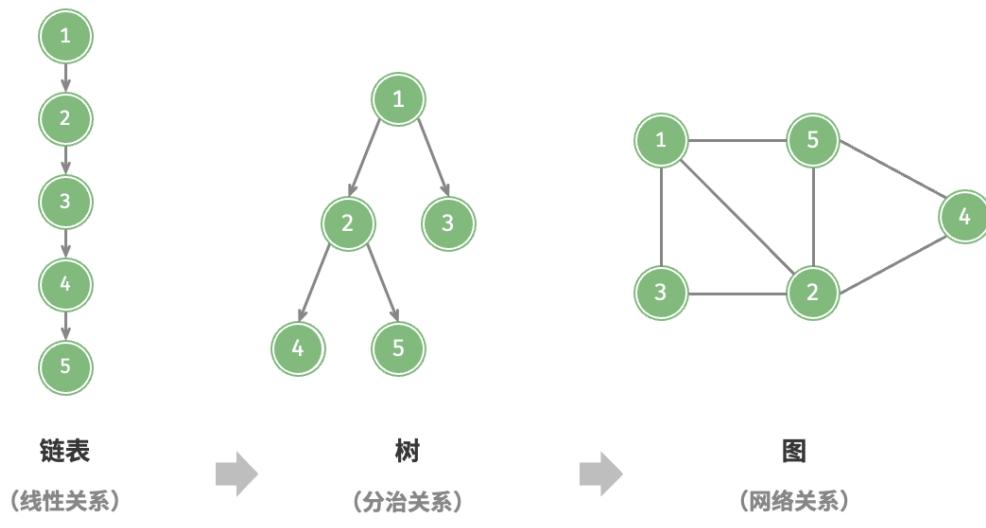
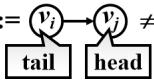


Figure 28: Graph

**Summary**

- (I)  $G(V, E)$ , 其中  $G := \text{graph}$ ,  $V = V(G) :=$  有限非空的顶点集合,  $E = E(G) :=$  有限的边集合。  
 (II) 有向图 (directed graph) 和无向图(undirected graph); 注意: 在题目中没有明确指出时, 默认是无向图

**Directed graph (digraph):**  $\langle v_i, v_j \rangle ::=$    $\neq \langle v_j, v_i \rangle$

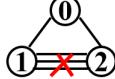
- (III) 限制: 禁止自环和重边

**Restrictions :**

(1) **Self loop** is illegal.



(2) **Multigraph** is not considered



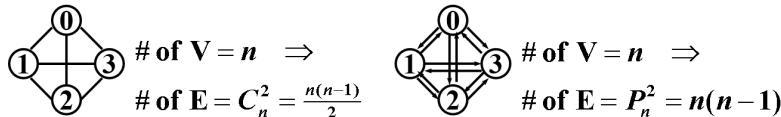
- (IV) 完全图 (complete graph) 和稀疏图 (sparse graph)

- 完全图: a graph that has the maximum number of edges
- **完全图的性质:**
  - 完全图的每一对不同的顶点都有一条边相连。
  - 完全图的边数为

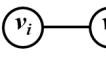
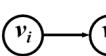
$$E = \frac{n(n-1)}{2} \text{ for undirected graph;}$$

$$E = n(n-1) \text{ for directed graph.}$$

其中  $n$  是顶点的数量。



- (V) Adjacent | 相邻

- ✓   $v_i$  and  $v_j$  are **adjacent**;  $\langle v_i, v_j \rangle$  is **incident on**  $v_i$  and  $v_j$
- ✓   $v_i$  is **adjacent to**  $v_j$ ;  $v_j$  is **adjacent from**  $v_i$ ;  $\langle v_i, v_j \rangle$  is **incident on**  $v_i$  and  $v_j$
- ✓ **Subgraph  $G' \subset G$**  ::=  $V(G') \subseteq V(G)$  &&  $E(G') \subseteq E(G)$
- ✓ **Path ( $\subset G$ ) from  $v_p$  to  $v_q$**  ::= { $v_p, v_{i1}, v_{i2}, \dots, v_{in}, v_q$ } such that  $(v_p, v_{i1}), (v_{i1}, v_{i2}), \dots, (v_{in}, v_q)$  or  $\langle v_p, v_{i1} \rangle, \dots, \langle v_{in}, v_q \rangle$  belong to  $E(G)$
- ✓ **Length of a path** ::= number of edges on the path
- ✓ **Simple path** ::=  $v_{i1}, v_{i2}, \dots, v_m$  are distinct
- ✓ **Cycle** ::= simple path with  $v_p = v_q$
- ✓  $v_i$  and  $v_j$  in an undirected  $G$  are **connected** if there is a path from  $v_i$  to  $v_j$  (and hence there is also a path from  $v_j$  to  $v_i$ )
- ✓ An undirected graph  $G$  is **connected** if every pair of distinct  $v_i$  and  $v_j$  are connected

- (VI) 强连通图 | Strongly Connected Graph

- (VII) 弱连通图 | Weakly Connected Graph

- 有向图中任意两个顶点之间都有路径相连的图称为强连通图。
- 如果有向图中存在从一个顶点到另一个顶点的路径, 但反之不成立, 则称为弱连通图。
- 强连通图的性质: 如果一个有向图是强连通的, 则它的任意两个顶点之间都有路径相连。

- (VIII) (Tree) 树 | A connected acyclic graph

- (IX) 有向无环图 | Directed Acyclic Graph (DAG)
- (X) 有向图的入度和出度 | In-Degree and Out-Degree of Directed Graphs
- (XI) 无向图的度 | Degree of Undirected Graphs
- (XII) (Connected) Component of an undirected G ::= the maximal connected subgraph | 无向图的连通分量
  - 无向图的连通分量是指无向图中任意两个顶点之间都有路径相连的子图。
  - 连通分量的性质：无向图的连通分量是无向图中最大的连通子图。
- (XIII) 通过所有节点的度来计算边数

 Given G with  $n$  vertices and  $e$  edges, then

$$e = \left( \sum_{i=0}^{n-1} d_i \right) / 2 \text{ where } d_i = \text{degree}(v_i)$$

## 2. 图的存储 | Graph Storage

### 2.1) 邻接矩阵 | Adjacency Matrix

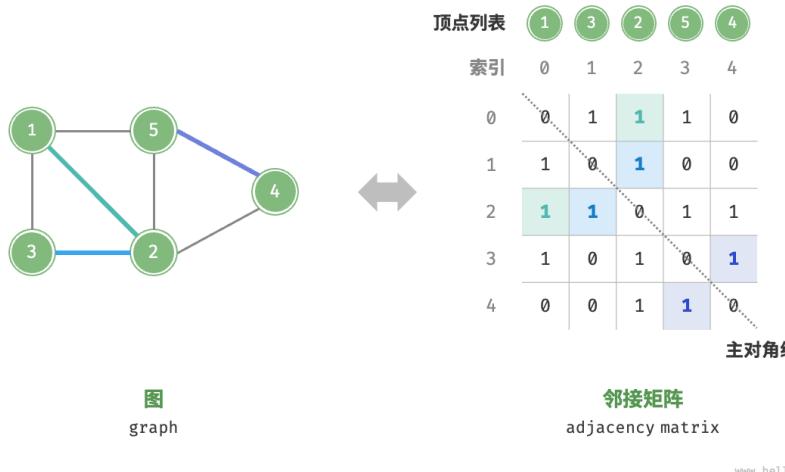


Figure 29: 邻接矩阵

邻接矩阵具有以下特性。

- 在简单图中，顶点不能与自身相连，此时邻接矩阵主对角线元素没有意义。
- 对于无向图，两个方向的边等价，此时邻接矩阵关于主对角线对称。
- 将邻接矩阵的元素从 0 和 1 替换为权重，则可表示有权图

使用邻接矩阵表示图时，我们可以直接访问矩阵元素以获取边，因此增删查改操作的效率很高，时间复杂度均为  $O(1)$ 。然而，矩阵的空间复杂度为  $O(n^2)$ ，内存占用较多。

#### Note

- (I) 如果无向图的存储要节省空间——将邻接矩阵转换为一维数组(实际上是用时间来换取空间)

The trick is to store the matrix as a 1-D array:  
 $\text{adj\_mat}[n(n+1)/2] = \{a_{11}, a_{21}, a_{22}, \dots, a_{n1}, \dots, a_{nn}\}$   
 The index for  $a_{ij}$  is  $(i * (i - 1) / 2 + j)$ .

(II) 通过邻接矩阵计算节点的度:

$$\text{Degree}(i) = \sum_{j=0}^{n-1} (\text{adj\_mat}[i][j]) + \sum_{j=0}^{n-1} (\text{adj\_mat}[j][i]) \text{ 有向图}$$

$$\text{Degree}(i) = \sum_{j=0}^{n-1} (\text{adj\_mat}[i][j]) \text{ 无向图}$$

## 2.2 邻接表 | Adjacency List

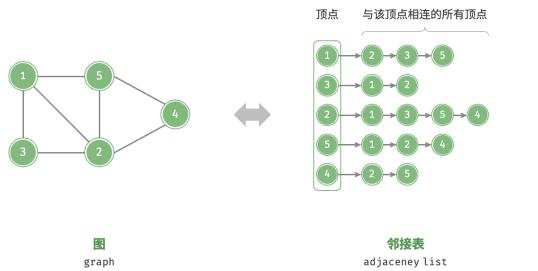


Figure 30: 邻接表

邻接表仅存储实际存在的边，而边的总数通常远小于  $n^2$ ，因此它更加节省空间。然而，在邻接表中需要通过遍历链表来查找边，因此其时间效率不如邻接矩阵。

## 2.3 效率对比

设图中共有  $n$  个顶点和  $m$  条边，表 Table 1 对比了邻接矩阵和邻接表的时间效率和空间效率。请注意，邻接表（链表）对应本文实现，而邻接表（哈希表）专指将所有链表替换为哈希表后的实现。

操作	邻接矩阵	邻接表（链表）
判断是否邻接	$O(1)$	$O(n)$
添加边	$O(1)$	$O(1)$
删除边	$O(1)$	$O(n)$
添加顶点	$O(n)$	$O(1)$
删除顶点	$O(n^2)$	$O(n + m)$
内存空间占用	$O(n^2)$	$O(n + m)$

Table 1: 邻接矩阵与邻接表对比

观察表 Table 1，似乎邻接表（哈希表）的时间效率与空间效率最优。但实际上，在邻接矩阵中操作的效率更高，只需一次数组访问或赋值操作即可。综合来看，邻接矩阵体现了“以空间换时间”的原则，而邻接表体现了“以时间换空间”的原则。

### 3. 图的基础操作 | Basic Operations of Graphs

#### 3.1) 基于邻接矩阵 | Adjacency Matrix

(以无向图为例)

- **添加或删除边:** 直接在邻接矩阵中修改指定的边即可，使用  $O(1)$  时间。而由于是无向图，因此需要同时更新两个方向的边。

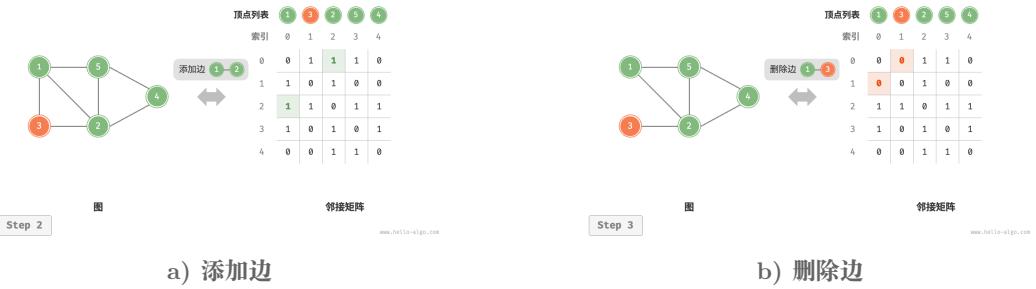


Figure 31: 添加或删除边

- **添加顶点:** 在邻接矩阵的尾部添加一行一列，并全部填 0 即可，使用  $O(n)$  时间。
- **删除顶点:** 在邻接矩阵中删除一行一列。当删除首行首列时达到最差情况，需要将  $(n - 1)^2$  个元素“向左上移动”，从而使用  $O(n^2)$  时间。
- **初始化:** 传入  $n$  个顶点，初始化长度为  $n$  的顶点列表 vertices，使用  $O(n)$  时间；初始化  $n \times n$  大小的邻接矩阵 adjMat，使用  $O(n^2)$  时间。

代码相对简单，略

#### 3.2) 基于邻接表 | Adjacency List

(以无向图为例) 设无向图的顶点总数为  $n$ 、边总数为  $m$

- **添加边:** 在顶点对应链表的末尾添加边即可，使用  $O(1)$  时间。因为是无向图，所以需要同时添加两个方向的边。
- **删除边:** 在顶点对应链表中查找并删除指定边，使用  $O(m)$  时间。在无向图中，需要同时删除两个方向的边。
- **添加顶点:** 在邻接表中添加一个链表，并将新增顶点作为链表头节点，使用  $O(1)$  时间。
- **删除顶点:** 需遍历整个邻接表，删除包含指定顶点的所有边，使用  $O(n + m)$  时间。
- **初始化:** 在邻接表中创建  $n$  个顶点和  $2m$  条边，使用  $O(n + m)$  时间。

代码实现：

```

Graph

#include<stdio.h>
#include<string.h>
#include<stdlib.h>
#include <assert.h>

#define MAX_SIZE 1000

typedef int Vertex;

/*节点结构体*/
struct AdjListNode{
    Vertex val;
    struct AdjListNode* next;
};


```

```

typedef struct AdjListNode* Node;

/*使用邻接表实现无向图*/

struct GraphAdjList{
    Node heads[MAX_SIZE]; //Ptr
    int size;      //Number of Nodes
};

typedef struct GraphAdjList* Graph;

/*Create a new Graph*/
Graph createGraph(){
    Graph newGraph = (Graph)malloc(sizeof(struct GraphAdjList));
    newGraph->size = 0;
    for(int i = 0; i < MAX_SIZE; i++){
        newGraph->heads[i] = NULL;
    }
    return newGraph;
}

/*Create NewNodes*/
Node createNode(Vertex val){
    Node newNode = (Node)malloc(sizeof(struct AdjListNode));
    newNode->next = NULL;
    newNode->val = val;
    return newNode;
}

/*Add Nodes*/
void addVertex(Graph g, Vertex vet){
    assert(g != NULL && g->size < MAX_SIZE);
    Node newNode = (Node)malloc(sizeof(struct AdjListNode));
    newNode->next = NULL;
    newNode->val = vet;
    g->heads[g->size++] = newNode;
}

/*Add or Delete Edges helper function*/
Node findGraph(Graph g, Vertex vet){
    // for(int i = 0 ; i < MAX_SIZE; i++){
    for(int i = 0 ; i < g->size; i++){ // 修正1: 应该是g->size而不是MAX_SIZE
        // if(g->heads[i]->val == vet){
        if(g->heads[i]!= NULL && g->heads[i]->val == vet){ // 修正2: 检查NULL指针
            // return g->heads;
            return g->heads[i]; // 修正3: 返回g->heads[i]而不是g->heads
        }
    }
    printf("Number ERROR of findGraph");
    return NULL;
}

/*Add Edges*/
void addEdges(Graph g, Vertex a, Vertex b){
    Node head_a = findGraph(g, a);
    Node head_b = findGraph(g, b);

    if(head_a == NULL || head_b == NULL){ // 修正4: 检查是否找到顶点
        printf("Vertex not found\n");
        return;
    }

    Node ptr_a = createNode(a);
    Node ptr_b = createNode(b);
}

```

```

//这里使用头插法
// ptr_a->next = head_a->next;
// head_a->next = ptr_a;

// ptr_b->next = head_b->next;
// head_b->next = ptr_b;
ptr_b->next = head_a->next;
head_a->next = ptr_b;

ptr_a->next = head_b->next;
head_b->next = ptr_a;
}

~~> 这里注释的部分是错的，插入的时候插反了

/*Delete helperFunction*/
void deleteEdgeHelper(Node head, Vertex vex){
    Node ptr = head->next, pre = head;
    // while(ptr != NULL || ptr->val != vex){
    while(ptr != NULL && ptr->val != vex){ // 修正6：应该是&&而不是 ||
        pre = ptr;
        ptr = ptr->next;
    }
    if(ptr == NULL){
        printf("ERROR of DELETE");
        return;
    }
    pre->next = ptr->next;
    free(ptr);
}

/*Delete Edges*/
void deleteEdges(Graph g, Vertex a, Vertex b){
    assert(g != NULL && g->size < MAX_SIZE);
    Node head_a = findGraph(g, a);
    Node head_b = findGraph(g, b);

    if(head_a != NULL && head_b != NULL){ // 修正8：检查顶点是否存在
        deleteEdgeHelper(head_a, b);
        deleteEdgeHelper(head_b, a);
    }
}

void deleteVertex(Graph g, Vertex vex){
    Node head = findGraph(g, vex);
    Node ptr = head, pre = NULL;

    //释放内存
    while(ptr){
        pre = ptr;
        ptr = ptr->next;
        free(pre);
    }

    for(int i = 0; i < g->size && g->heads[i] != head; i++){ // 跳过已删除的顶点
        ptr = g->heads[i];
        pre = NULL;
        while(ptr){
            pre = ptr;
            ptr = ptr->next;
            // if(ptr->val == vex){
            if(ptr && ptr->val == vex){ //修正9：加入ptr检测是否为NULL
                ~~> 这里很重要，如果要删除的是最后一个节点，那么 ptr 将等于 NULL，因此需要特别关注
                    pre->next = ptr->next;
                    free(ptr);
                    break;
            }
        }
    }
}

```

```

        }
    }

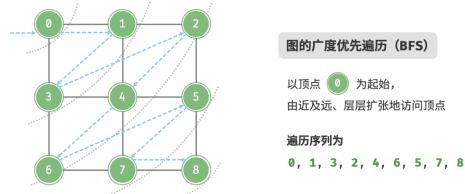
    int idx;
    for(int i = 0; i < g->size; i++){
        if(g->heads[i] == head){
            idx = i;
            break;
        }
    }
    for(int i = idx; i < g->size - 1; i++){
        g->heads[i] = g->heads[i + 1];
    }
    g->size--;
}

```

## 4. 图的遍历 | Graph Traversals

### 4.1) 广度优先遍历 | Breadth-First Search (BFS)

广度优先遍历是一种优先访问当前顶点的所有邻接顶点的遍历方式。如图 Figure 32 所示，从左上角顶点出发，访问当前顶点的所有邻接顶点，然后再访问这些邻接顶点的邻接顶点，以此类推，直至所有顶点遍历完成。



www.hello-algo.com

Figure 32: BFS

#### 4.1.1) 算法实现 | Implementation

广度优先遍历通常基于队列来实现。每次访问一个顶点时，将其所有未访问的邻接顶点入队，然后从队列中取出下一个顶点进行访问。

```

Breadth First Search
#include<stdio.h>

#define MAX_SIZE 1000
typedef int Vertex;

/* 节点队列结构体 */
typedef struct {
    Vertex *vertices[MAX_SIZE];
    int front, rear, size;
} Queue;

/* 构造函数 */
Queue *newQueue() {
    Queue *q = (Queue *)malloc(sizeof(Queue));
    q->front = q->rear = q->size = 0;
    return q;
}

```

```

}

/* 判断队列是否为空 */
int isEmpty(Queue *q) {
    return q->size == 0;
}

/* 入队操作 */
void enqueue(Queue *q, Vertex *vet) {
    q->vertices[q->rear] = vet;
    q->rear = (q->rear + 1) % MAX_SIZE;
    q->size++;
}

/* 出队操作 */
Vertex *dequeue(Queue *q) {
    Vertex *vet = q->vertices[q->front];
    q->front = (q->front + 1) % MAX_SIZE;
    q->size--;
    return vet;
}

/* 检查顶点是否已被访问 */
int isVisited(Vertex **visited, int size, Vertex *vet) {
    // 遍历查找节点, 使用 O(n) 时间
    for (int i = 0; i < size; i++) {
        if (visited[i] == vet)
            return 1;
    }
    return 0;
}

/* 广度优先遍历 */
// 使用邻接表来表示图, 以便获取指定顶点的所有邻接顶点
void graphBFS(GraphAdjList *graph, Vertex *startVet, Vertex **res, int *resSize, Vertex **visited,
int *visitedSize) {
    // 队列用于实现 BFS
    Queue *queue = newQueue();
    enqueue(queue, startVet);
    visited[(*visitedSize)++] = startVet;
    // 以顶点 vet 为起点, 循环直至访问完所有顶点
    while (!isEmpty(queue)) {
        Vertex *vet = dequeue(queue); // 队首顶点出队
        res[(*resSize)++] = vet; // 记录访问顶点
        // 遍历该顶点的所有邻接顶点
        AdjListNode *node = findNode(graph, vet);
        while (node != NULL) {
            // 跳过已被访问的顶点
            if (!isVisited(visited, *visitedSize, node->vertex)) {
                enqueue(queue, node->vertex); // 只入队未访问的顶点
                visited[(*visitedSize)++] = node->vertex; // 标记该顶点已被访问
            }
            node = node->next;
        }
    }
    // 释放内存
    free(queue);
}

```

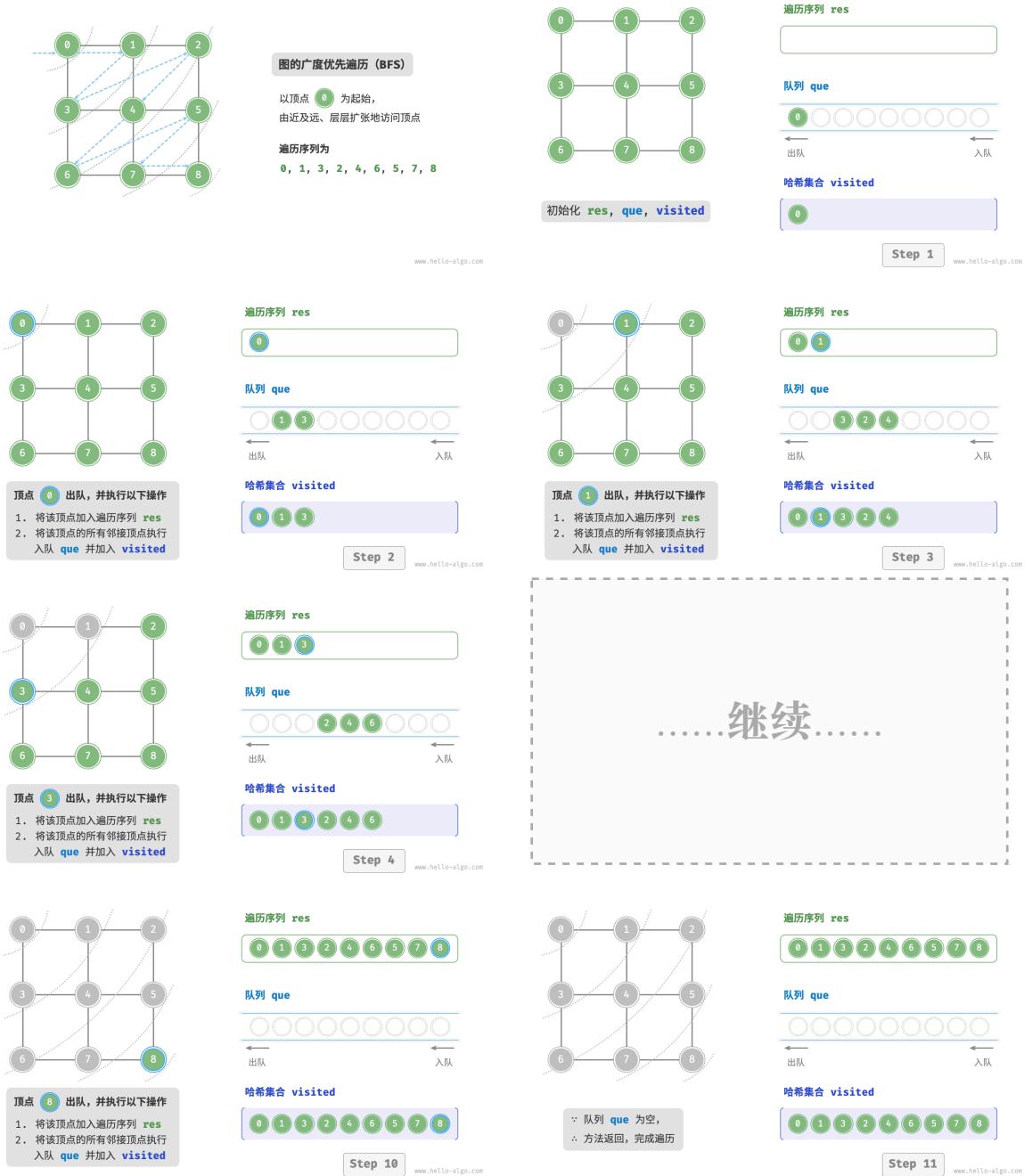


Figure 33: BFS Traversal Steps

### Tip

广度优先遍历的序列是否唯一？

不唯一。广度优先遍历只要求按“由近及远”的顺序遍历，而多个相同距离的顶点的遍历顺序允许被任意打乱。以图 Figure 32 为例，顶点 1、3 的访问顺序可以交换，顶点 2、4、6 的访问顺序也可以任意交换。

### 4.1.2) 复杂度分析

(I) **时间复杂度：**所有顶点都会从队入队一次，使用  $O(|V|)$  时间；在遍历邻接顶点的过程中，由于是无向图，因此所有边都会被访问 2 次，使用  $O(2|E|)$  时间；总体使用  $O(|V| + |E|)$  时间。

(II) **空间复杂度：**列表 res，哈希集合 visited，队列 que 中的顶点数量最多为  $|V|$ ，使用  $O(|V|)$  空间。

## 4.2) 深度优先遍历 | Depth-First Search (DFS)

深度优先遍历是一种优先走到底、无路可走再回头的遍历方式。如图 Figure 34 所示，从左上角顶点出发，访问当前顶点的某个邻接顶点，直到走到尽头时返回，再继续走到尽头并返回，以此类推，直至所有顶点遍历完成。

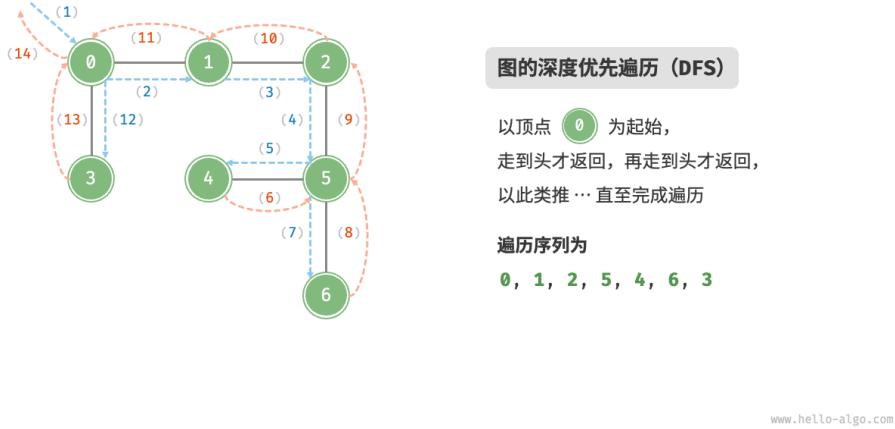


Figure 34: DFS

### 4.2.1) 算法实现 | Implementation

这种“走到尽头再返回”的算法范式通常基于递归来实现。

#### Depth First Search

```
/* 检查顶点是否已被访问 */
int isVisited(Vertex **res, int size, Vertex *vet) {
    // 遍历查找节点，使用 O(n) 时间
    for (int i = 0; i < size; i++) {
        if (res[i] == vet) {
            return 1;
        }
    }
    return 0;
}

/* 深度优先遍历辅助函数 */
void dfs(GraphAdjList *graph, Vertex **res, int *resSize, Vertex *vet) {
    // 记录访问顶点
    res[(*resSize)++] = vet;
    // 遍历该顶点的所有邻接顶点
    AdjListNode *node = findNode(graph, vet);
    while (node != NULL) {
        // 跳过已被访问的顶点
        if (!isVisited(res, *resSize, node->vertex)) {
            // 递归访问邻接顶点
            dfs(graph, res, resSize, node->vertex);
        }
        node = node->next;
    }
}

/* 深度优先遍历 */
// 使用邻接表来表示图，以便获取指定顶点的所有邻接顶点
void graphDFS(GraphAdjList *graph, Vertex *startVet, Vertex **res, int *resSize) {
    dfs(graph, res, resSize, startVet);
```

}

**Tip**

深度优先遍历的序列是否唯一？

与广度优先遍历类似，深度优先遍历序列的顺序也不是唯一的。给定某顶点，先往哪个方向探索都可以，即邻接顶点的顺序可以任意打乱，都是深度优先遍历。

以树的遍历为例，“根左右”“左根右”“左右根”分别对应前序、中序、后序遍历，它们展示了三种遍历优先级，然而这三者都属于深度优先遍历。

#### 4.2.2) 复杂度分析

- (I) **时间复杂度：**所有顶点都会被访问 1 次，使用  $O(|V|)$  时间；所有边都会被访问 2 次，使用  $O(2|E|)$  时间；总体使用  $O(|V| + |E|)$  时间。
- (II) **空间复杂度：**列表 `res`, 哈希集合 `visited` 顶点数量最多为  $|V|$ , 递归深度最大为  $|V|$ , 因此使用  $O(|V|)$  空间。

### 5. 拓扑排序 | Topological Sort

拓扑排序是对有向无环图 (DAG) 的一种线性排序，使得对于图中的每一条边  $(u, v)$ ，顶点  $u$  在顶点  $v$  之前。拓扑排序通常用于处理依赖关系，如任务调度、编译顺序等。

```
Topological Sort
typedef int Vertex
Vertex quene[MAX_SIZE];
~~> 队列创建
int font = 0, rear = 0, count = 0, size = MAX_SIZE;
void enqueue(Vertex val){
    if(count > MAX_SIZE){
        printf("maxsize!\n");
        return;
    }
    quene[rear] = val;
    rear = (rear + 1) % size;
    count++;
}

Vertex dequeue(){
    if(!count){
        printf("No elements\n");
        return -1;
    }
    Vertex temp = quene[font];
    font = (font + 1) % size;
    count--;
    return temp;
}
int findIdx(Graph g, int val){
    for(int j = 0; j < g->size; j++){
        if(g->heads[j]->val == val){
            return j;
        }
    }
    printf("error in findIdx\n");
    return -1;
}

bool topologicalSort(Graph g, int result[]) {
    if (g == NULL || g->size == 0) return true;

    int inDegree[MAX_SIZE] = {0};
```

```

// compute Indegree
for(int i = 0; i < g->size; i++){
    Node ptr = g->heads[i]->next;
    while(ptr){
        int j = findIdx(g, ptr->val);
        inDegree[j]++;
        ptr = ptr->next;
    }
}

for(int i = 0; i < g->size; i++){
    if(inDegree[i] == 0){
        enqueue(i);
    }
}
int idx = 0;
while(count){
    int j = dequeue();
    result[idx++] = g->heads[j]->val;
    Node ptr = g->heads[j]->next;
    while(ptr{
        int k = findIdx(g, ptr->val);
        inDegree[k]--;
        if(inDegree[k] == 0){
            enqueue(k);
            ~~~ 此处是算法的核心，每次做完递减之后如果是0就入队，大大减少了每次重新检索入度为0的点的次数
        }
        ptr = ptr->next;
    }
}

return idx == g->size;
}

```

## 6. 图的最短路径 | Shortest Path in Graphs

### 6.1) 无权图的最短路径 | Shortest Path in Unweighted Graphs

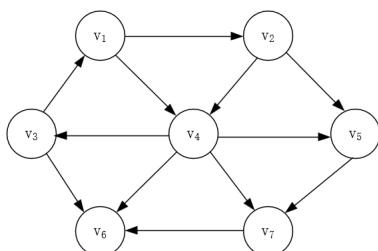


图3 一个无权有向图

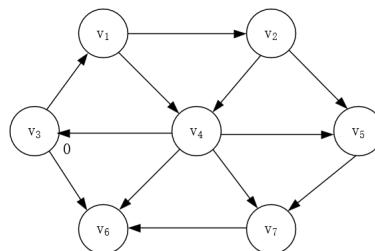


图4 将开始节点标记为通过0条边可以到达的节点后的图

CSDN @wild\_wolf

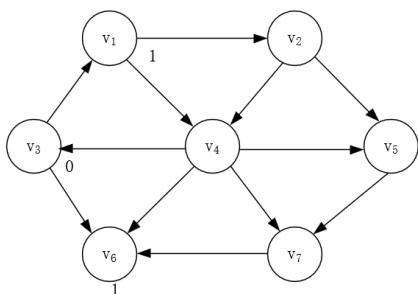


图5 找出所有从s出发路径长为1的顶点之后的图

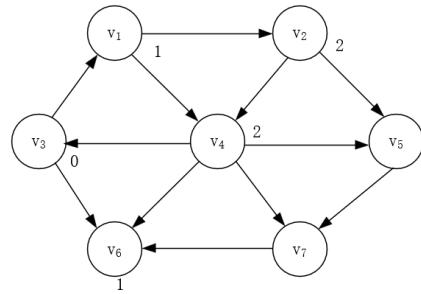


图6 找出所有从s出发路径长为2的顶点之后的图

CSDN @wild\_wolf

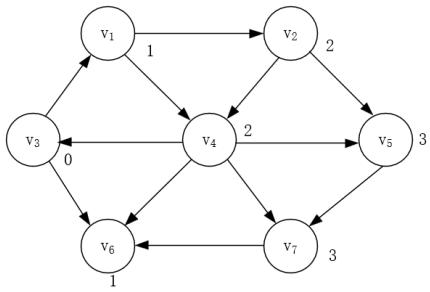


图7 最终得到的各条最短路径

顶点	known	$d_v$	$p_v$
v <sub>1</sub>	F	$\infty$	0
v <sub>2</sub>	F	$\infty$	0
v <sub>3</sub>	F	0	0
v <sub>4</sub>	F	$\infty$	0
v <sub>5</sub>	F	$\infty$	0
v <sub>6</sub>	F	$\infty$	0
v <sub>7</sub>	F	$\infty$	0

图8 用于无权最短路径计算的表的初始配置

CSDN @wild\_wolf

Figure 35: BFS Traversal and Shortest Path

无权图的最短路径可以通过**广度优先搜索（BFS）**来实现。BFS 会从起点开始，逐层访问所有邻接顶点，直到找到终点为止。由于 BFS 是按层次遍历的，因此它保证了找到的路径是最短的。

下图所示算法最坏复杂度为 $O(|V|^2)$ ，效果并不好，实际应用中通常使用 BFS 来实现无权图的最短路径。

#### Shortest Path in Unweighted Graphs I

```
void Unweighted(Table T)
{
    int CurrDist;
    Vertex V, W;
    for (CurrDist = 0; CurrDist < NumVertex; CurrDist++){
        for (each vertex V)
            if (!T[V].Known && T[V].Dist == CurrDist)
            {
                T[V].Known = true;
                for (each W adjacent to V)
                    if (T[W].Dist == Infinity){
                        T[W].Dist = CurrDist + 1;
                        T[W].Path = V;
                    } /* end-if Dist == Infinity */
            } /* end-if !Known && Dist == CurrDist */
    } /* end-for CurrDist */
}
```

#### Shortest Path in Unweighted Graphs II(BFS Version)

```
void Unweighted(Table T)
{ /* T is initialized with the source vertex S given */
    Queue Q;
    Vertex V, W;
    Q = CreateQueue(NumVertex);
    MakeEmpty(Q);
    Enqueue(S, Q); /* Enqueue the source vertex */
    while (!IsEmpty(Q)){
        V = Dequeue(Q);
        T[V].Known = true; /* not really necessary */
        for (each W adjacent to V)
            if (T[W].Dist == Infinity){
                T[W].Dist = T[V].Dist + 1;
                T[W].Path = V;
                Enqueue(W, Q);
                ~~> 此处即为与BFS的区别
            } /* end-if Dist == Infinity */
    } /* end-while */
    DisposeQueue(Q); /* free memory */
}
```

## 6.2) 有权图的最短路径 | Shortest Path in Weighted Graphs

有权图的最短路径通常使用 Dijkstra 算法来实现。Dijkstra 算法是一种贪心算法，它通过逐步扩展已知最短路径的顶点集合，直到找到从起点到终点的最短路径。

### 6.2.1) 算法 | Algorithm

Dijkstra 算法是一种贪心算法，用于计算从起点到所有其他顶点的最短路径。

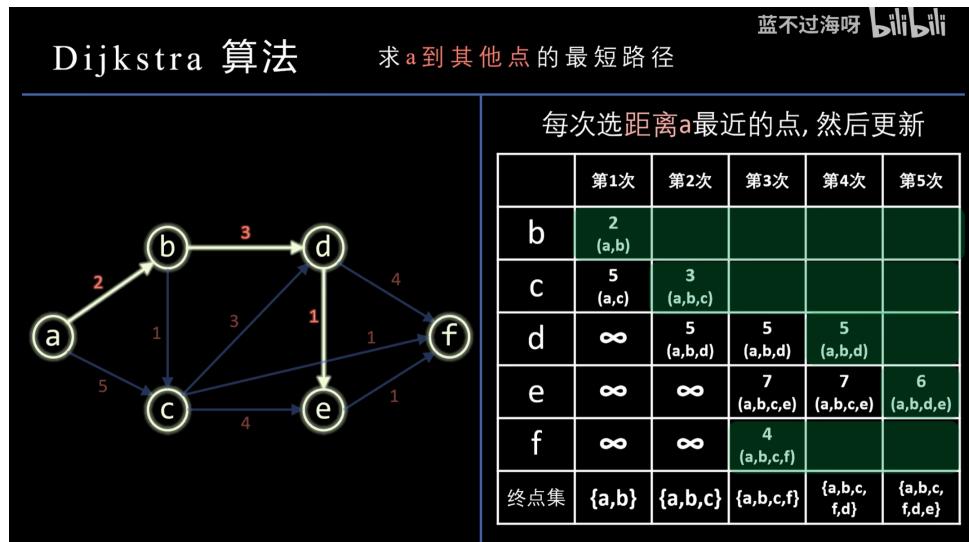


Figure 36: Dijkstra Algorithm

### 6.2.2) 具体实现

```

Dijkstra's Algorithm

/* 初始化Dijkstra表 */
void initializeTable(Graph g, Table T, Vertex start) {
    // 初始化
    for(int i = 0; i < g->size; i++){
        T[i].Dist = INT_MAX;
        T[i].Path = NotAVertex;
        T[i].Known = false;
    }
    // 设置源起点
    int startIdx = getVertexIndex(g, start);

    T[startIdx].Dist = 0;
}

/* 找到距离最小的未知顶点 */
Vertex findSmallestUnknown(Graph g, Table T) {
    int minDist = INT_MAX;
    Vertex minVertex = NotAVertex;
    for(int i = 0; i < g->size; i++){
        if((!T[i].Known) && (minDist > T[i].Dist)){
            minDist = T[i].Dist;
            minVertex = i;
        }
    }
    return minVertex;
}

/* Dijkstra算法实现 */
void Dijkstra(Graph g, Table T) {
    Vertex V, W;
}

```

```

while(1){
    V = findSmallestUnknown(g, T);
    if(V == NotAVertex){
        break;
    }
    T[V].Known = true;
    Node ptr = g->heads[V]->next;
    while(ptr){
        Vertex W = getVertexIndex(g, ptr->val);
        // if(W != -1 && T[W].Dist > ptr->weight){
        //     T[W].Dist = ptr->weight;
        // }
        ~~> 更新距离时需要考虑当前顶点到其他点的距离和边的权重
        if(W != -1 && (!T[W].Known)){
            if(T[V].Dist + ptr->weight < T[W].Dist){
                ~~> dist是当前顶点到其他点的距离，而不是从起点到其他点的距离
                T[W].Dist = T[V].Dist + ptr->weight;
                T[W].Path = V;
            }
        }
        ptr = ptr->next;
    }
}

```

### 6.2.3) 复杂度分析

**时间复杂度:** Dijkstra 算法的时间复杂度为  $O(|V|^2)$ ，其中  $|V|$  是图中顶点的数量。对于稀疏图，可以使用优先队列优化到  $O((|V| + |E|) \log |V|)$ ，其中  $|E|$  是边的数量。

## 6.3) 负边权图 | Graphs with Negative Edge Costs

负边权图是指图中存在边的权重为负数的情况。Dijkstra 算法无法处理负边权图，因为它假设一旦找到某个顶点的最短路径，就不会再更新该顶点的距离。

使用 SPFA 算法，Shortest Path Faster Algorithm，可以解决负边权图的最短路径问题。

```

SPFA Algorithm
void WeightedNegative(Table T[]) {
    Queue Q;
    Vertex V, W;

    Q = CreateQueue(NumVertex);
    MakeEmpty(Q);
    Enqueue(S, Q);

    while (!IsEmpty(Q)) {
        V = Dequeue(Q);
        for (each W adjacent to V) {
            if (T[V].Dist + Cvw < T[W].Dist) {
                T[W].Dist = T[V].Dist + Cvw;
                T[W].Path = V;
                if (W is not already in Q)
                    Enqueue(W, Q);
            }
        }
    }

    DisposeQueue(Q);
}

```

## 7. 网络流问题

网络流问题是指在一个有向图中，边上有容量限制，要求在源点和汇点之间找到最大流的路径。常用的算法有 Ford-Fulkerson 算法和 Edmonds-Karp 算法。

### 7.1) Ford-Fulkerson 算法 | Ford-Fulkerson Algorithm

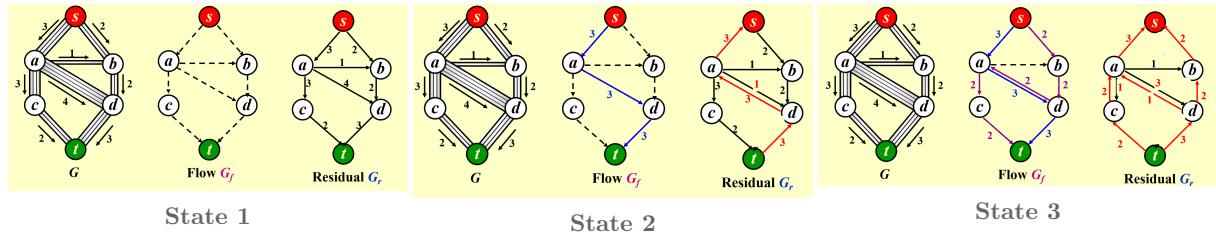


Figure 37: Ford-Fulkerson 算法流程图

### 7.2) 具体实现

#### Ford-Fulkerson Algorithm

```
#include <stdio.h>           // 标准输入输出库
#include <stdlib.h>          // 标准库函数(malloc, free等)
#include <limits.h>           // 定义各种限制常量(如INT_MAX)
#include <stdbool.h>          // 布尔类型支持(true/false)
#include <string.h>           // 字符串处理函数

#define MAXV 100              // 定义最大顶点数为100
#define INF INT_MAX            // 定义无穷大为整数最大值

// ===== 数据结构定义 =====

// 流网络结构体 - 存储整个网络的信息
typedef struct {
    int capacity[MAXV][MAXV]; // 容量矩阵: capacity[i][j] = 从顶点i到顶点j的边容量
    int flow[MAXV][MAXV];     // 流量矩阵: flow[i][j] = 从顶点i到顶点j当前的流量
    int residual[MAXV][MAXV]; // 剩余容量矩阵: residual[i][j] = 从i到j的剩余容量
    int V;                   // 顶点数量
} FlowNetwork;

// ===== 初始化函数 =====

// 初始化流网络 - 将所有矩阵元素设为0
void InitFlowNetwork(FlowNetwork* fn, int V) {
    fn->V = V;               // 设置顶点数

    // 双重循环遍历所有可能的顶点对
    for (int i = 0; i < V; i++) {
        for (int j = 0; j < V; j++) {
            fn->capacity[i][j] = 0; // 初始时所有边容量为0
            fn->flow[i][j] = 0;     // 初始时所有边流量为0
            fn->residual[i][j] = 0; // 初始时所有剩余容量为0
        }
    }
}

// ===== 图构建函数 =====

// 添加一条从from到to容量为capacity的边
void AddEdge(FlowNetwork* fn, int from, int to, int capacity) {
    fn->capacity[from][to] = capacity; // 设置边的容量
    fn->residual[from][to] = capacity; // 初始剩余容量等于容量(因为初始流量为0)
}
```

```

    // 注意: 这里只添加正向边, 反向边在UpdateResidualNetwork中处理
}

// ===== 剩余网络更新 =====

// 根据当前流量更新剩余网络
void UpdateResidualNetwork(FlowNetwork* fn) {
    // 遍历所有顶点对
    for (int i = 0; i < fn->V; i++) {
        for (int j = 0; j < fn->V; j++) {
            // 正向剩余容量 = 原始容量 - 当前流量
            fn->residual[i][j] = fn->capacity[i][j] - fn->flow[i][j];

            // 如果正向边有流量, 则反向边有剩余容量(允许流量"撤销")
            if (fn->flow[i][j] > 0) {
                fn->residual[j][i] = fn->flow[i][j];
                // 这行很关键: 反向边的剩余容量 = 正向边的当前流量
                // 这样可以"撤销"之前的流量决策
            }
        }
    }
}

// ===== DFS寻找增广路径 =====

// 使用深度优先搜索寻找从current到sink的增广路径
bool DFS_FindAugmentingPath(FlowNetwork* fn, int current, int sink,
                             bool visited[], int path[], int* pathLen, int* minFlow) {

    // 递归终止条件: 如果当前顶点就是汇点
    if (current == sink) {
        path[(*pathLen)++] = current;      // 将汇点加入路径
        return true;                      // 成功找到路径
    }

    // 标记当前顶点为已访问, 防止回路
    visited[current] = true;

    // 将当前顶点加入路径
    path[(*pathLen)++] = current;

    // 尝试访问所有邻接顶点
    for (int next = 0; next < fn->V; next++) {
        // 检查是否可以访问next顶点:
        // 1. next未被访问过
        // 2. 从current到next有剩余容量
        if (!visited[next] && fn->residual[current][next] > 0) {

            // 获取当前边的剩余容量
            int currentFlow = fn->residual[current][next];

            // 更新路径的瓶颈容量(最小剩余容量)
            if (currentFlow < *minFlow) {
                *minFlow = currentFlow;
            }

            // 递归搜索: 从next顶点继续寻找到sink的路径
            if (DFS_FindAugmentingPath(fn, next, sink, visited, path, pathLen, minFlow)) {
                return true;      // 如果找到路径, 立即返回成功
            }
        }
    }
}

// 如果所有邻接顶点都无法到达汇点, 进行回溯

```

```

        (*pathLen)--;           // 从路径中移除当前顶点
        return false;          // 返回失败
    }

// ===== Ford-Fulkerson主算法 =====

// Ford-Fulkerson算法主函数
int FordFulkerson(FlowNetwork* fn, int source, int sink) {
    int maxFlow = 0;         // 初始化最大流为0

    // 主循环: 不断寻找增广路径直到无路径可找
    while (true) {
        // 每次循环重新初始化搜索状态
        bool visited[MAXV] = {false};   // 访问标记数组, 全部初始化为false
        int path[MAXV];                // 存储找到的路径
        int pathLen = 0;                // 路径长度
        int minFlow = INF;              // 路径的瓶颈容量, 初始化为无穷大

        // 使用DFS寻找从source到sink的增广路径
        if (!DFS_FindAugmentingPath(fn, source, sink, visited, path, &pathLen, &minFlow)) {
            break;      // 如果找不到增广路径, 算法结束
        }

        // 找到增广路径后, 沿路径更新流量
        for (int i = 0; i < pathLen - 1; i++) {
            int u = path[i];           // 路径中的当前顶点
            int v = path[i + 1];        // 路径中的下一个顶点

            // 检查这条边是正向边还是反向边
            if (fn->capacity[u][v] > 0) {
                // 正向边: 增加流量
                fn->flow[u][v] += minFlow;
            } else {
                // 反向边: 减少反向流量(相当于增加正向流量)
                fn->flow[v][u] -= minFlow;
            }
        }

        // 增加总流量
        maxFlow += minFlow;

        // 更新剩余网络以反映新的流量分布
        UpdateResidualNetwork(fn);
    }

    return maxFlow;    // 返回最大流值
}

```

## 8. 最小生成树 | Minimum Spanning Tree

最小生成树是指在一个无向图中, 找到一棵包含所有顶点的树, 使得树的边权和最小。常用的算法有 Prim 算法和 Kruskal 算法。

### 8.1) Prim 算法 | Prim's Algorithm

Prim 算法是一种贪心算法, 用于找到无向图的最小生成树。它从一个起始顶点开始, 逐步扩展到所有顶点, 确保每次选择的边都是当前未连接顶点中权重最小的边。

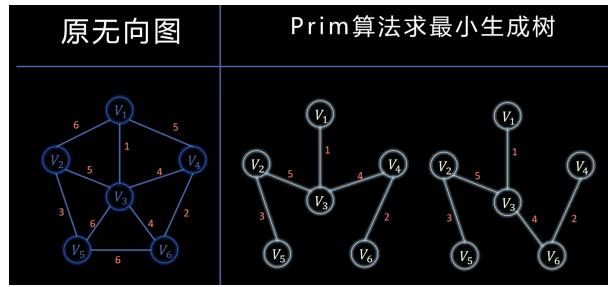


Figure 38: Prim's Algorithm

**Note**

它更适合稠密图，因为是以顶点为中心进行扩展。

## 8.2) Kruskal 算法 | Kruskal's Algorithm

Kruskal 算法是一种贪心算法，用于找到无向图的最小生成树。它通过对所有边按权重排序，然后逐步添加边到生成树中，确保不会形成环，直到包含所有顶点为止。

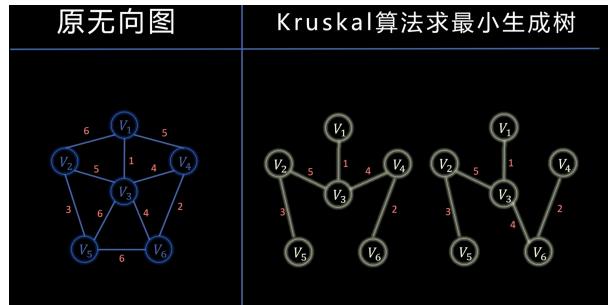


Figure 39: Kruskal's Algorithm

**Note**

它更适合稀疏图，因为是以边为中心进行扩展。

### 8.2.1) 伪代码

#### Kruskal's Algorithm

```
// Kruskal最小生成树算法 - 贪心策略, 时间复杂度O(E log E)
void Kruskal(Graph G)
{
    T = {} // 初始化生成树为空

    // 主循环: 需要选择|V|-1条边构成生成树
    while (T contains less than |V| - 1 edges && E is not empty)
    {
        choose a least cost edge(v, w) from E; // 贪心选择: 选权重最小的边
        delete(v, w) from E; // 从候选边集中移除

        if ((v, w) does not create a cycle in T) // 回路检测: 使用并查集
            add(v, w) to T; // 安全边: 加入生成树
        else
            discard(v, w); // 危险边: 丢弃以避免回路
    }

    // 验证: 连通图的生成树必须有|V|-1条边
    if (T contains fewer than |V| - 1 edges)
        Error("No spanning tree"); // 图不连通, 无法构建生成树
}
```

**Attention**

最小生成树可能不唯一，但是最小生成树的权重是唯一的。  
如果图中存在多条边的权重相同，则可能有多种不同的最小生成树。  
但是所有最小生成树的权重总和是相同的。

## 七、哈希表 | Hash Table