

数据结构与算法分析 FDS

引论

排序 sort

Quick Sort 快速排序

Heap Sort 堆排序

Insertion Sort 插入排序

Shell Sort 希尔排序

Selection Sort 选择排序

Merge Sort 归并排序

Bucket Sort 桶排序

间接/表排序

Q

散列 hashing

分离链接法

开放定址法

线性探测法 Linear probing

平方探测法 Quadratic probing

Q

表 List

Q

栈 stack

Q

队列 queue

Q

树 Tree

Binary Tree

完全二叉树 complete binary tree

insert 插入

二叉查找树 Binary Search Tree (BST)

Delete

ternary tree 三叉树

Perfect binary tree 理想二叉树

Q

堆 Heap 优先队列

Insert

上滤 percolate up

DeleteMin

下滤 percolate down

Build

IncreaseKey

DecreaseKey

关系 Relation

等价关系 equivalence relation

等价类 equivalence class

Union/Find 算法（并查算法）

Find

Union

union-by-size 按大小求并

union-by-height 按高度求并

路径压缩 path compression

Q

图 Graph

introduction

connected

割点 articulation point/cut vertex

拓扑排序 topological

Dijkstra algorithm

Maximum Network Flow 最大网络流

Dinic

MST 最小生成树

Necessary-And-Sufficient-Condition-for-Unique-MST

Prim 算法

Kruskal 算法

DFS 深度优先搜索

Find articulation point 找割点

Strongly Connected Components 强连通组件

Tarjan 算法 找割点? 还是强连通组件

Q

Questions

homework

Midterm

Pta code

True or Flase

function



数据结构与算法（周测1-算法分析） – nonlinearthink – 博客园

判断题 1.In a singly linked list of N nodes, the time complexities for query and insertion are $O(1)$...

博客园

引论

ADT abstract data type 抽象数据类型

$$O(N) \leq$$

$$\Omega(N) \geq$$

$$\Theta(N) =$$

$$o(N) <$$

排序 sort

stable 稳定: A sorting algorithm is said to be stable if two items with equal keys *in the same order* in the sorted output as they appear in the input array. That is, the order of

elements with identical keys is preserved.

Quick Sort 快速排序

Quick Sort is a sorting algorithm that works using the divide-and-conquer approach. It chooses a pivot places it in its correct position in the sorted array and partitions the smaller elements to its left and the greater ones to its right. This process is continued for the left and right parts and the array is sorted.

两边分组排序

时间复杂度: $O(N \log N)$ 平均

每一轮排序 run 都有一个数 pivot 被放到最终正确的位置上

The position of the pivot element is finalized after each partitioning.

Heap Sort 堆排序

 [Heap Sort – Data Structures and Algorithms Tutorials – GeeksforGeeks](#)

Heap sort is a comparison-based sorting technique based on Binary Heap data structure. It is similar to the selection sort where we first find the minimum element and place the minimum element at the beginning. Repeat the same process for the remaining elements.

首先使用 heapify (percolate down) 将数组转换为堆数据结构, 然后逐个删除 Max-heap 的根节点, 将其替换为堆中的最后一个节点, 然后堆化堆的根。重复此过程, 直到堆的大小大于 1。

- 从给定的输入数组构建堆。
- 重复以下步骤, 直到堆只包含一个元素:
 - 将堆的根元素 (即最大的元素) 与堆的最后一个元素交换。
 - 删除堆的最后一个元素 (现在位于正确位置)。
 - 堆砌堆的其余元素。
- 排序后的数组是通过反转输入数组中元素的顺序来获得的。

性质: unstable

时间复杂度: $O(N^2)$??

```
1  #include <stdio.h>
2
3  // Function to swap the position of two elements
4  void swap(int* a, int* b)
5  {
6
7      int temp = *a;
8      *a = *b;
9      *b = temp;
10 }
11
12 // To heapify a subtree rooted with node i
13 // which is an index in arr[].
14 // n is size of heap
15 void heapify(int arr[], int N, int i)
16 {
17     // Find largest among root,
18     // left child and right child
19
20     // Initialize largest as root
21     int largest = i;
22
23     // left = 2*i + 1
24     int left = 2 * i + 1;
25
26     // right = 2*i + 2
27     int right = 2 * i + 2;
```

Insertion Sort 插入排序

 [Insertion Sort – Data Structure and Algorithm Tutorials – GeeksforGeeks](#)

Insertion sort is a simple sorting algorithm that works similarly to the way you sort playing cards in your hands. The array is virtually split into a sorted and an unsorted part. Values from the unsorted part are picked and placed at the correct position in the sorted part.

若要按升序对大小为 N 的数组进行排序，请遍历该数组并将当前元素（键）与其前一个元素进行比较，如果关键元素小于其前一个元素，请将其与之前的元素进行比较。将较大的元素向上移动一个位置，以便为交换的元素腾出空间。

时间复杂度：

```
1 // C++ program for insertion sort
2 #include <bits/stdc++.h>
3 using namespace std;
4 // Function to sort an array using insertion sort
5 void insertionSort(int arr[], int n)
6 {
7     int i, key, j;
8     for (i = 1; i < n; i++) {
9         key = arr[i];
10
11         // Move elements of arr[0..i-1],
12         // that are greater than key,
13         // to one position ahead of their
14         // current position
15         for (j = i; j > 0 && arr[j-1] > key; j--){
16             arr[j] = arr[j-1];
17         }
18         arr[j] = key;
19     }
20 }
21 // A utility function to print an array of size n
22 void printArray(int arr[], int n)
23 {
24     int i;
25     for (i = 0; i < n; i++)
26         cout << arr[i] << " ";
27     cout << endl;
28 }
29
30 // Driver code
31 int main()
32 {
33     int arr[] = { 12, 11, 13, 5, 6 };
```

Shell Sort 希尔排序

Shell sort is mainly a variation of *Insertion Sort*. In insertion sort, we move elements only one position ahead. When an element has to be moved far ahead, many movements are involved. The idea of ShellSort is to allow the exchange of far items. In Shell sort, we make the array *h*-sorted for a large value of *h*. We keep reducing the value of *h* until it becomes 1. An array is said to be *h*-sorted if all sublists of every *h*'th element are sorted.

分组 *h*-插入排序

性质: unstable

时间复杂度:

Algorithm:

Step 1 — Start

Step 2 — Initialize the value of gap size. Example: h.

Step 3 — Divide the list into smaller sub-part. Each must have equal intervals to h.

Step 4 — Sort these sub-lists using insertion sort.

Step 5 — Repeat this step 2 until the list is sorted.

Step 6 — Print a sorted list.

Step 7 — Stop.

```
1 void Shellsort( ElementType A[ ], int N )
2 {
3     int i, j, Increment;
4     ElementType Tmp;
5     for ( Increment = N / 2; Increment > 0; Increment /= 2 )
6         /*h sequence */
7         for ( i = Increment; i < N; i++ ) { /* insertion sort */
8             Tmp = A[ i ];
9             for ( j = i; j >= Increment; j -= Increment )
10                if( Tmp < A[ j - Increment ] )
11                    A[ j ] = A[ j - Increment ];
12            else
13                break;
14            A[ j ] = Tmp;
15        } /* end for-I and for-Increment loops */
16 }
```

Hibbard 增量序列

$H_k = 2^k - 1$, 且其最坏情形下运行时间为 $O(N^{3/2})$

Selection Sort 选择排序

Merge Sort 归并排序

性质: stable

时间复杂度: $O(N \log N)$

```
merge sort

1 // C program for Merge Sort
2 #include <stdio.h>
3 #include <stdlib.h>
4 // Merges two subarrays of arr[].
5 // First subarray is arr[l..m]
6 // Second subarray is arr[m+1..r]
7 void merge(int arr[], int l, int m, int r)
8 {
9     int i, j, k;
10    int n1 = m - l + 1;
11    int n2 = r - m;
12
13    // Create temp arrays
14    int L[n1], R[n2];
15
16    // Copy data to temp arrays L[] and R[]
17    for (i = 0; i < n1; i++)
18        L[i] = arr[l + i];
19    for (j = 0; j < n2; j++)
20        R[j] = arr[m + 1 + j];
21
22    // Merge the temp arrays back into arr[l..r]
23    i = 0;
24    j = 0;
25    k = l;
26    while (i < n1 && j < n2) {
27        if (L[i] <= R[j]) {
28            arr[k] = L[i];
29            i++;
30        }
31        else {
32            arr[k] = R[j];
33            j++;
34        }
35        k++;
36    }
37
38    // Copy the remaining elements of L[],
39    // if there are any
40    while (i < n1) {
41        arr[k] = L[i];
```


Bucket Sort 桶排序

间接/表排序

排序的元素是结构大，移动指针数组进行排序

N 个数字的排列一定是由若干个独立的环组成的

- 每访问一个环，当 `table[i]==i` 时环结束

时间复杂度：（最坏）向下取整 $N/2$ 个环，每个环包括两个元素；总 $O(M * n)$ ，M 是每个元素 A 复制的时间

Q

1. If there are less than 20 inversions in an integer array, then Insertion Sort will be the best method among Quick Sort, Heap Sort and Insertion Sort. **T**
2. For the quicksort implementation with the left pointer stops at an element with the same key as the pivot during the partitioning, but the right pointer does not stop in a similar case, what is the running time when all keys are equal?
 - a. $O(\log N)$
 - b. $O(N)$
 - c. $O(N \log N)$
 - d. $O(N^2)$

The running time is $O(n^2)$ in the worst case [6]. This is because in such a situation, the partitioning doesn't effectively divide the array into smaller subproblems, leading to a degenerate case where the algorithm essentially performs a linear scan. Quicksort's typical efficiency relies on dividing the problem into subproblems, and when this doesn't occur due to equal keys, the algorithm's performance degrades. (answer from AI)

3. To sort { 49, 38, 65, 97, 76, 13, 27, 50 } in **increasing order**, which of the following is the result after the 1st run of *Shell sort* with the initial increment 4?
 - a. 13,27,38,49,50,65,76,97
 - b. 49,13,27,50,76,38,65,97
 - c. 49,76,65,13,27,50,97,38
 - d. 97,76,65,50,49,38,27,13

First Pass (Increment 4):

- Compare elements at positions 1 and 5 (49 and 76), no swap needed.
 - Compare elements at positions 2 and 6 (38 and 13), swap.
 - Compare elements at positions 3 and 7 (65 and 27), swap.
 - Compare elements at positions 4 and 8 (97 and 50), swap.
 - if not present due to length, no swap needed.
4. Among the following sorting methods, which ones will be *slowed down* if we store the elements in a **linked structure** instead of a sequential structure?
1. Insertion sort; 2. Selection Sort; 3. Bubble sort; 4. Shell sort; 5. Heap sort
- 1 and 2 only
 - 2 and 3 only
 - 3 and 4 only
 - 4 and 5 only

Heap sort是在数组中, heap本身在数组中, shell sort也是在数组中, 链表查询较慢

5. To sort N elements by heap sort, the extra space complexity is: $O(1)$
6. During the sorting, processing every element which is not yet at its final position is called a "run". To sort a list of integers using quick sort, it may reduce the total number of recursions by processing the small partition first in each run. **F**

希望平均分, 处理两个都一样的

7.

散列 hashing

冲突 collision: Two elements with different keys share the **same hash value**

装填因子 load factor: $\lambda = n/tablesize$

散列平均查找期望是 $O(1)$, 几乎与关键字空间 n 无关

- 以较小的装填因子为前提, 以空间换时间
- 不便于顺序查找关键字、范围查找、最大最小值查找等

分离链接法

把所有有冲突的 key 用链表串联在一起

装填因子可能超过 1，成功查找期望略大于不成功

链表储存效率和查找效率比较低

关键字删除不需要“懒惰删除”法

太小的装填因子可能浪费空间，太大将付出时间代价

开放定址法

如果发生第 i 次冲突，探测的下一个地址 $+d_i$

装填因子越大，（不）成功查找期望次数越大（指数级增长，不成功 > 成功）；装填因子较小时，各种期望探测次数都不大且比较接近。

散列表是个数组，储存效率高，随机查找，有聚集现象

线性探测法 Linear probing

$$d_i = i$$

平均查找长度（次数）一般失败 > 成功

成功查找长度 ASL_s：散列中每个元素要找 x_i 次（ x_i =冲突次数+1），相加取平均（ \div 实际哈希表元素个数）

失败查找长度 ASL_u：找不在散列中的元素，对于每个哈希值 $h(x)$ ，照样 $\text{mod} +$ 后移找，若是遇到空格则证明不在散列中，此时的查找次数 x_i 相加取平均

平方探测法 Quadratic probing

增量序列：1, -1, 2^2 , -2^2 , 3^2 , -3^2 , ..., q^2 , $-q^2$ 且 $q \leq \lfloor \text{tablesize}/2 \rfloor$

可能出现表有位置但找不到的情况（ i^2 也一样）

使用平方探测法：

当表的大小是素数且表有一半是空的时候，总能插入一个新的元素

如果表的大小是形如 $4k+3$ 的素数, 使用 $F(i) = + - i^2$, 那么整个表都能被探测到

Q

1. Which of the following statements about HASH is true?
 - a. the expected number of probes for insertions is greater than that for successful searches in linear probing method
 - b. insertions are generally quicker than deletions in separate chaining method
 - c. if the table size is prime and the table is at least half empty, a new element can always be inserted with quadratic probing
 - d. all of the above
2. The average search time of searching a hash table with N elements is:
 - a. $O(1)$
 - b. $O(\log N)$
 - c. $O(N)$
 - d. cannot be determined

表 List

Q

For a **sequentially stored linear list** of length N , the time complexities for **query** and **insertion** are $O(1)$ and $O(N)$, respectively. T

栈 stack

栈 (Stack) : 是只允许在一端进行插入或删除的线性表。首先栈是一种线性表, 但限定这种线性表只能在某一端进行插入和删除操作。

Last-in-First-out

▼ Push&Pop in linkedlist

```
1 void Push(Elementtype X, Stack S){
2     PtrtoNode Tmp;
3     //略去判断是不是NULL
4     Tmp->Element=X;
5     Tmp->Next=S->Next;//换表头
6     S->Next=Tmp;
7 }
8 void Pop(Stack S){
9     PtrtoNode first;
10    //略去判断是不是空
11    first=S->Next;
12    S->Next=S->Next->Next;//换表头
13    free(first);
14 }
```

▼ Push&Pop in array

```
1 void Push(Stack S){
2     //略去判断是不是满了
3     S->Array[++S->TopOfStack]=X;
4 }
5 void Pop(Stack S){
6     //略去判断是不是空
7     S->TopOfStack--;
8 }
```

Q

- Stacks and queues are lists with insertion/deletion constraints.

队列 queue

First-in-first-out

circular array

检测队列是不是空是很重要的

Q

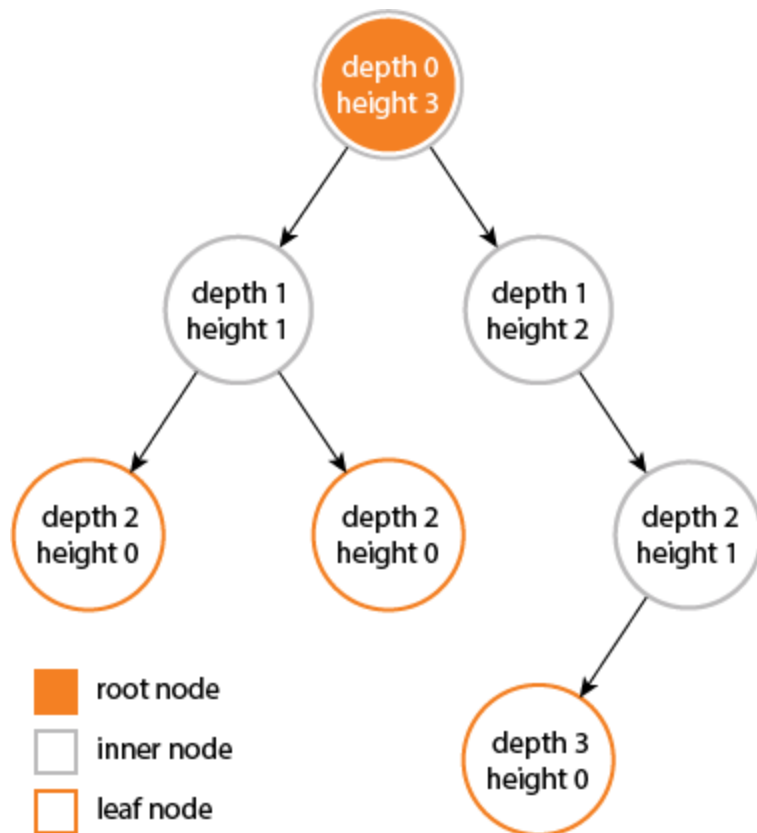
Suppose that an array of size m is used to store a **circular queue**. If the front position is $front$ and the current size is $size$, then the $rear$ element must be at $(front+size-1)\%m$.

树 Tree

深度 根 到 节点 n_i (根的深度为 0, 树的深度是它最深树叶的深度)

高度 节点 n_i 到叶子 (树的高度是根的高度)

路径 节点的一个顺序, 一棵树中从根到每个节点恰好存在一条路径



Binary Tree

度数为 0 的节点 = 度数为 2 的节点+1

完全二叉树 complete binary tree

The parent of a node at index i is located at index $\lfloor i/2 \rfloor$

insert 插入

如果队列里没有 X, `Insert(Elemnttype X, SearchTree T)` 将 X 插入到遍历路径的最后一点上

二叉查找树 Binary Search Tree (BST)

左边都比根小, 右边都比根大

树的平均深度 $O(N \log N)$

Delete

左子树最大或者右子树最小的数来代替被删掉的节点 (不是叶子)

ternary tree 三叉树

The number of leaf nodes in a ternary tree (三叉树) is only related to the number of degree 2 nodes and that of degree 3 nodes, namely, it has nothing to do with the number of degree 1 nodes.

Perfect binary tree 理想二叉树

满二叉树, 是一种特殊类型的二叉树。在理想二叉树中, 除了叶子节点之外, 每个节点都有两个子节点, 且所有叶子节点都位于同一层次上。这使得理想二叉树具有良好的平衡性。

Q

1. In a **complete binary tree** with 1102 nodes, there must be __ leaf nodes.
 - a. 79
 - b. 551
 - c. 1063
 - d. cannot be determined
 - n 为偶数, leaf nodes 的数量 = $n/2$; n 为奇数, leaf nodes 的数量 = $(n + 1)/2$
2. In-order traversal of a binary tree can be done iteratively. Given the stack operation sequence as the following: `push(1), push(2), push(3), pop(), push(4), pop(), pop(), push(5), pop(), pop(), push(6), pop()`

Which one of the following statements is TRUE?

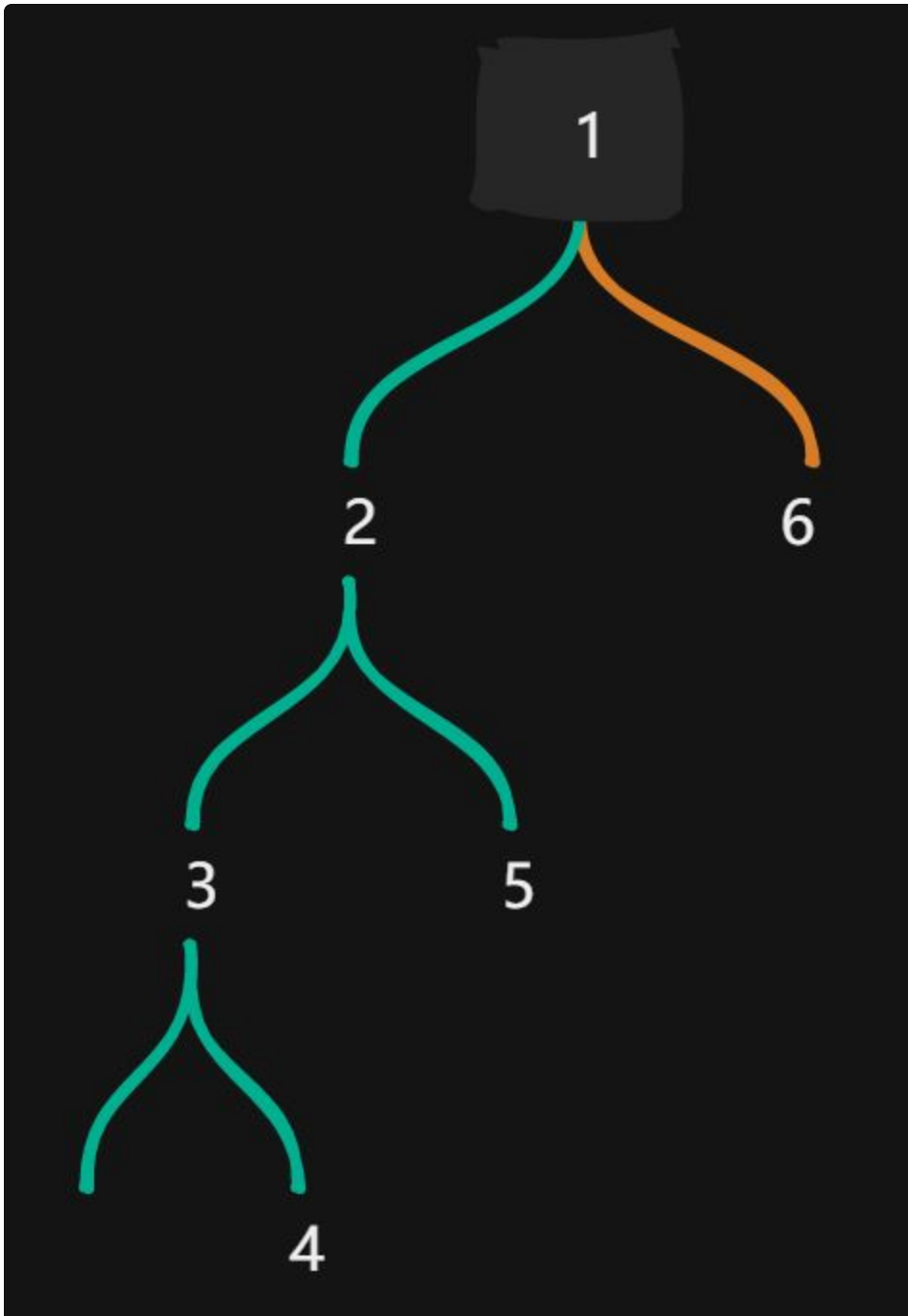
a. 6 is the root

b. 2 is the parent of 4

c. 2 and 6 are siblings

d. None of the above

- 入栈顺序即为先序遍历的顺序，出栈顺序即为中序遍历的顺序
- 入栈 123456 出栈 342516



▼ build a binary tree from its inorder and preorder traversal sequences

```
1  Tree BuildTree( int in[], int pre[], int N )
2  { //in[] stores the inorder traversal sequence
3    //and pre[] stores the preorder traversal sequence
4    //N is the number of nodes in the tree
5      Tree T;
6      int i;
7      if (!N) {
8          return NULL;
9      }
10     T = (Tree)malloc(sizeof(struct Node));
11     T->Data = pre[0];
12     for (i=0; i<N; i++)
13         if (in[i]==T->Data) break;
14     T->Left = BuildTree( in, pre+1, i);
15     T->Right = BuildTree( in+i+1, pre+i+1, N-i-1);
16     return T;
17 }
18
```

▼ build a binary tree from its inorder and postorder traversal sequences

```
1  Tree BuildTree( int in[], int post[], int N )
2  {
3      Tree T;
4      int i;
5      if (!N) {
6          return NULL;
7      }
8      T = (Tree)malloc(sizeof(struct Node));
9      T->Data = post[N-1];
10     for (i=0; i<N; i++)
11         if (in[i]==T->Data) break;
12     T->Left = BuildTree( in, post, i);
13     T->Right = BuildTree( in+i+1, post+i+1, N-i);
14     return T;
15 }
16
```

堆 Heap 优先队列

堆序性 heap order

- the nodes along the path from the root to any node are in sorted order

二叉堆 binary heap 是完全填满的二叉树 complete binary tree

节点数 $2^h \sim 2^h - 1$

高度 h $\lfloor \log N \rfloor$

父亲在 $\lfloor i/2 \rfloor$ 位置上

Insert

上滤 percolate up

在下一个空的位置建一个空穴，向根的方向上走，直到能放入 X

DeleteMin

下滤 percolate down

将删除元素中儿子的较小值放入空穴，空穴下移一层，重复操作，直到最后一个元素放到正确的位置上（要满足完全二叉树）

```
▼ percolate down C++ |  
1
```

Build

用 Insert $O(N)$

IncreaseKey

▼ IncreaseKey in a max-heap

```
1 void IncreaseKey( int P, int D, PriorityQueue H )
2 {
3     int i, key;
4     key = H->Elements[P] + D;
5     for ( i = P; H->Elements[i/2] < key; i/=2 )
6         H->element[i]=H->Element[i/2];
7     H->Elements[i] = key;
8 }
```

DecreaseKey

▼ DecreaseKey in a min-heap

```
1 void DecreaseKey( int P, int D, PriorityQueue H )
2 {
3     int i, key;
4     key = H->Elements[P] - D;
5     for ( i = P; H->Elements[i/2] > key; i/=2 )
6         H->Elements[i] = H->Elements[i/2];
7     H->Elements[i] = key;
8 }
```

关系 Relation

等价关系 equivalence relation

- 自反性 reflexive
- 对称性 symmetric
- 传递性 transitive

eg. 相似、模运算、图像连通性

等价类 equivalence class

对于 集合 S 有 n 个元素，等价类 equivalence class 数量 x ，有 $1 \leq x \leq n$ 个

等价类形成对 S 的一个划分： S 的每个成员恰好出现在一个等价类中

不相交 disjoint

Union/Find 算法（并查算法）

array[] 中每个元素存的是它的根节点的值

 [Introduction to Disjoint Set \(Union-Find Algorithm\) – GeeksforGeeks](#)

Find

Find(i) $O(X)$ 与 X 节点的深度成正比 返回等价类名字（当且仅当两个元素属于相同集合时，Find 返回相同名字

```
Find
C++ |

1 // Finds the representative of the set
2 // that i is an element of
3 #include<bits/stdc++.h>
4 using namespace std;
5
6 int find(int i)
7 {
8     // If i is the parent of itself
9     if (parent[i] == i) {
10         // Then i is the representative of
11         // this set
12         return i;
13     }
14     else {
15         // Else if i is not the parent of
16         // itself, then i is not the
17         // representative of his set. So we
18         // recursively call Find on its parent
19         return find(parent[i]);
20     }
21 }
22 // The code is contributed by Nidhi goel
```

```

1 Find ( ElementType X, DisjSet S )
2 {
3     ElementType root, trail, lead;
4
5     for ( root = X; S[root] > 0; root=S[root] ) ;
6     for ( trail = X; trail != root; trail = lead ) {
7         lead = S[trail] ;
8         S[trail]=root;
9     }
10    return root;
11 }

```

Union

Union(a,b) $\Theta(N)$ 将 a 和 b 两个等价类合并成一个新的等价类

Union(a,b)后新的根是 a

```

1 // Unites the set that includes i
2 // and the set that includes j
3
4 #include <bits/stdc++.h>
5 using namespace std;
6
7 void union(int i, int j) {
8     // Find the representatives
9     // (or the root nodes) for the set
10    // that includes i
11    int irep = this.Find(i),
12    // And do the same for the set
13    // that includes j
14    int jrep = this.Find(j);
15    // Make the parent of i's representative
16    // be j's representative effectively
17    // moving all of i's set into j's set)
18    this.Parent[irep] = jrep;
19 }

```

union-by-size 按大小求并

小的并到大的上

任何节点的深度 depth 不会超过 $\log(N)$

$$\text{height}(T) \leq \lfloor \log_2 N \rfloor + 1$$

union-by-height 按高度求并

任何节点的深度 depth \leq 不会超过 $\log(N)$

路径压缩 path compression

在 Find 操作中执行，从 X 到根的路径上的每一个节点都使它的父节点变成根，即使 S[X] 的值等于 Find 返回的值

路径压缩与按大小求并完全兼容

Find as a Union/Find operation with path compression.

```
1  SetType Find ( ElementType X, DisjSet S )
2  {
3      ElementType root, trail, lead;
4
5      for ( root = X; S[root] > 0; root=S[root] ) ;
6      for ( trail = X; trail != root; trail = lead ) {
7          lead = S[trail] ;
8          S[trail]=root;
9      }
10     return root;
11 }
```

find the K-th largest element in a list A of N elements

C |

```
1 //The function BuildMinHeap(H, K) is to arrange elements H[1] ... H[K] into a min-heap.
2 //Please complete the following program.
3
4 ElementType FindKthLargest ( int A[], int N, int K )
5 { /* it is assumed that K<=N */
6     ElementType *H;
7     int i, next, child;
8
9     H = (ElementType *)malloc((K+1)*sizeof(ElementType));
10    for ( i=1; i<=K; i++ ) H[i] = A[i-1];
11    BuildMinHeap(H, K);
12
13    for ( next=K; next<N; next++ ) {
14        H[0] = A[next];
15        if ( H[0] > H[1] ) {
16            for ( i=1; i*2<=K; i=child ) {
17                child = i*2;
18                if ( child!=K && H[child+1]<H[child] ) child++;
19                if ( H[0] > H[child] )
20                    H[i] = H[child];
21                else break;
22            }
23            H[i] = H[0];
24        }
25    }
26    return H[1];
27 }
```

Q

1. The array representation of a disjoint set is given by { 4, 6, 5, 2, -3, -4, 3 }. If the elements are numbered from 1 to 7, the resulting array after invoking `Union(Find(7), Find(1))` with **union-by-size** and **path-compression** is:
 - a. { 4, 6, 5, 2, 6, -7, 3 }
 - b. { 4, 6, 5, 2, -7, 5, 3 }
 - c. { 6, 6, 5, 6, -7, 5, 5 }
 - d. { 6, 6, 5, 6, 6, -7, 5 }

图 Graph

introduction

In a directed graph, the sum of the in-degrees and out-degrees of all the vertices is twice the total number of edges.

求和 $\text{degree} = 2 * E$

有最多的边的数量有向图是 $n(n-1)$,无向图是 $n(n-1) / 2$

connected

There are n vertices.

The **minimum** number of edges in a **connected graph** is $(n-1)$. The **maximum** for this question is $(n-1)(n-2)/2 + 1$. This is because $(n-1)$ edges can be connected by maximum $(n-1)(n-2)/2$ edges, and 1 edge to connect to the lonely vertex.

割点 articulation point/cut vertex

*A vertex v is an **articulation point** (also called **cut vertex**) if removing v increases the number of connected components.*

拓扑排序 topological

对有向无圈图的顶点的一种排序

```
▼ topological C++ |
1
```

Dijkstra algorithm

找到从一个给定点到所有点之间的最短路径

时间复杂度: $O(|E| + |V|^2)$

Maximum Network Flow 最大网络流

给定一个表示流网络的图，其中每条边都有容量。还给定图中的两个顶点源“s”和汇“t”，在以下约束下找到从 s 到 t 的最大可能流量：

1. 边缘上的流量不会超过边缘的给定容量。
2. 对于除 s 和 t 之外的每个顶点，传入流等于传出流。

Dinic

 [Dinic's algorithm for Maximum Flow – GeeksforGeeks](#)

Dinic 的算法使用以下概念：

1. 将残差图 G 初始化为给定图。
2. 对 G 进行 BFS 来构造一个级别图（或将级别分配给顶点），并检查是否可以有更多流。
 - a. 如果无法获得更多流量，则返回
 - b. 使用级别图在 G 中发送多个流，直到达到 **阻塞流**。

这里**使用级别图**意味着，在每个流中，从 s 到 t，路径节点的级别应该是 0、1、2...（按顺序）。

时间复杂度 $O(EV^2)$

```
1 // C++ implementation of Dinic's Algorithm
2 #include <bits/stdc++.h>
3 using namespace std;
4
5 // A structure to represent a edge between
6 // two vertex
7 struct Edge {
8     int v; // Vertex v (or "to" vertex)
9           // of a directed edge u-v. "From"
10           // vertex u can be obtained using
11           // index in adjacent array.
12
13     int flow; // flow of data in edge
14
15     int C; // capacity
16
17     int rev; // To store index of reverse
18              // edge in adjacency list so that
19              // we can quickly find it.
20 };
21
22 // Residual Graph
23 class Graph {
24     int V; // number of vertex
25     int* level; // stores level of a node
26     vector<Edge>* adj;
27
28 public:
29     Graph(int V)
30     {
31         adj = new vector<Edge>[V];
32         this->V = V;
33         level = new int[V];
34     }
35
36     // add edge to the graph
37     void addEdge(int u, int v, int C)
38     {
39         // Forward edge : 0 flow and C capacity
40         Edge a{ v, 0, C, (int)adj[v].size() };
41
42         // Back edge : 0 flow and 0 capacity
43         Edge b{ u, 0, 0, (int)adj[u].size() };
44
45         adj[u].push_back(a);
```

```

46         adj[v].push_back(b); // reverse edge
47     }
48 }
49
50 bool BFS(int s, int t);
51 int dfsFlow(int s, int t, int flow, int ptr[]);

```

Ford-

MST 最小生成树

最小生成树(MST) 或带权连通无向图的最小权重生成树是权重小于或等于所有其他生成树权重的生成树。

Necessary-And-Sufficient-Condition-for-Unique-MST

Necessary-And-Sufficient-Condition-for-Unique-MST

Let G be a connected weighted graph and T a minimum spanning tree of G . Show that T is a unique minimum spanning tree **if and only if** the weight of each edge e of G that is not in T exceeds the weight of every other edge on the cycle in $T+e$.

1. Existence of Minimum Spanning Tree (MST):

- If the graph is connected, a minimum spanning tree always exists.
- For an undirected connected graph, the minimum spanning tree is a subset of edges that forms a tree connecting all vertices with the minimum possible total edge weight.

2. Conditions for Existence:

- In a connected graph, a minimum spanning tree is guaranteed.
- If the graph is not connected, there won't be a minimum spanning tree [4].

3. Algorithms:

- Kruskal's and Prim's algorithms are commonly used to find the minimum spanning tree of a connected graph.

Prim 算法

从已经在 MST 里面的顶点中，连接未在 MST 中的顶点，使其边权最小

Step 1: Determine an arbitrary vertex as the starting vertex of the MST.

Step 2: Follow steps 3 to 5 till there are vertices that are not included in the MST (known as fringe vertex).

Step 3: Find edges connecting any tree vertex with the fringe vertices.

Step 4: Find the minimum among these edges.

Step 5: Add the chosen edge to the MST if it does not form any cycle.

Step 6: Return the MST and exit

```
1 // A C program for Prim's Minimum
2 // Spanning Tree (MST) algorithm. The program is
3 // for adjacency matrix representation of the graph
4
5 #include <limits.h>
6 #include <stdbool.h>
7 #include <stdio.h>
8
9 // Number of vertices in the graph
10 #define V 5
11
12 // A utility function to find the vertex with
13 // minimum key value, from the set of vertices
14 // not yet included in MST
15 int minKey(int key[], bool mstSet[])
16 {
17     // Initialize min value
18     int min = INT_MAX, min_index;
19
20     for (int v = 0; v < V; v++)
21         if (mstSet[v] == false && key[v] < min)
22             min = key[v], min_index = v;
23
24     return min_index;
25 }
26
27 // A utility function to print the
28 // constructed MST stored in parent[]
29 int printMST(int parent[], int graph[V][V])
30 {
31     printf("Edge \tWeight\n");
32     for (int i = 1; i < V; i++)
33         printf("%d - %d \t%d \n", parent[i], i,
34             graph[i][parent[i]]);
35 }
36
37 // Function to construct and print MST for
38 // a graph represented using adjacency
39 // matrix representation
40 void primMST(int graph[V][V])
41 {
42     // Array to store constructed MST
43     int parent[V];
44     // Key values used to pick minimum weight edge in cut
45     int key[V];
```

```

46 // To represent set of vertices included in MST
47 bool mstSet[V];
48
49 // Initialize all keys as INFINITE
50 for (int i = 0; i < V; i++)
51     key[i] = INT_MAX, mstSet[i] = false;
52
53 // Always include first 1st vertex in MST.
54 // Make key 0 so that this vertex is picked as first
55 // vertex.
56 key[0] = 0;
57
58 // First node is always root of MST
59 parent[0] = -1;
60
61 // The MST will have V vertices
62 for (int count = 0; count < V - 1; count++) {
63
64     // Pick the minimum key vertex from the
65     // set of vertices not yet included in MST
66     int u = minKey(key, mstSet);
67
68     // Add the picked vertex to the MST Set
69     mstSet[u] = true;
70
71     // Update key value and parent index of
72     // the adjacent vertices of the picked vertex.
73     // Consider only those vertices which are not
74     // yet included in MST
75     for (int v = 0; v < V; v++)
76
77         // graph[u][v] is non zero only for adjacent
78         // vertices of m mstSet[v] is false for vertices
79         // not yet included in MST Update the key only
80         // if graph[u][v] is smaller than key[v]
81         if (graph[u][v] && mstSet[v] == false
82             && graph[u][v] < key[v])
83             parent[v] = u, key[v] = graph[u][v];
84     }
85
86     // print the constructed MST
87     printMST(parent, graph);
88 }
89
90 // Driver's code
91 int main()
92 {
93     int graph[V][V] = { { 0, 2, 0, 6, 0 },

```

```

94         { 2, 0, 3, 8, 5 },
95     { 0, 3, 0, 0, 7 },
96     { 6, 8, 0, 0, 9 },
97     { 0, 5, 7, 9, 0 } };
98
99
100     // Print the solution
101     primMST(graph);
102
103     return 0;
104 }

```

时间复杂度： $O(V^2)$ ，如果输入图使用邻接表来表示，那么借助二叉堆，Prim算法的时间复杂度可以降低到 $O(E * \log V)$ 。在这个实现中，我们总是考虑生成树从图的根开始

空间： $O(V)$

Kruskal 算法

 [Kruskal's Minimum Spanning Tree \(MST\) Algorithm – GeeksforGeeks](#)

在克鲁斯卡尔算法中，按升序对给定图的所有边进行排序。如果新添加的边不形成环，则继续在MST中添加新的边和节点。它首先选择**最小加权边**，最后选择最大加权边。因此我们可以说它在每一步中都做出局部最优选择以找到最优解。因此这是一个**贪婪算法**。

1. 按权重非降序对所有边进行排序。
2. 选择最小的边。检查是否与目前形成的生成树形成环。如果未形成循环，则包括该边。否则，丢弃它。
3. 重复步骤2，直到生成树中有 $(V-1)$ 条边。

```
1 // C++ program for the above approach
2
3 #include <bits/stdc++.h>
4 using namespace std;
5
6 // DSU data structure
7 // path compression + rank by union
8 class DSU {
9     int* parent;
10    int* rank;
11
12 public:
13     DSU(int n)
14     {
15         parent = new int[n];
16         rank = new int[n];
17
18         for (int i = 0; i < n; i++) {
19             parent[i] = -1;
20             rank[i] = 1;
21         }
22     }
23
24     // Find function
25     int find(int i)
26     {
27         if (parent[i] == -1)
28             return i;
29
30         return parent[i] = find(parent[i]);
31     }
32
33     // Union function
34     void unite(int x, int y)
35     {
36         int s1 = find(x);
37         int s2 = find(y);
38
39         if (s1 != s2) {
40             if (rank[s1] < rank[s2]) {
41                 parent[s1] = s2;
42             }
43             else if (rank[s1] > rank[s2]) {
44                 parent[s2] = s1;
45             }
46         }
47     }
48 }
```



```

46         else {
47             parent[s2] = s1;
48             rank[s1] += 1;
49         }
50     }
51 }
52 };
53
54 class Graph {
55     vector<vector<int>> > edgelist;
56     int V;
57 }

```

时间复杂度: $O(E * \log E)$ 或 $O(E * \log V)$

- 边排序需要 $O(E * \log E)$ 时间。
- 排序后, 我们迭代所有边并应用查找并集算法。查找和并集操作最多需要 $O(\log V)$ 时间。
- 所以总体复杂度是 $O(E * \log E + E * \log V)$ 时间。
- E的值最多可以是 $O(V^2)$, 因此 $O(\log V)$ 和 $O(\log E)$ 是相同的。因此, 总体时间复杂度为 $O(E * \log E)$ 或 $O(E * \log V)$

DFS 深度优先搜索

1. 最初堆栈和访问数组都是空的。
2. 访问某一节点, 将其未访问过的相邻节点放入栈中。
3. 访问栈顶并将其从栈中弹出, 并将其所有未访问的相邻节点放入栈中。
4. 重复步骤 3 直到栈变空, DFS 结束

Find articulation point 找割点



Strongly Connected Components 强连通组件

Tarjan 算法 找割点? 还是强连通组件

DFS+栈实现

 [哔哩哔哩视频参考](#)

```

1  #include<malloc.h>
2  typedef struct VNode *PtrToVNode;
3  struct VNode {
4      Vertex Vert;
5      PtrToVNode Next;
6  };
7  typedef struct GNode *Graph;
8  struct GNode {
9      int NumOfVertices;
10     int NumOfEdges;
11     PtrToVNode *Array;
12 };
13
14 int count=0;
15 #define min(a,b) (((a)<(b))?(a):(b))
16 void find(Graph G, Vertex v, int num[],int low[], PtrToVNode stack, int visited[]);
17 void push(int x, PtrToVNode s);
18 int top(PtrToVNode s);
19 void pop(PtrToVNode s);
20 void StronglyConnectedComponents( Graph G, void (*visit)(Vertex V) ){
21     int low[MaxVertices]={0};
22     int visited[MaxVertices]={0};
23     int num[MaxVertices]={0};
24     PtrToVNode stack;
25     stack=malloc(sizeof(struct VNode));
26     for (int i = 0; i < G->NumOfVertices; i++) {
27         if (num[i] == 0) {
28             find(G, i,num,low,stack,visited);
29         }
30     }
31 }
32 void find(Graph G, Vertex v, int num[],int low[], PtrToVNode stack, int visited[]){
33     visited[v]=1;
34     num[v] = low[v] = count++;
35     push(v,stack);
36     PtrToVNode w;
37     for(w=G->Array[v];w!=NULL;w=w->Next){

```

时间复杂度: $O(E + V)$ 邻接表; $O(V^2)$ 邻接矩阵

Q

1. If an undirected graph $G = (V, E)$ contains 10 vertices. Then to guarantee that G is connected in any cases, there has to be at least __ edges. ?
 - a. 45
 - b. 37
 - c. 36
 - d. 9
2. Which of the following statements is TRUE about **topological sorting**?
 - a. If a graph has a topological sequence, then its adjacency matrix must be triangular.
 - b. If the adjacency matrix is triangular, then the corresponding directed graph must have a unique topological sequence.
 - c. In a DAG, if for any pair of distinct vertices V_i and V_j , there is a path either from V_i to V_j or from V_j to V_i , then the DAG must have a unique topological sequence.
 - d. If V_i precedes V_j in a topological sequence, then there must be a path from V_i to V_j .
3. Let P be the shortest path from S to T . If the weight of every edge in the graph is **incremented by 2**, P will still be the **shortest path** from S to T . F

Let G be the complete graph on three vertices A , B , and C . Let edges AB and BC have weight 2, and edge AC have weight 4.5.

Then the shortest path from A to C is via B . But if you increase the weights to 3, 3, and 5.5, the shortest path is the edge AC .

4. The minimum spanning tree of any connected weighted graph: 任意连通加权图的最小生成树

C.may not be unique

5. Apply DFS to a directed acyclic graph (有向无圈图), and output the vertex before the end of each recursion. The output sequence will be:

C.reversely topologically sorted

6. Graph G is an undirected complete graph of 20 nodes. Is there an Euler circuit in G ?
If not, in order to have an **Euler circuit**, what is the minimum number of edges which should be removed from G ?

Each Node has exactly 19 degrees

- Euler Circuit (Strong Form) requires every node to be even degrees
- Euler Tour (Weak Form) requires 0 or 2 odd degrees

Remove 1 edge, every 2 nodes will lose 1 degrees, so we lose 10 edges

Questions

homework

1. For a sequentially stored linear list of length N , the time complexities for deleting the first element and inserting the last element are $\Theta(1)$ and $\Theta(N)$, respectively.

F $O(N)$ and $O(1)$

- 顺序存储的线性表支持随机存取，所以查询的时间是常数时间，但插入需要把后面每一个元素的位置都进行调整，所以是线性时间。插入最后一个时间为 $O(1)$ 。

2. 循环队列满时 $\text{rear} == \text{front} - 1$. enqueue时 rear 增加, dequeue front 增加.

3. To insert **s** after **p** in a **doubly linked circular list**, we must do:

- a. $\text{p} \rightarrow \text{next} = \text{s}$; $\text{s} \rightarrow \text{prior} = \text{p}$; $\text{p} \rightarrow \text{next} \rightarrow \text{prior} = \text{s}$; $\text{s} \rightarrow \text{next} = \text{p} \rightarrow \text{next}$;
- b. $\text{p} \rightarrow \text{next} \rightarrow \text{prior} = \text{s}$; $\text{p} \rightarrow \text{next} = \text{s}$; $\text{s} \rightarrow \text{prior} = \text{p}$; $\text{s} \rightarrow \text{next} = \text{p} \rightarrow \text{next}$;
- c. $\text{s} \rightarrow \text{prior} = \text{p}$; $\text{s} \rightarrow \text{next} = \text{p} \rightarrow \text{next}$; $\text{p} \rightarrow \text{next} = \text{s}$; $\text{p} \rightarrow \text{next} \rightarrow \text{prior} = \text{s}$;
- d. $\text{s} \rightarrow \text{prior} = \text{p}$; $\text{s} \rightarrow \text{next} = \text{p} \rightarrow \text{next}$; $\text{p} \rightarrow \text{next} \rightarrow \text{prior} = \text{s}$; $\text{p} \rightarrow \text{next} = \text{s}$;

4. It is always possible to represent a tree by a one-dimensional integer array. T

- It is always possible to represent a tree by a one-dimensional integer array using various techniques such as breadth-first or depth-first traversal.

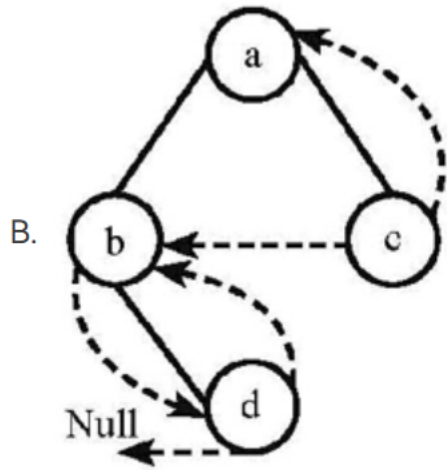
5. If a general tree **T** is converted into a binary tree **BT**, then which of the following **BT** traversals gives the same sequence as that of the post-order traversal of **T**?

- a. Pre-order traversal
- b. In-order traversal
- c. Post-order traversal
- d. Level-order traversal

T的preorder = BT的preorder

T的postorder = BT的inorder

6. Among the following **threaded binary trees** (the threads are represented by dotted curves), which one is the **postorder** threaded tree?



线索二叉树中，左线索为上一个结点，右线索为下一个结点

后序：左右根

中序：左根右

前序：根左右

7. Suppose that an array of size 6 is used to store a circular queue, and the values of front and rear are 0 and 4, respectively. Now after 2 dequeues and 2 enqueues, what will the values of **front** and **rear** be?

- a. 2 and 0
- b. 2 and 2
- c. 2 and 4
- d. 2 and 6

- 头是 0 而不是 6

8. Suppose that an array of size **m** is used to store a circular queue. If the head pointer **front** and the current size variable **size** are used to represent the range of the queue instead of **front** and **rear**, then the maximum capacity of this queue can be: (5分)

- a. $m-1$
- b. **m**
- c. $m+1$
- d. cannot be determined

- 就是数组的大小

Midterm

1. The time complexity of Selection Sort will be the same no matter we store the elements in an array or a linked list. **T**
2. If N numbers are stored in a singly linked list in increasing order, then the average time complexity for binary search is $O(\log N)$. **F** 链表是不能使用折半查找的
3. The time complexity of Selection Sort will be the same no matter we store the elements in an array or a linked list. **T**
4. If a stack is used to convert the infix expression $a+b*c+(d*e+f)*g$ into a postfix expression, what will be in the stack (listing from the bottom up) when f is read?
 - a. $+(+$
 - b. $+(* +$
 - c. abcde
 - d. $++(+$


中缀表达式转化为后缀表达式，转化的算法如下：

- 初始化一个栈
 - 逐个读取元素（数字或者操作符）
 - 如果遇到数字，直接输出
 - 如果遇到操作符（不考虑括号），如果其优先级大于栈顶元素，就将栈顶弹出，并重复此步骤，否则将该操作符压入栈中（栈为空的时候也直接压栈即可）
 - 如果遇到左括号"(", 直接将其压入栈中，如果遇到右括号")", 循环弹出栈顶元素，直到左括号为止（左括号也需要弹出，右括号不需要压栈），并且输出所有被弹栈顶元素（左括号除外）
5. Suppose that a polynomial（多项式） is represented by a linked list storing its non-zero terms. Given two polynomials with N_1 and N_2 non-zero terms, and the highest exponents being M_1 and M_2 , respectively. Then the time complexity for adding them up is:
 - a. $O(N_1 \times N_2)$
 - b. $O(N_1 + N_2)$
 - c. $O(M_1 \times M_2)$
 - d. $O(M_1 + M_2)$

6.

Pta code

True or Flase

 [算法竞赛基础训练题_判断题_it is always possible to represent a tree by a one_白术_竹苓的博客-CSDN博客](#)

function

```
Reverse Linked List

1 List Reverse( List L ){
2     List head,node,temp;
3     node=L->Next;
4     while(node){
5         temp=node->Next;
6         node->Next=head;
7         head=node;
8         node=temp;
9     }
10    L->Next=head;
11    return L;
12 }
13
14
15 typedef struct Node *PtrToNode;
16 typedef PtrToNode List;
17 typedef PtrToNode Position;
18 struct Node {
19     ElementType Element;
20     Position Next;
21 };
```

The function `Reverse` is supposed to return the reverse linked list of `L`, with a dummy header.

```
1 Polynomial Add( Polynomial a, Polynomial b ){
2     Polynomial head,temp,node;
3     if(a==NULL) return b;
4     if(b==NULL) return a;
5     node=(Polynomial)malloc(sizeof(Polynomial));
6     head=node;
7     a=a->Next;
8     b=b->Next;
9     while(b && a){
10         temp=(Polynomial)malloc(sizeof(Polynomial));
11
12         if(b->Exponent > a->Exponent){
13             temp->Coefficient=b->Coefficient;
14             temp->Exponent=b->Exponent;
15             b=b->Next;
16         }else if(b->Exponent < a->Exponent){
17             temp->Coefficient=a->Coefficient;
18             temp->Exponent=a->Exponent;
19             a=a->Next;
20         }else{
21             temp->Coefficient=a->Coefficient + b->Coefficient;
22             temp->Exponent=a->Exponent;
23             a=a->Next;
24             b=b->Next;
25             if(temp->Coefficient==0) continue;
26         }
27         temp->Next=NULL;
28         node->Next=temp;
29         node=temp;
30     }
31     if(a) node->Next=a;
32     if(b) node->Next=b;
33     return head;
34 }
35
36 typedef struct Node *PtrToNode;
37 struct Node {
38     int Coefficient;
39     int Exponent;
40     PtrToNode Next;
41 };
42 typedef PtrToNode Polynomial;
43 /* Nodes are sorted in decreasing order of exponents.*/
```


▼ No Less Than X in BST

C

```
1 void Print_NLT( Tree T, int X ){
2     if(T == NULL) return; //如果树为空, 返回null
3
4     //直接进行树的遍历, 因为是从大到小输出不小于X的数
5     //所以先遍历右子树, 大于X的直接输出, 再遍历左子树。
6     Print_NLT(T->Right, X);
7     if(T->Element >= X){
8         printf("%d ", T->Element);
9     }
10    Print_NLT(T->Left, X);
11
12 }
```

▼ Is tree Isomorphic 同构

C

```
1 int Isomorphic( Tree T1, Tree T2 ){
2     if(T1==NULL && T2==NULL) return 1;
3     if((T1==NULL&&T2!=NULL) || (T1!=NULL&&T2==NULL)) return 0;
4     else if(T1->Element != T2->Element) return 0;
5     return (Isomorphic(T1->Left, T2->Left) && Isomorphic(T1->Right, T2->Right
6 )) || (Isomorphic(T1->Right, T2->Left) && Isomorphic(T1->Left, T2->Right));
7 }
```

```
1 void PercolateUp( int p, PriorityQueue H )
2 {
3     while (H->Elements[p] < H->Elements[p/2] && p>1)
4     {
5         int temp;
6         temp=H->Elements[p];
7         H->Elements[p]=H->Elements[p/2];
8         H->Elements[p/2]=temp;
9         p /=2;
10    }
11 }
12
13 void PercolateDown( int p, PriorityQueue H )
14 {
15     while (H->Elements[p] > H->Elements[p*2] && 2*p <= H->Size )
16     {
17         int temp;
18         temp=H->Elements[p];
19         H->Elements[p]=H->Elements[p*2];
20         H->Elements[p*2]=temp;
21         p *=2;
22     }
23 }
```