# 数据结构与算法

## **\*\*\*\*\*算法复习注意事项：**

* 多复习。多重复
* 注意解题思路。（尤其注意关键点。比如49题的关键是字符串排序+map的值为字符串数组。）
* 注意讲自己不熟悉的内容进行整理和总结。.
* 对每道题都写自己的理解，后边可以写一本类似于labuladuo算法一样的题解的书。。

一举多得。

* 。

## 基本概念:

二叉树：除了叶子节点外，每个节点只有两个分支，左子树和右子树，每个节点的最大度数为2.

满二叉树：除了叶结点外每一个结点都有左右子叶且叶结点都处在最底层的二叉树,

完全二叉树：只有最下面的两层结点度小于2，并且最下面一层的结点都集中在该层最左边的若干位置的二叉树。

## 算法。。

### LFU:最不经常使用算法

### KMP算法

* 目的是为了减少无畏的字符比较。。
* 把专注点放在已经匹配的前缀

为了两个子串，没必要每次都去遍历，可以实现缓存到一个集合中。。用的时候再去集合里取。。

成为next数组。

### 建堆的复杂度和建堆的过程

### 哈夫曼编码

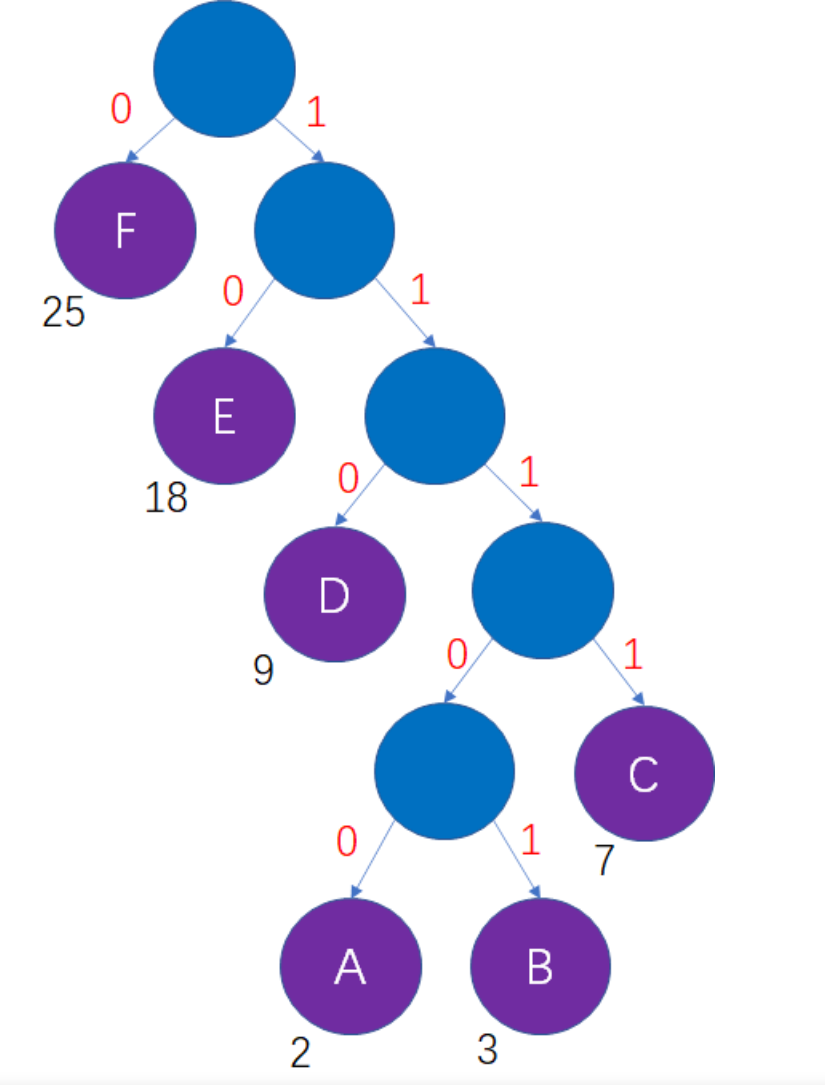
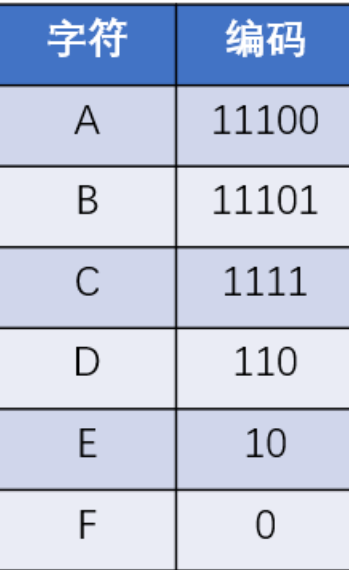
* 高效的编码方式、对信息进行压缩。。

计算机所有的信息都是二进制编码的。。

定长编码和不定长编码。。

* 利用树的性质
* 举例

1. B，C，D，E，F这6个字符，他们出现的次数依次是2次，3次，7次，9次，18次，25次，



### 100万的数据里找最大的前100个数

* 方法1：快排
* 方法2：将100万和数据划分为100组，每组找最大的数据。进行比较。得到最大的一组的最大的数，再继续查找。

## 算法必备内容

## 排序算法

<https://blog.csdn.net/kuaizi_sophia/article/details/87954222>

### **冒泡排序（稳定）**

#### 概述

从头开始，每次比较两元素，若大者在前，则交换两元素，直至数组末尾，此时最大元素为数组最后的元素；

#### 稳定性

因为只涉及相邻两个元素的交换，当相邻两元素相等时就不进行交换，

#### 实现：

// 冒泡排序  
void bubbleSort(vector<int>& array) {  
   for (size\_t i = 0; i < array.size(); i++) {  
       // 当前轮是否发生过交换事件标志位，若未发生交换，则表明列表已有序。  
       bool isExchanged = false;  
       for (size\_t j = 0; j < array.size() - i - 1; j++) {  
           if (array[j] > array[j + 1]) {  
               int temp = array[j];  
               array[j] = array[j + 1];  
               array[j + 1] = temp;  
               isExchanged = true;  
           }  
       }  
       if (!isExchanged){  
           break;  
       }  
   }  
}

### **选择排序（不稳定）**

#### 概述

选择（最小值）  
​  
遍历n轮  
以第i轮为例，每轮都从第i个元素依次向后找到比第i个元素小的交换到第i个位置上

#### 稳定性

举个例子，序列5 8 5 2 9， 我们知道第一遍选择第1个元素5会和2交换，那么原序列中2个5的相对前后顺序就被破坏了，所以选择排序不是一个稳定的排序算法。  
​  
因为有非相邻两个元素的swap 所以不能保证稳定

#### 实现

void selectSort(vector<int>& array){  
   for (size\_t i = 0; i < array.size(); i++){  
       size\_t minIndex = i;  
       for (size\_t j = i + 1; j < array.size(); j++){  
           if (array[minIndex] > array[j]){  
               minIndex = j;  
           }  
       }  
       if (minIndex != i){  
           swap(array[i], array[minIndex]);  
       }  
   }  
}

### **插入排序（稳定）**

#### 概述

将第一个元素看作有序序列，后续元素当作无序序列，依次将无序序列元素插入有序序列当中

#### 应用场景

适用于小规模数据或者基本有序时。。

可以和快排结合使用。

#### 稳定性

只涉及相邻元素的比较   没有非相邻元素的swap   所以是稳定的  
两个while循环  
​  
是比较类似于动态规划的

#### 实现

Void insertionSort(vector<int>& array){  
   // i 代表无序序列首元素（无序序列前为有序序列）

int i;

int j;

for (i = 1;i < arr.size() ; i++)

{

tmp = arr[i];

for (j = i - 1;j >= 0 && arr[j] > tmp;j--)

{

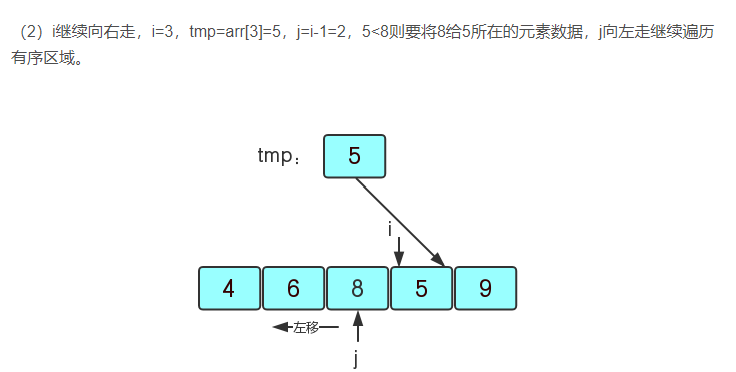
arr[j + 1] = arr[j];

}

arr[j + 1] = tmp;

}  
}

对于插入排序通过下图来理解。。



### **希尔排序（非稳定）（缩小增量排序）**

#### 概述

是对插入排序的改进  
提高对大规模或者无序数据的效率。  
依次使得间隔为h的元素有序、  
h一直两倍减小

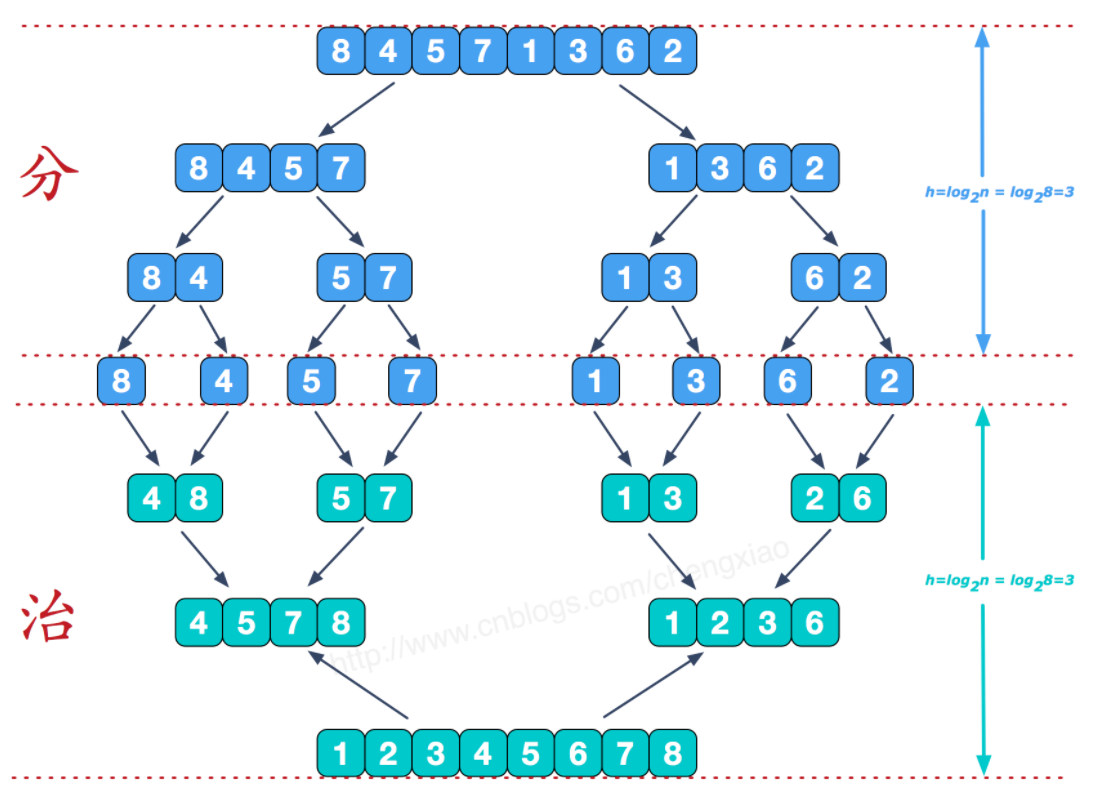
#### 实现

void shellSortCore(vector<int>& nums, int gap, int i) {  
     int inserted = nums[i];  
     int j;  
     // 插入的时候按组进行插入  
     for (j = i - gap; j >= 0 && inserted < nums[j]; j -= gap) {  
     nums[j + gap] = nums[j];  
     }  
     nums[j + gap] = inserted;   
 }  
   
void shellSort(vector<int>& nums) {  
     int len = nums.size();  
     //进行分组，最开始的时候，gap为数组长度一半  
     for (int gap = len / 2; gap > 0; gap /= 2) {  
     //对各个分组进行插入分组  
             for (int i = gap; i < len; ++i) {  
             //将nums[i]插入到所在分组正确的位置上  
             shellSortCore(nums,gap,i);  
           }  
       }  
     for (auto a : nums) {  
       cout << a << "";  
     }  
}

### **归并排序（稳定）（分治思想）**

#### 概述

​  
将列表从正中间分为两个子列表；  
按照第一步，递归拆分每个子列表，直至子列表最大长度为1；  
按照拆分层级，依次按大小合并各子列表，直至全部合并完成。



#### 迭代实现

// 归并排序  
void mergeSort(vector<int>& data) {  
     int len = data.size();  
     vector<int> dataTemp(len, 0);  
     for (int seg = 1; seg < len; seg += seg) {  
         for (int start = 0; start < len; start += seg + seg) {  
         int low = start, mid = min(start + seg, len), high = min(start + seg +seg,len);  
         int index = low, start1 = low, end1 = mid, start2 = mid, end2 = high;  
                 while (start1 < end1 && start2 < end2) {  
                 dataTemp[index++] = data[start1] < data[start2] ? data[start1++] : data[start2++];  
               }  
                 while (start1 < end1) {  
                 dataTemp[index++] = data[start1++];  
                 }  
               while (start2 < end2) {  
                 dataTemp[index++] = data[start2++];  
               }  
         }  
         swap(data, dataTemp);  
     }  
       for (auto a : data)  
       cout << a << " ";   
}

#### 分治-递归实现

void mergeSortCore(vector<int>& data, vector<int>& dataTemp, int low, int high) {  
       if (low >= high) return;  
       int len = high - low, mid = low + len / 2;  
       int start1 = low, end1 = mid, start2 = mid + 1, end2 = high;  
       mergeSortCore(data, dataTemp, start1, end1);  
       mergeSortCore(data, dataTemp, start2, end2);  
       int index = low;  
       while (start1 <= end1 && start2 <= end2) {  
       dataTemp[index++] = data[start1] < data[start2] ? data[start1++] :   
       data[start2++];  
           }  
         while (start1 <= end1) {  
         dataTemp[index++] = data[start1++];  
           }  
           while (start2 <= end2) {  
         dataTemp[index++] = data[start2++];  
             }  
           for (index = low; index <= high; ++index) {  
         data[index] = dataTemp[index];  
           }  
}  
​  
void mergeSort(vector<int>& data) {  
     int len = data.size();  
     vector<int> dataTemp(len, 0);  
       mergeSortCore(data, dataTemp, 0, len - 1);  
}

### **快速排序（不稳定--分治思想）**

#### \*\*思路概述

从列表中选出一个元素，作为“基准”pivot，基准一般随机选择，或采用最左端、最右端和中间位置3元素的中值；  
基准+双指针  
以选择最左侧为基准为例  
先右侧-- 直到找到第一个小于基准的值  
再左侧++ 直到找到第一个大于基准的值。  
​  
当数据量很小时，快排效果不如插入排序。。算法不稳定且有递归开销。

#### 稳定性

会发生两个不相邻的元素的swap  
所以是不稳定的

#### 实现

// 快速排序（递归）  
// 选则最左端、最右端和中间位置3元素的中值作为基准值，并将3元素排序，返回基准值  
void quickSort(vector<int>& a, int low, int high) {  
   if (low >= high)   return;   // 结束标志  
   int first = low; // 低位下标  
   int last = high; // 高位下标  
   int key = a[first]; // 设第一个为基准  
   while (first < last) {  
           // 从后往前走，将比第一个小的移到前面  
         while (first < last && a[last] > key)   last--;  
         if (first < last)   a[first++] = a[last];  
         //从前往后走， 将比第一个大的移到后面  
           while (first < last && a[first] <= key) first++;  
         if (first < last)   a[last--] = a[first];  
     }  
     a[first] = key;  
     quickSort(a, low, first - 1);   // 前半递归  
     quickSort(a, first + 1, high);   // 后半递归  
}

quickSort(A, 0,A.size()-1);  
 for (auto a : A) {  
 cout << a << endl;  
 }

#### 优化

解决的问题：若待排序序列时部分有序的，固定选取基准点会使得快排的效率地下、

优化1：随机取基准点、rand（）（应用基本有序的情况）

优化2:小数据用插排来优化。（应用小范围数据）

优化3：聚集优化（应用大量重复数据）

### **堆排序（不稳定）**

#### 概述（基本过程）

取出堆顶元素（最大元素），作为有序数数组末尾元素，并对二叉树进行调整使其满足堆的特性；  
重复上一步骤，依次取出堆顶元素，并插入到有序数组中，上一插入元素之前的位置，直到堆空为止；  
​  
基本过程：（以大根堆的建立为例）  
1.将N个元素的序列构建成一个大根堆  
2.将堆顶的元素放到序列末尾  
3.将前n-1个元素重新构建大顶堆

#### 1--口述堆排序

物理上是线性的，逻辑上是树状的。  
假设一个节点的下标索引是i，其子节点的下标分别是2\*i+1和2\*i+2。  
基于此性质，从倒数第二层依次往上进行堆的有序化，使得满足根节点大于其左右子节点。。  
​  
​

#### 实现

//以建立大根堆为例//以下算法的实现分为两部  
1.先建立大根堆  
2.在大根堆的基础上，依次将堆的首元素放到队尾、然后对剩下的元素进行排序。  
3.以下的三个函数中都是由heapify()函数的。。  
//  
​  
// 调整堆，根元素沿树向下移动，直至其合适位置，first和last分别为堆顶和堆底在数组array中的索引  
void heapify(vector<int>& nums, int n, int i)//对有一定顺序的堆，  
       //当前第i个结点取根左右的最大值（这个操作称heapfiy） {  
       int l = i \* 2 + 1, r = i \* 2 + 2;  
       int max = i;  
       if (l<n && nums[l]>nums[max])  
           max = l;  
       if (r<n && nums[r]>nums[max])  
           max = r;  
       if (max != i) {  
             swap(nums[max], nums[i]);  
             heapify(nums, n, max); //如果发生了交换 对交换了的位置递归向下判断-  
       }  
}  
​  
//建立大根堆，从树的倒数第二层最右侧结点开始，往回遍历  
void heapify\_build(vector<int>& nums, int n)  
{     //对每个结点进行heapify操作，然后向上走  
     int temp = (n - 2) / 2;  
     for (int i = temp; i >= 0; i--)  
         heapify(nums, n, i);  
}  
            
            
void heapify\_sort(vector<int>& nums, int n)  
//建立大根堆之后，每次交换最后一个结点和根节点（最大值），  
//对交换后的根节点继续进行heapify（此时堆的最后一位是最大值，因此不用管他，Heapify的范围变成由n变为n-1） {  
     heapify\_build(nums, n);  
     for (int i = 0; i < n; i++){  
         swap(nums.front(), nums[n - i - 1]);  
         heapify(nums, n - i - 1, 0);  
     }  
}  
​  
​

计数排序（外部+稳定排序）

桶排序（外部排序）

​

基树排序（稳定排序）

#### 应用：

**广泛应用于处理大数据、最典型的应用就是百万数据找前100个数的操作。**

**可以通过建立100个大根堆来实现。。**

### **排序算法复杂度分析**



### **2--排序算法的适用场景：稳定性：**

* **数据小时复杂度较好的情况：冒泡、直接选择、**
* **数据大时较好的情况：堆排序、归并排序、**
* **有序时：直接插入复杂度好。快排反而不好。**

**快排在基本又序的情况下会变成冒泡排序。**

* **稳定性：两个相等的树在排序前后的相对位置时相同的。**

**希尔、快排、堆排、归并、采用了跳跃式的比较。降低了比较次数。**

**但是牺牲了稳定性。**

## 树的三种遍历

## 面试注意事项、手撕代码注意事项

0.不能紧张、  
0.组织好语言再说、不要因为语速过快导致很磕巴、要理清逻辑思路。  
​  
1.题目就知道我写不出来了，就多打很多注释，解释思路，规范命名，往高代码质量方面努力。  
2.想流程、做好备注  
3.代码注意初始化问题。。可以边写边讲解。。注意做好注释。。

## 重要结构体（二叉树）

struct TreeNode {  
     int val;  
     TreeNode \*left;  
     TreeNode \*right;  
     TreeNode() : val(0), left(nullptr), right(nullptr) {}  
     TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}  
 };

## 常用数据结构及其api

unordered\_map< , > mp  
//初始化     
在初始化列表 用{,} 进行初始化  
​  
https://blog.csdn.net/charles1e/article/details/52042066

## 常见算法模板

回溯  
迪杰斯特拉  
最短路

### **回溯**

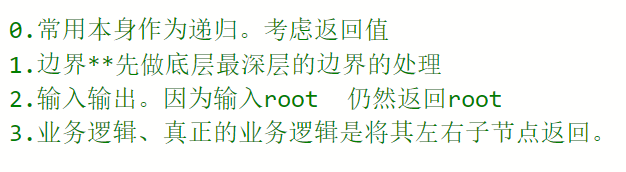
回溯的四个传入参数：输入、输出、临时输出、index  
回溯里边的内容（以剑值offer34为例）  
1.先确定边界条件  
2.根据index判断是否满足条件  
3.然后进行递归 首先在index操作、对临时tmp操作  
4.然后向下层递归  
5.然后执行回退操作。  
​

### **递归（dfs模板）**

输入、返回值、边界条件、业务处理

二叉树的递归是一大类。递归有两种、分别是自底向上或者顶向下。。  
先递归再处理业务逻辑就是自底向上、先处理业务逻辑再递归就是子顶向下。、  
典型的就是归并排序里的自底向上。  
​  
0.常用本身作为递归。考虑返回值  
1.边界\*\*先做底层最深层的边界的处理  
2.输入输出。因为输入root 仍然返回root  
3.业务逻辑、真正的业务逻辑是将其左右子节点返回。

\*\*dfs模板参考79题，单词搜索。。



## 图论--高频

### **迪杰斯特拉算法实现1-优先队列-稀疏图**

有向图的最短路径问题--计算一个顶点到各个顶点的最短路径  
特点是以起始中心向外扩展-广度优先思想。  
​  
class Solution {  
public:  
   int networkDelayTime(vector<vector<int>>& times, int N, int K) {  
       vector<int> dis(N+1,-1);   //表示初始点到达各个点的距离。。  
       dis[K]=0;                     
       using Pair=pair<int,int>;   //first是距离，second是目标点  
       priority\_queue<Pair,vector<Pair>,greater<Pair>> pq; //表示小根堆  
       pq.emplace(0,K);           //起点先入队  
          
       while(!pq.empty()){  
           auto e=pq.top();pq.pop();             //e为连接visited和unvisited的最小边  
           if(e.first>dis[e.second]) continue;   //如果e的权比K到e.second还大，就不可能缩短路径了  
           for(int i=0;i<times.size();i++){  
               if(times[i][0]==e.second){//遍历一遍所有以e.second（起点）为起点的边，做relax，并将relax之后的点入队  
                   int v=times[i][1];  
                   int w=e.first+times[i][2];  
                   if(dis[v]==-1||dis[v]>w){  
                       dis[v]=w;  
                       pq.emplace(w,v);  
                   }  
               }  
           }  
              
       }  
          
       int ans=0;  
       for(int i=1;i<=N;i++){  
           if(dis[i]==-1) return -1;  
           ans=max(ans,dis[i]);  
       }  
       return ans;  
   }  
};

### **迪杰斯特拉算法实现2-邻接矩阵-稠密图**

//两层for循环，第一层表示更新初始点到各个点的距离，第二个点表示更新的过程中以各个点作为中介点为桥梁。。  
​  
class Solution {  
public:  
   int networkDelayTime(vector<vector<int>>& times, int N, int K) {  
       vector<vector<long long>> graph(N+1,vector<long long>(N+1,INT\_MAX));  
       for(int i=1;i<=N;i++)   graph[i][i]=0;  
       for(auto e:times)   graph[e[0]][e[1]]=e[2];  
       vector<bool> visited(N+1,false);   visited[K]=true;  
          
       for(int i=1;i<N;i++){//进行一次表示，从K到j，经过一个点能不能缩短路径。一共进行N-1就可以。  
           int min\_id=0,min\_dis=INT\_MAX;//每次在unused的点中找到K最近的那个  
           for(int j=1;j<=N;j++){  
               if(visited[j]==false && graph[K][j]<min\_dis){  
                   min\_dis=graph[K][j];  
                   min\_id=j;  
               }  
           }  
           visited[min\_id]=true;//把这个点标记为“使用过了”  
           for(int j=1;j<=N;j++){//relax  
               if(graph[K][min\_id]+graph[min\_id][j]<graph[K][j]){  
                   graph[K][j]=graph[K][min\_id]+graph[min\_id][j];  
               }  
           }  
       }  
       int ans=0;  
       for(int i=1;i<=N;i++){  
           if(graph[K][i]==INT\_MAX) return -1;  
           ans=max(ans,(int)graph[K][i]);  
       }  
       return ans;  
   }  
};

### **floyd算法**

多源最短路径算法。  
求任意两点间的最短路径  
临界矩阵的方法上再加一层for循环。。  
​  
class Solution {  
public:  
   int networkDelayTime(vector<vector<int>>& times, int N, int K) {  
       vector<vector<long long>> graph(N+1,vector<long long>(N+1,INT\_MAX));  
       for(int i=1;i<=N;i++)   graph[i][i]=0;  
       for(auto e:times)   graph[e[0]][e[1]]=e[2];  
          
       for(int k=1;k<=N;k++)//k放在最外层  
           for(int i=1;i<=N;i++)  
               for(int j=1;j<=N;j++)  
                   graph[i][j]=min(graph[i][j],graph[i][k]+graph[k][j]);  
          
       int ans=0;  
       for(int i=1;i<=N;i++){  
           if(graph[K][i]==INT\_MAX) return -1;  
           ans=max(ans,(int)graph[K][i]);  
       }  
       return ans;  
   }  
};

### **SPFA算法**

class Solution {  
public:  
   int networkDelayTime(vector<vector<int>>& times, int N, int K) {  
       vector<vector<long long>> graph(N+1,vector<long long>(N+1,INT\_MAX));  
       for(int i=1;i<=N;i++)   graph[i][i]=0;  
       for(auto e:times)   graph[e[0]][e[1]]=e[2];  
          
       vector<int> disToK(N+1,INT\_MAX);  
       queue<int> q;  
       q.push(K);disToK[K]=0;  
       while(!q.empty()){  
           int front=q.front();   q.pop();  
           for(int target=1;target<=N;++target){//尝试对所有的点进行relax  
               if(disToK[front]+graph[front][target]<disToK[target]){  
                   disToK[target]=disToK[front]+graph[front][target];  
                   q.push(target);  
               }  
           }  
       }  
          
       int ans=0;  
       for(int i=1;i<=N;i++){  
           if(disToK[i]==INT\_MAX) return -1;  
           ans=max(ans,disToK[i]);  
       }  
       return ans;  
          
   }  
};

输入输出（头文件/字符串注意事项）

​

## 算法的常用API

### **字符串处理函数（<https://blog.csdn.net/qq_37941471/article/details/82107077>）**

https://blog.csdn.net/qq\_37941471/article/details/82107077  
https://blog.csdn.net/Big\_laoshu/article/details/78659135  
​  
数组的初始化int[] dp = new int[n + 1];  
​  
------增  
字符串后方插入(插入单个字符、插入字符串)  
s+'xxx'                     // 可以拼接字符串的操作、   如果向+单个元素的话   可以to\_string()  
s.push\_back('x');   s+'x'; // 向字符串添加\*\*\*\*\*单个字符    
s.append("xyz");           // 向字符串添加多个字符  
insert(i,"xyz")   //从第i位置开始插入字符串xyz  
string(num,'x');   //将num个字符转换成字符串.  
to\_string(); //将数值转换成字符串形式.   //剑指offer 45.数组排成最小的数。先将整数转整字符串的形式  
​  
改  
s.replace(i,j,"xxx")   //将下标i和下标j位置的字符串替换为  
s.resize(num);//重新进行调整//剑指050替换空格的题目  
stoi(s);     //将数字转成字符串。。  
​  
查  
s.size() ;   //查大小   \*\*\*注意其返回值是size\_t 应该将其转成int(s.size())  
s.length();  
find("ab")       //查找字符串ab第一次出现的地方  
find("ab",x)     //查找字符串从第x位以后，ab第一次出现的文职  
rfind("ab",x)   //在前x位内，查找字符串第一次出现的位置  
​  
​  
s.swap(s1); //实现两个字符串交换函数  
s.compare(s1)   //逐位按照字典序进行查找  
substr(x)       //截取x位以后的子串   //剑指58. 直接考察的就是substr的两个操作  
substr(x,i)     //截取从x开始的i个字符串 剑指46。。  
​  
---技巧  
​  
总结：字符串和vector的操作函数是有些类似的。。。剑指38题。全排列问题因为不能用重复，所以存数据的时候用的set ，可以yongset赋值给vector类型，但是不能直接返回。因为对应的操纵是不匹配的。  
​  
sort();   // 函数 字符串也可以sort函数 剑指45.    
       static bool compare(string &a,string &b){ //自定义排序 必须是static静态函数  
         return a+b<b+a;                                 //将a+b或者b+a进行字符拼接后   进行比较   从小到大排序  
       }  
to\_string() ; //剑指46.为了方便的对整数进行诸位的操作,可以先用to\_string将其转成整数   将整数或者字符转换为字符串  
stoi();   //将字符串转换为整数  
被重载的+运算符也比较常见。。  
><号   字符串可以直接比较字符//比如剑指46.，s.substr(index,2)>"25";  
+/push\_back();   //字符串的push\_back()是向其添加单个元素。+是添加多个元素。如果想用+添加单个元素。可以to\_string

### **vector的API**

https://blog.csdn.net/weixin\_41743247/article/details/90635931  
#include<vector>     //头文件  
​

初始化方式（五种。。assign相当于二次初始化。。resize也相当于二次赋值）  
vector<int>a(10); //但是size还是0。。见题621。。  
vector<int>a(10,1);  
vector<int>a(b);  
vector<int>a(b.begin(),b.begin+3);  
vector<int> a(b,b+7）;//从数组中获得初值。  
\*\*\*\*\*vector<int> ans={1,2,3,4,5};   //多个元素初始化//下次初始化的时候可以用。 //合并区间的题有用过

不使用等号也是可以的。。

类似于int这种是可以基本类型的赋值是需要int a（1）；

vector这种也可以直接vector{1，2，3}；

\*\*\*作为返回值还可以直接vector<int,int > {0,1};  
a.assign(b.begin(),b.begin()+3);//b为向量，将b的0-2个元素赋值给向量a  
a.assign(4,2);//a含有4个值为2的元素  
最后return的话可以直接vector<int>(a.begin(),a.begin()+k); //剑指40  
  
  
增（尾巴插入和其余位置插入。区域位置插入有三种，）  
a.emplace\_back(5);//向尾部添加元素  
a.insert(a.begin()+1,5);     //在a的第一个元素（从第0个算起）位置插入数值5,  
a.insert(a.begin()+1,3,5); //在a的第一个元素（从第0个算起）位置插入3个数，其值都为5  
a.insert(a.begin()+1,b+3,b+6);//b为数组，在a的第一个元素（从第0个元素算起）的位置  
  
删  
a.pop\_back();                 //删除a向量的最后一个元素  
a.erase(a.begin()+1,a.begin()+3);//删除范围  
a.erase(a.begin()+x);           //删除特定的某个值 //迭代器可以有+x操作。表明可以是随机访问迭代器。  
  
改  
a.clear();//清空a中的元素  
a.resize(10);//将a的有效元素个数调整至10个，多则删，少则补，其值随机    
a.resize(10,2);//将a的有效元素个数调整至10个，多则删，少则补，其值为2  
a.reserve(100);//将a的容量扩充至100。   //reserve 存储  
a.reverse();   //元素反转  
assign（）； ///是直接替换容器的内容。而不是重设或增加。

查  
a.back();//返回a的最后一个元素  
a.front();//返回a的第一个元素  
a.size();//返回a中元素的个数  
a.capacity();//返回a在内存中总共可以容纳的元素个数  
a.empty();//判断a是否为空，空则返回true，非空则返回false  
​

### **set和unordered\_set的API**

set<int> s;//默认升序,相当于 set<int,less<int> >。  
set<int,less<int> > s; //该容器是按升序方式排列元素。  
\*\*\*\* set<int,greater<int>> s; //该容器是按降序方式排列元素。  
​  
​  
------set  
默认是升序。可以进行自定义  
set<int > st;     //默认是升序。  
//set<int,cmp> st; //可以通过自定义将其变为降序。(通过仿函数)  
https://blog.csdn.net/weixin\_41874599/article/details/91359342  
​  
---增  
s.emplace();  
s.insert();   //添加元素.或通过迭代器插入到特定的位置之  
​  
---删(删除迭代器（索引）/具体的元素值/删除范围)  
s2.erase(s2.begin());         //删除对应的迭代器  
s2.erase(20);                 //\*\*\*\*删除某个元素值  
s2.erase(s.begin(),s.end()); //删除某个范围的值  
  
  
---查  
s.empty();  
s.size();  
s.find(elem) //如果存在就返回迭代器  
s.count  
s.lower\_bound(elem) //查找第一个大于等于该元素的迭代器  
s.upper\_bound(elem) //查找最后一个大于等于该元素的迭代器  
s,begin();  
s.end();  
​  
​  
---遍历(1.迭代器+while 2.范围for循环)  
auto iter = temp1.begin();  
while (iter!=temp1.end())  
{  
 cout<<\*iter;  
 iter++;  
}  
   
for(auto & a:num){  
   ans+=a;  
}   
​  
---小技巧  
set<int> ans;     //可以直接赋值给vector<int>类型的数据结构  
通过vector<int>(set.begin(),set.end()); //\*\*类似于强制类型转换  
//

### **map和unordered\_map的API**

初始化  
map<T1,T2> m;//默认按键的升序方式排列元素,相当于下方的less  
map<T1,T2,less<T1> > m; //该容器是按键的升序方式排列元素。  
map<T1,T2,greater<T1>> m; //该容器是按键的降序方式排列元素。\*\*\*和sort函数的greater<int >() 是比较类似的。。  
初始化实例：  
 unordered\_map<char, string> phoneMap{{'2', "abc"},{'3', "def"}};

为了更加的形象、可以使用{

{‘a’,”abc”}，

{‘b’,”def”}

}

//见力扣17、电话号码的字母组合。

//或者可以在phonemap和后边的实例后加=。  
​  
增  
mp.emplace('x',100);  
mp[]=XXX ;可以直接使用xxx进行添加  
\*\*\*insert(pair<int,int>(1,2)); //\*\*\*\*map插入的是pair数据结构。注意插入的形式。。。

对于multimap，需要insert(make\_pair()); //无序写出类别，就可以生成一个pair对象。。。

Insert（{x,xx}）； //以此可以看出{x,xx}和pair<int,int>(x,xx) //是类似的。。。  
删  
srase(); //通过迭代器删除、通过键删除、删除迭代器范围  
         //独特之处是可以通过键来进行删除  
改  
mp.clear();  
  
查  
\*\*mp.find(); //返回的是迭代器  (见lru)应该使用-> 也是一个pair<>类型的元素  
           其对元素操作要使用->first或者->second   //hot1：两数之和。。。

（。。常用mp.find（a）!=mp.end()来判断某个值是否在map中。。）  
mp.size();  
mp.count(); //返回的是值所对应的个数  
mp.empty()  
mp的[]和at（）函数。  
​  
map的遍历。

    for(auto it=mp.begin();it!=mp.end();++it){     //哈表表的遍历。。

        ans.emplace\_back(it->second);    //it是迭代器。。it使用的是间接访问符。。说明迭代器是一种指针

    }

注意：一般情况下，对于键没有特殊需要的，使用unordered\_map 因为是无序的，更节省时间，同时查找更快。。  
但是小程序体现的并不是很明显。。

### **stack&queue&priority\_queue(底层是大根堆)和pair<>。的API。**

------stack  
//弹入弹出交换  
emplace();  
pop();  
swap();  
//首（尾）元素      
top(); //对应queue的front和back  
​  
//大小或判空  
size();  
empty();  
​  
------queue/priority\_queue  
//弹入弹出交换  
emplace();  
pop();  
swap();  
//首位元素  
front();  
back();  
//大小和判空  
size();  
empty();  
​  
----priority\_queue   //典型题--剑指59题--

Priority<int> minheap。 默认是以vector实现的小根堆。。  
priority\_queue<int, vector<int>, less<int>> maxHeap;     //大根堆 从大到小。。  
priority\_queue<int, vector<int>, greater<int>> minHeap; //小根堆 从小到大。。

注意：优先队列的less和greater和map以及set是不同的。。

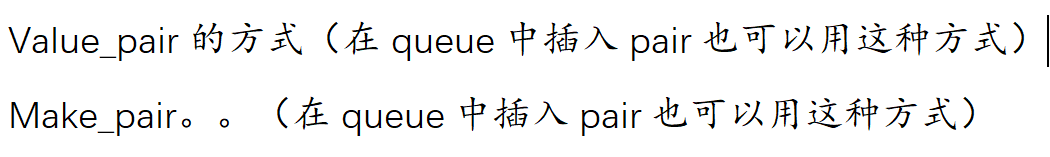
Priority\_queue的三个参数都分别是什么意思:第一个是数据类型，第二个是实现的底层容器、第三个是元素的比较方式。。

虽然说是队列、但是接口函数都是和stack相对应的。。有六个  
emplace();  
pop();  
swap();  
//首（尾）元素      
top(); //对应queue的front和back  
//大小或判空  
size();  
empty();  
​  
----------pair<>  
应用：将两个数据组合程一组，或者要返回两个值的情况  
定义：  
​  
pair<T1, T2> p1; //创建一个空的pair对象（使用默认构造），它的两个元素分别是T1和T2类型，采用值初始化。  
pair<T1, T2> p1(v1, v2);//创建一个pair对象，它的两个元素分别是T1和T2类型，其中first成员初始化为v1，second成员初始化为v2。  
make\_pair(v1, v2);         // 以v1和v2的值创建一个新的pair对象，其元素类型分别是v1和v2的类型。  
还可以拷贝构造、赋值运算符

应用vector<pair<int,int>> direction{{0，-1}，{-1，0}，{0，1}，{1，0}};

\*\*\*\*\*注意stack和queue通过top或者queue得到队首的元素之后还需要通过pop弹出。。  
自己常忘，需要注意。

* 在其他数据结构中插入pair的方式。。



在队列中可以用q.push(x,x);

在优先队列中可以用q.push（{x，x}）；

### **list和deque**

注：这两者是比较类似的，都是可以从前边插入和删除。

------list  
增  
push\_back();push\_front();//向两端插入。  
insert();  
删  
pop\_back();//从后方删除

pop\_front();//从两端删除  
erase();   //根据迭代器删除某个值、或者删除范围值  
改  
查

Size()； //在

Front(); //访问首位元素

Back(); //访问末尾元素

------deque  
初始化赋值和vector是类似的。  
--增  
de.emplace\_back();  
de.emplace\_front();  
de.insert(); //根据迭代器删除某个值、删除范围值  
​  
--删  
de.pop\_back();  
de.pop\_front();  
de.erase(); //基于迭代器删除  
​de.clear();

--改  
splice();函数三种。实现两个链表的拼接。

（splice函数的作用是将参数2（链表的）参数3迭代器所指向的链表拼接到参数1的位置。。

\_lru.splice(\_lru.begin(),\_lru,it->second);             //将该元素插入到链表的首位）

见LRU缓存机制的算法

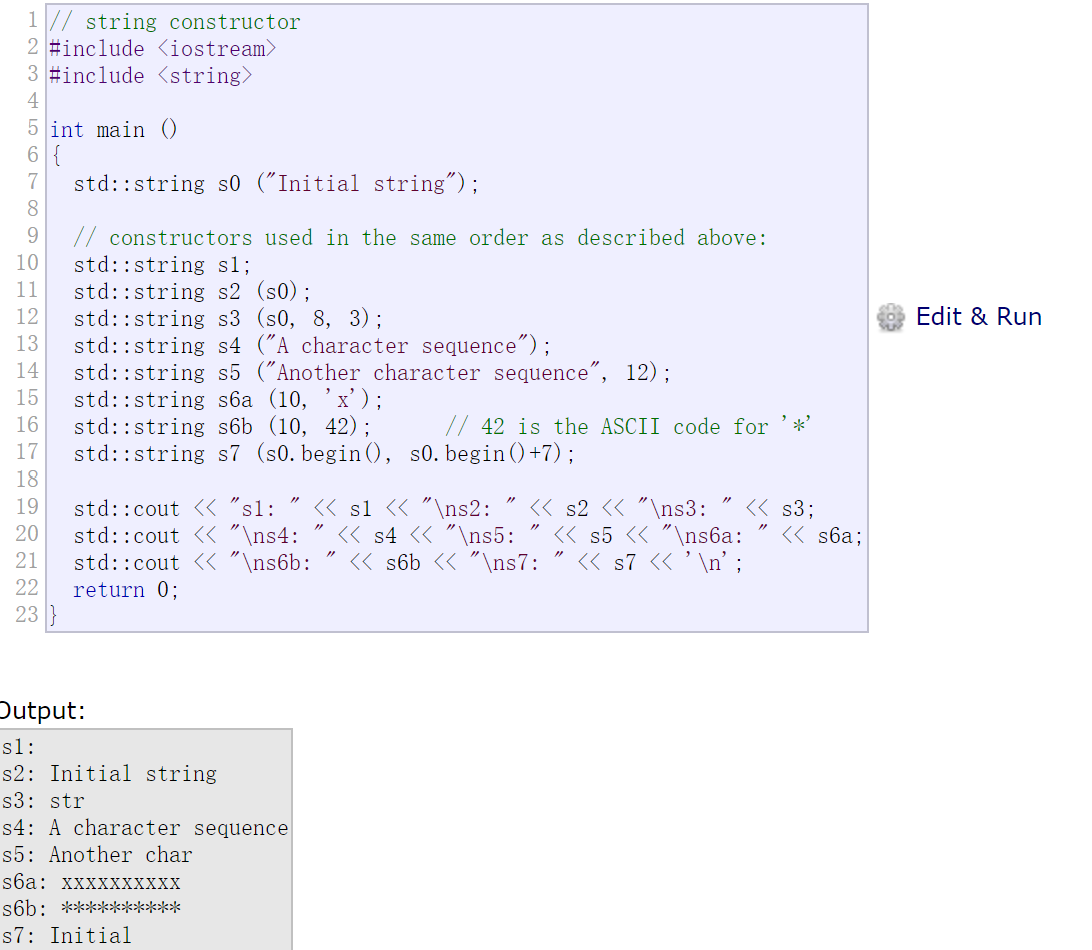
在LRU中有应用。

https://blog.csdn.net/yjunyu/article/details/77969234

​  
--查  
de.size();  
de.back();  
de.front();  
de.empty();  
de.begin(); de.end();  
de.rbegin(); //rbegin 指向的是尾元素.++是向前移动。  
​

### **String**

**String的构造函数**



* 字符串解码用到了s6a。。可以将单个字符构造成字符串。

### **位运算**

https://blog.csdn.net/deaidai/article/details/78167367  
展示自己一题多解的能力  
​  
\*注意要为unsigned int时，才能够进行位运算。。比如剑指offer56。。改成unsigned之后就可以运行  
此时&相当于取余。/2相当于>>1.的操作。。  
综上可以使用位运算，来代替加减法的计算。  
    
以下为常见的位运算符、 在进行位运算时、默认将其先变换成二进制的。  
&   剑指15. 二进制中1的个数   通过不断的 1<<i ,并进行&操作、判断某位上是否是1  
     与1相&就能够判断 某数的最后一位是否是1.  
|   |=1表示将某位的最后一位置为1     剑指56.找没出现三次的元素  
~      
>> 剑指16 n>>=1 ( 即n=n>>1 ) //   作用等于/2  
<<    
^   n^=num;  
\*在判断的时候，应该用&&   ||

### **Pair<>**

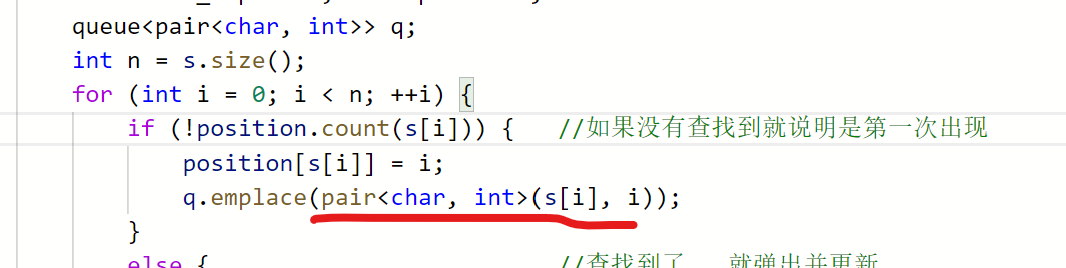
* map里的键值对实质就是pair<>;
* 347题中，vector<pair<int,int>> value;

For(auto & kv:coourrences){

Values.push\_back(kv);

}

* 在queue中插入pair



* 使用make\_pair时不需要注明类型。。
* 在list中插入pair的方式。

在lru缓存策略中，\_lru.emplace\_front(key,value);   //先往头部添加元素

直接将组成pair的两个元素拼接在一起即可。

（但是也可以直接make\_pair<int,int>()）;

### **STL的库函数总结**

1.对于erase函数。。  
stack和queue不能删除。。  
list、deque、vector//只能通过迭代器删除单个值或者范围值。。  
map和set还可以根据键删除。。  
​  
​  
2.C++11新增。cbegin(); //常量迭代器  
crbegin(); //常量反向迭代器  
emplace\_back(); //零拷贝。。  
​  
3.顺序容器增的两种方式。向端点处插入。  
或者通过迭代器向中间的位置插。。  
​  
关联容器的两种插入方式就是 insert 插入某一元素或者insert通过迭代器插入到指定的位置。

### **常用函数api**

pow(x,y);   //x的y次方  
\*\*sort(); 默认使元素从小到达排序。。

\*\*\*\*Sort（vec.begin(),vec.end(),greater<int>()）； //使得从大到小进行排序  
reverse(); //剑指06：从头打印链表   //剑指32.从上到下打印二叉树  
swap(); // 49题，还可以对字符串进行排序。  
rand() % (r-l+1) +l ; //用在快排中，注意分母不能是0.否则会出现错误。

### **注意事项（less和greater）**

对于非指针变量可以用int num(5); //类似的方式赋值  
而对于指针变量只能用Treenode\* num=nullptr; //这样的方式赋值

Greater和less。

Map、vector、sort等的less表示升序。

Priority\_queue的less表示大根堆，降序。

### **笔试输入输出注意事项**

cin 遇到空格、TAB、回车就结束  
​  
1.输入一行数据  
int n;  
vector<int> res;  
while (cin >> n) {               //如果读到数据 就把数据存入res               
 res.push\_back(n);  
 if (cin.get() == '\n') break;//点回车结束  
}

2.

//getline()  
getline (char\* s, streamsize n, char delim)   //遇到delim或者字数到达了一定的限制就会终止。  
​  
//istream& getline ( istream &is , string &str , char delim ); //&is是一个输入流、把输入流读入的字符串放在这里  
getline（cin,line,'#');   //将数据读到字符串line中 当遇到#就结束

//输出二维矩阵  
vector<vector<int>> matrix;  
print(matrix, N, M);

3.

串流的输入输出操作 stringstream  
腾讯笔试题目：计算一串字符流中的单词出现的频次的时候。  
使用如下的方式  
   string tmp;  
   getline(cin,tmp);  
   stringstream ss(tmp);  
   while(ss>>tmp){}

4.getline的dellim功能。。Getline默认是在空格处停止

大疆笔试题。读入类似 A：12，B：32，C：23 这样的数据

string a;

getline(cin,a);

stringstream ss(a);

while(getline(ss,a,',')){

stringstream sss(a);

string zimu;

string shuzi;

while(getline(sss,zimu,':')){

while(getline(sss,shuzi,':')){

cout<<zimu<<":"<<shuzi<<endl;

}

}

}

## 常见算法题

### **二分法系列问题**

### **链表题（结构体定义和初始化和）**

#include<bits/stdc++.h>  
using namespace std;  
​  
struct node {  
   int data;  
   node\* next;  
   node(int \_data) :data(\_data), next(nullptr) {  
   }  
};  
​  
 node\* init() {  
 node\* head = new node(1);  
 node\* node1 = new node(2);  
 node\* node2 = new node(3);  
 node\* node3 = new node(4);  
 node\* node4 = new node(5);  
 head->next = node1;  
 node1->next = node2;  
 node2->next = node3;  
 node3->next = node4;  
 node4->next = nullptr;  
 return head;   
 }  
​  
​  
node\* reverse(node\* head) {  
   node\* pre = new node(-1);  
   node\* temp = new node(-1);  
   pre = head;  
   temp = head->next;  
   pre->next = nullptr;   
   node\* cur = new node(-1);  
   cur = temp;  
   while (cur != nullptr) {  
       temp = cur;  
       cur = cur->next;  
       temp->next = pre;  
       pre = temp;  
   }  
 return pre;   
}  
​  
​  
int main(){  
 auto head = init();  
 head = reverse(head);  
   while (head != nullptr) {  
         cout << head->data << endl;  
         head = head->next;  
   }  
     
   system("pause");  
   return 0;   
 }

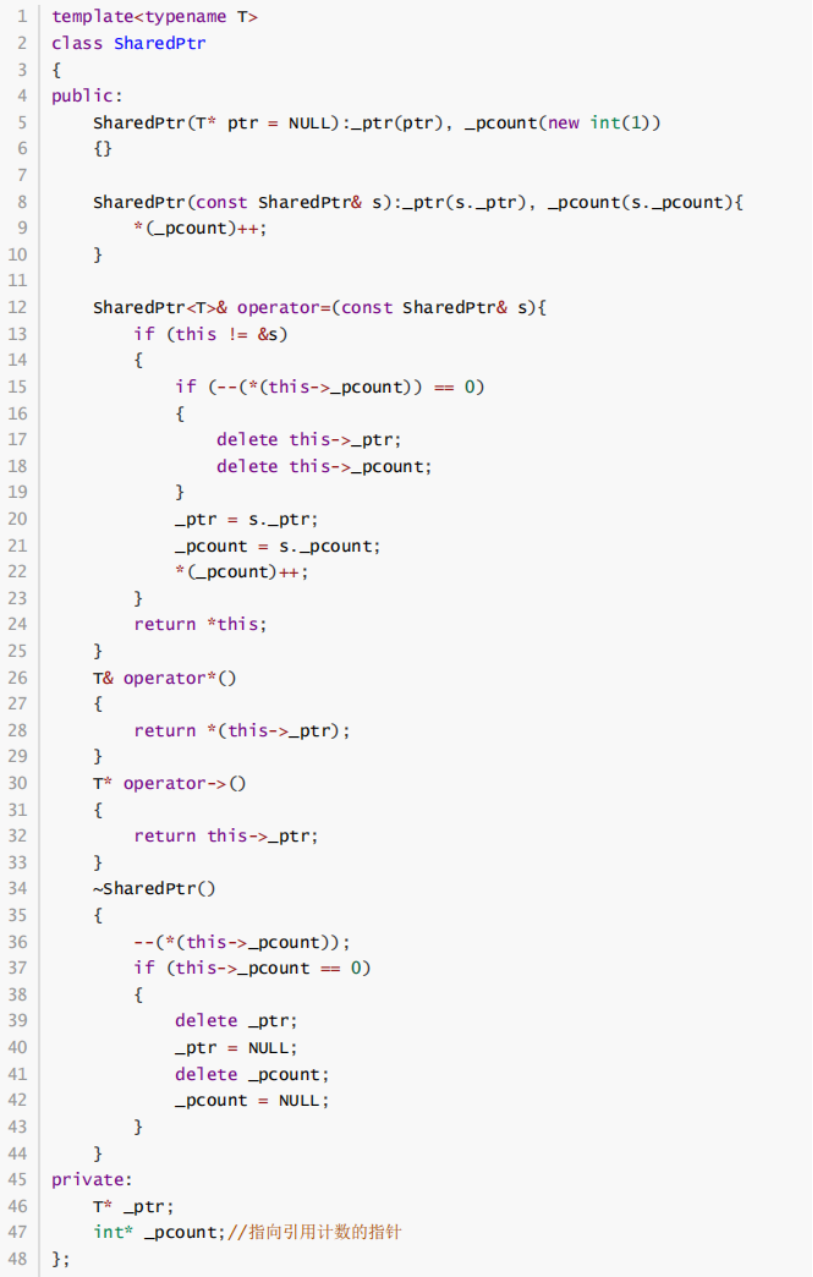
### **单例模式(构造函数私有化)**

双重检查锁--懒汉式  
​  
class singlepattern{  
private:  
     singlepattern(){};   //私有的构造函数  
     static singlepattern\* instance ; //单例未进行初始化  
public:  
     static slnglepattern\* getinstance(){  
           if(instance==NULL){   //判断是否第一调用  
               lock();  
               if(instance==NULL){ //双重检查锁  
                   instance = new singleton () ;  
               }  
               unlock();  
           }  
           return instance;  
     }   
        
     ~singlepattern(){  
       if(Signerpattern::instance)  
         delete Signerpattern::instance;  
     }  
}  
​  
​  
/\*饿汉式单例模式\*/  
class Singlepattern{  
private:  
   Singlepattern(){};         //构造函数私有化 防止被调用  
   static Singlepattern\* singlepattern;  
public:  
   static Singlepattern\* getInstance(){  
       return singlepattern;  
   }  
      
     ~singlepattern(){  
       if(Signerpattern::instance)  
         delete Signerpattern::instance;  
     }  
​  
};  
​  
Singlepattern\* Singlepattern::singlepattern=new Singlepattern(); //类外初始化：直接创建出类的实例

### **三线程交替打印ABC**

mutex mymutex;  
condition\_variable cv;  
int flag=0;  
void printa(){  
       unique\_lock<mutex> lk(mymutex);  
       int count=0;  
       while(count<10){  
       while(flag!=0) cv.wait(lk);  
       cout<<"thread 1: a"<<endl;  
       flag=1;  
       cv.notify\_all();  
       count++;  
 }  
 cout<<"my thread 1 finish"<<endl;   
}  
​  
void printb(){  
       unique\_lock<mutex> lk(mymutex);  
       for(int i=0;i<10;i++){  
       while(flag!=1) cv.wait(lk);  
       cout<<"thread 2: b"<<endl;  
       flag=2;  
       cv.notify\_all();  
 }  
 cout<<"my thread 2 finish"<<endl;   
}  
​  
void printc(){  
       unique\_lock<mutex> lk(mymutex);  
       for(int i=0;i<10;i++){  
       while(flag!=2) cv.wait(lk);  
       cout<<"thread 3: c"<<endl;  
       flag=0;  
       cv.notify\_all();  
 }  
 cout<<"my thread 3 finish"<<endl;   
}  
​  
int main(){  
       thread th2(printa);  
       thread th1(printb);  
       thread th3(printc);  
       th1.join();  
       th2.join();  
       th3.join();  
 cout<<" main thread "<<endl;  
}

### **2--智能指针代码实现**



#include "stdafx.h"  
#include<iostream>  
using namespace std;  
//引用计数类  
class counter  
{  
public:  
 counter(){};     //两种构造函数  
 counter(int parCount) :count(parCount){}  
 void increaseCount() { count++; }  
 void decreasCount(){ count--; }  
 int getCount(){ return count; }  
private:  
 int count;  
};  
   
//智能指针  
template<class T>  
class SmartPointer  
{  
public:  
 explicit SmartPointer(T\* pT) :mPtr(pT), pCounter(new counter(1)){} //防止隐性转换  
 explicit SmartPointer():mPtr(NULL),pCounter(NULL){} //构造函数2  
 ~SmartPointer()         //析构函数，在引用计数为0时，释放原指针内存  
 {  
 if (pCounter != NULL)  
 {  
 pCounter->decreasCount();  
 if (pCounter->getCount() == 0)  
 {  
 delete pCounter;  
 delete mPtr;  
 pCounter = NULL; //将pCounter赋值为NULL,防止悬垂指针  
 mPtr = NULL;  
 cout << "delete original pointer" << endl;  
 }  
 }  
 }  
   
 SmartPointer(SmartPointer<T> &rh) //拷贝构造函数，引用加1  
 {  
 this->mPtr=rh.mPtr;  
 this->pCounter = rh.pCounter;  
 this->pCounter->increaseCount();  
 }  
   
 SmartPointer<T>& operator=(SmartPointer<T> &rh) //赋值操作符，引用加1  
 {  
 if (this->mPtr == rh.mPtr)  
 return \*this;  
 this->mPtr = rh.mPtr;  
 this->pCounter = rh.pCounter;  
 this->pCounter->increaseCount();  
 return \*this;  
 }  
 T& operator\*() //重载\*操作符  
 {  
 return \*mPtr;  
 }  
   
 T\* operator->() //重载->操作符  
 {  
 return mptr;  
 }  
 T\* get()  
 {  
 return mPtr;  
 }  
private:  
 T\* mPtr;  
 counter\* pCounter;  
};  
int \_tmain(int argc, \_TCHAR\* argv[])  
{  
 SmartPointer<int> sp1(new int(10));  
 SmartPointer<int> sp2 = sp1;  
 SmartPointer<int> sp3;  
 sp3 = sp2;  
 return 0;  
}

### **线程池代码实现**

​

### **1--生产者消费者模型-百度考过**

//任务队列  
queue<int>products ;         //消费者队列  
​  
mutex m ;  
condition\_variable cond ;  
bool notify = false ;       //条件变量是否执行  
​  
bool done = false ;         //是否继续生产的标志位  
​  
void consumer(){  
   while(!done){  
       //上锁保护共享资源,unique\_lock一次实现上锁和解锁  
       unique\_lock<mutex>lk(m);  
​  
       while(!notify){           //等待生产者者通知有资源  
           cond.wait(lk);  
       }  
          
       while(!products.empty()){     //要是队列不为空的话  
           cout<<"consumer..."<<products.front()<<endl;  
           products.pop();  
           //通知生产者仓库容量不足,生产产品  
       }  
         notify = false ;  
         cond.notify\_one();  
   }  
}  
​  
void producer(){  
      
   int i ;  
   for(i=0;i<10;i++){  
 //主动让出cpu，不参与cpu 的本次调度，让其他线程使用,等一秒后再参与调度  
       unique\_lock<mutex>lk(m);  
       cout<<"producer..."<<i<<endl;  
       //如果仓库中有产品,就等待消费者消费完后在生产  
       while(notify||!products.empty()){  
           cond.wait(lk);  
       }  
​  
       products.push(i);   //当前仓库里面没有东西了,就将产品装入仓库  
​  
       notify = true ;       //设置有产品的通知  
       cond.notify\_one();           //通知消费者可以取产品了  
          
   }     
   done = true ;         //通知消费者端不生产了  
   cond.notify\_one();  
}     
​  
int main()  
{  
   thread t1(producer);  
   thread t2(consumer);  
   t1.join();  
   t2.join(); //主线程阻塞等待子线程的终止  
   return 0;  
}

### **手写bitset**

​

### **网络编程实现**

#### 服务器实现

#include <cstdio>  
#include <cstring>  
#include <cstdlib>  
#include <sys/socket.h>  
#include <sys/unistd.h>  
#include <sys/types.h>  
#include <sys/errno.h>  
#include <netinet/in.h>  
#include <signal.h>  
#define BUFFSIZE 2048  
#define DEFAULT\_PORT 16555   // 指定端口为16555  
#define MAXLINK 2048  
int sockfd, connfd;   // 定义服务端套接字和客户端套接字  
void stopServerRunning(int p)  
{  
   close(sockfd);  
   printf("Close Server\n");  
   exit(0);  
}  
int main()  
{  
   struct sockaddr\_in servaddr;   // 用于存放ip和端口的结构  
   char buff[BUFFSIZE];   // 用于收发数据  
   // 对应伪代码中的sockfd = socket();  
   sockfd = socket(AF\_INET, SOCK\_STREAM, 0);  
   if (-1 == sockfd)  
   {  
       printf("Create socket error(%d): %s\n", errno, strerror(errno));  
       return -1;  
   }  
   // END  
   // 对应伪代码中的bind(sockfd, ip::port和一些配置);  
   bzero(&servaddr, sizeof(servaddr));  
   servaddr.sin\_family = AF\_INET;  
   servaddr.sin\_addr.s\_addr = htonl(INADDR\_ANY);  
   servaddr.sin\_port = htons(DEFAULT\_PORT);  
   if (-1 == bind(sockfd, (struct sockaddr\*)&servaddr, sizeof(servaddr)))  
   {  
       printf("Bind error(%d): %s\n", errno, strerror(errno));  
       return -1;  
   }  
   // END  
   // 对应伪代码中的listen(sockfd);      
   if (-1 == listen(sockfd, MAXLINK))  
   {  
       printf("Listen error(%d): %s\n", errno, strerror(errno));  
       return -1;  
   }  
   // END  
   printf("Listening...\n");  
   while (true)  
   {  
       signal(SIGINT, stopServerRunning);   // 这句用于在输入Ctrl+C的时候关闭服务器  
       // 对应伪代码中的connfd = accept(sockfd);  
       connfd = accept(sockfd, NULL, NULL);  
       if (-1 == connfd)  
       {  
           printf("Accept error(%d): %s\n", errno, strerror(errno));  
           return -1;  
       }  
       // END  
       bzero(buff, BUFFSIZE);  
       // 对应伪代码中的recv(connfd, buff);  
       recv(connfd, buff, BUFFSIZE - 1, 0);  
       // END  
       printf("Recv: %s\n", buff);  
       // 对应伪代码中的send(connfd, buff);  
       send(connfd, buff, strlen(buff), 0);  
       // END  
       // 对应伪代码中的close(connfd);  
       close(connfd);  
       // END  
   }  
   return 0;  
}  
​

#### 客户端实现

#include <cstdio>  
#include <cstring>  
#include <cstdlib>  
#include <sys/socket.h>  
#include <sys/unistd.h>  
#include <sys/types.h>  
#include <sys/errno.h>  
#include <netinet/in.h>  
#include <arpa/inet.h>  
#define BUFFSIZE 2048  
#define SERVER\_IP "192.168.19.12"   // 指定服务端的IP，记得修改为你的服务端所在的ip  
#define SERVER\_PORT 16555           // 指定服务端的port  
int main()  
{  
   struct sockaddr\_in servaddr;  
   char buff[BUFFSIZE];  
   int sockfd;  
   sockfd = socket(AF\_INET, SOCK\_STREAM, 0);  
   if(-1 == sockfd)  
   {  
       printf("Create socket error(%d): %s\n", errno, strerror(errno));  
       return -1;  
   }  
   bzero(&servaddr, sizeof(servaddr));  
   servaddr.sin\_family = AF\_INET;  
   inet\_pton(AF\_INET, SERVER\_IP, &servaddr.sin\_addr));  
   servaddr.sin\_port = htons(SERVER\_PORT);  
   if (-1 == connect(sockfd, (struct sockaddr\*)&servaddr, sizeof(servaddr)))  
   {  
       printf("Connect error(%d): %s\n", errno, strerror(errno));  
       return -1;  
   }  
   printf("Please input: ");  
   scanf("%s", buff);  
   send(sockfd, buff, strlen(buff), 0);  
   bzero(buff, sizeof(buff));  
   recv(sockfd, buff, BUFFSIZE - 1, 0);  
   printf("Recv: %s\n", buff);  
   close(sockfd);  
   return 0;  
}  
​

### **vector实现**

#ifndef MY\_VECTOR\_H  
#define MY\_VECTOE\_H  
#include<cassert>  
typedef unsigned int size\_t;  
​  
template < class T>             //类模板     
class Vector {  
public:  
   typedef T \* iterator;       //迭代器  
      
   Vector();                     //初始化和析构  
   Vector(int size, T const& a);   
   Vector(const Vector<T> & a);   
   ~Vector();                      
      
   size\_t size() const;         //获取容量  
   size\_t capacity() const;  
   size\_t max\_capacity() const;  
   bool empty() const;  
   iterator begin() const;  
   iterator end() const;  
   T& operator[](it index);                  
      
   Vector<T>& operator=(const Vector<T>& a);   //拷贝  
      
   void push\_back(const T& val);             //压入元素  
   void pop\_back();  
​  
private:       //四个私有成员：当前大小、当前容量、最大容量、动态内存指针  
   size\_t \_size;                         
   size\_t \_capacity;  
   T\*     \_buf;  
   const size\_t \_max\_capacity = 65536;  
};  
​  
​  
template<class T>                       //函数模板  
Vector<T>::Vector()  
{     
   \_size = 0;  
   \_buf = new T[1];  
   \_capacity = 1;  
}  
​  
template<class T>  
Vector<T>::Vector(int s, const T& a)  
{     
   if (s > \_max\_capacity) {  
       s = \_max\_capacity;  
   }  
   \_size = s;  
   \_capacity = 1;  
   while (\_capacity < \_size) {         //扩容  
       \_capacity \*= 2;  
   }  
   \_buf = new T[\_capacity];  
   for (size\_t i = 0; i < \_size; i++) {  
       \_buf[i] = a;  
   }  
}  
​  
template<class T>  
Vector<T>::Vector(const Vector<T> & a)  
{  
   \_size = a.\_size;  
   \_capacity = a.\_capacity;  
   \_buf = new T[\_capacity];  
   for (size\_t i = 0; i < \_size; i++) {  
       \_buf[i] = a.\_buf[i];  
   }  
}  
​  
template<class T>  
Vector<T>::~Vector()  
{  
   delete[] \_buf;  
}  
​  
template<class T>  
size\_t Vector<T>::size() const  
{  
   return \_size;  
}  
​  
template<class T>  
size\_t Vector<T>::capacity() const  
{  
   return \_capacity;  
}  
template<class T>  
size\_t Vector<T>::max\_capacity() const  
{  
   return \_max\_capacity;  
}  
template<class T>  
T& Vector<T>::operator[](int index)  
{  
   assert(index >= 0 && index < \_size);  
   return \_buf[index];  
}  
​  
template<class T>  
void Vector<T>::push\_back(const T& val)   //考虑扩容  
{  
   if (\_size < \_capacity) {  
       \_buf[\_size] = val;  
       \_size++;  
       return ;  
   } else if (\_size == \_max\_capacity) {  
       return ;  
   }  
   \_capacity \*= 2;  
   if (\_capacity >= \_max\_capacity) {  
       \_capacity = \_max\_capacity;  
   }  
   T \* tmp = new T[\_capacity];  
   for (size\_t i = 0; i < \_size; i++) {  
       tmp[i] = \_buf[i];  
   }  
   tmp[\_size] = val;  
   \_size++;   
   delete[] \_buf;  
   \_buf = tmp;   
}  
​  
template<class T>  
void Vector<T>::pop\_back()  
{  
   assert(\_size > 0);  
   \_size--;  
}  
template<class T>  
bool Vector<T>::empty() const  
{  
   if (\_size == 0) {  
       return true;  
   }  
   return false;  
}  
​  
template<class T>  
typename Vector<T>::iterator Vector<T>::begin() const //返回头尾指针的迭代器  
{  
   return \_buf;  
}  
template<class T>  
typename Vector<T>::iterator Vector<T>::end() const  
{  
   return \_buf + \_size;  
}  
​  
template<class T>  
Vector<T>& Vector<T>::operator=(const Vector<T> & a)  
{  
   if (this == &a) {  
       return \*this ;  
   }  
   delete[] \_buf;  
   \_size = a.\_size;  
   \_capacity = a.\_capacity;  
   \_buf = new T[\_capacity];  
   for (size\_t i = 0; i < \_size; i++) {  
       \_buf[i] = a.\_buf[i];  
   }  
   return \*this;  
}  
#endif  
​

### **string类的实现**

class String    
{    
public:    
   //默认构造函数  
   String();     
   //拷贝构造函数  
   String(char \*str);  
   //拷贝构造函数  
   String(const String& other);  
   //析构函数  
   ~String();    
      
   //赋值运算符函数  
   String& operator=(char \*str);              
   String& operator=(const String& other);    
      
   //移动赋值运算符函数  
   char& operator[](unsigned int index);    
   const char& operator[](unsigned int index) const;    
    
   //拼接‘+’运算符  
   friend String operator+(const String& s1, const String& s2);    
      
      
   String& operator+=(const String& s);    
    
   friend ostream& operator<<(ostream& out, const String& s);    
   friend istream& operator>>(istream& in, String& s);    
    
   void Display();    
    
private:    
   char \*str\_;    
};

### **树的前/中/后序遍历**

前序遍历

class Solution {  
public:  
   vector<int> preorderTraversal(TreeNode\* root) {  
       vector<int> res;  
       if (root == nullptr)   return res;  
       stack<TreeNode\*> stk;  
       TreeNode\* node = root;  
       while (!stk.empty() || node != nullptr) {  
           while (node != nullptr) {  
               res.emplace\_back(node->val);     //前序和中序的区别所在  
               stk.emplace(node);  
               node = node->left;                  
           }  
           node = stk.top();         stk.pop();  
           node = node->right;                 
       }  
       return res;  
   }  
};

中序遍历

class Solution{  
public:  
   vector<int> inorderTraversal(TreeNode\* root){  
               stack< TreeNode\* > st;  
               vector<int> ans;  
               while(root!=nullptr||!st.empty()){  
                     while(root){  
                           st.push(root);  
                           root=root->left;  
                     }  
                     root=st.top(); st.pop();  
                     ans.push\_back(root->val);   //前序和中序的区别所在  
                     root=root->right;  
               }  
               return ans;  
   }  
};

后序遍历

右序的整体比较难。。需要再多看几遍。  
​  
class Solution {  
public:  
   vector<int> postorderTraversal(TreeNode \*root) {  
       vector<int> res;  
       if (root == nullptr) {  
           return res;  
       }  
       stack<TreeNode \*> stk;  
       TreeNode \*prev = nullptr;                 //进入迭代  
       while (root != nullptr || !stk.empty()) {  
           while (root != nullptr) {     //向左递归 找到最左结点  
               stk.emplace(root);  
               root = root->left;  
           }  
           root = stk.top();   stk.pop();  
           //如果该结点没有右节点（最左结点或者根节点）。或者其右节点已经被加入了。则将其加入  
           if (root->right == nullptr || root->right == prev) {  
               //1.加入到答案序列、2.将当前标记为pre、3.root置为空  
               res.emplace\_back(root->val);  
               prev = root;  
               root = nullptr;  
           } else {               //否则找到其右节点  
               stk.emplace(root);  
               root = root->right;  
           }  
       }  
       return res;  
   }  
};

### **并查集的实现**