

数据结构与算法 🍷

- 数据结构与算法 🍷
 - STL 🤖
 - 栈stack
 - 队列queue
 - 字符串string
 - 集合set
 - 双向队列deque
 - 包含在algorithm中的好用函数
 - 排序算法 🤖
 - sort排序
 - 冒泡排序
 - 快速排序
 - 归并排序
 - 经典归并排序
 - 链表归并排序
 - 求逆序对数量
 - 计数排序
 - 树tree 🤖
 - 二叉树
 - 头文件1
 - 构建二叉树结构体1
 - 构建二叉树
 - 前序序列反序列化
 - 前中序序列反序列化，无"#"
 - 删除树
 - 前序，中序，后序遍历
 - 层序遍历
 - 主函数1
 - 二叉搜索树
 - 头文件2
 - 构建二叉树结构体2
 - 查找
 - 插入1
 - 删除1

- 主函数2
- AVL树
 - AVL头文件
 - 构建二叉树结构体3
 - //获取高度
 - 更新高度
 - 维护高度属性
 - 左右旋
 - 插入2
 - 删除2
 - 自平衡
- 红黑树
- 二叉堆BinaryHeap 🤪
 - 头文件及定义
 - 上调
 - 下调
 - 插入
 - 删除顶元素
 - 朴素建堆
 - 快速建堆
- 静态查找Search 🤪
 - 顺序查找
 - 查找最大最小值
 - 查找素数
 - 埃氏筛选法
 - 欧拉筛选法
 - 二分查找
 - 查找顺序表元素
 - 未排序序列快速查找k小元素
 - 寻找最大的最小距离
- 图graph 🤪
 - 图基础功能
 - 头文件
 - 邻接矩阵的存储结构
 - 邻接链表表示
 - 用邻接矩阵法删除节点
 - 深度优先遍历
 - DFS算法实现

- 应用：寻找路径
 - 应用：走迷宫
 - 广度优先遍历
 - BFS算法实现
 - 欧拉回路
 - 无向图的割点与桥
 - 有向图的强连通分量
 - 拓扑排序
 - 最短路径
 - Dijkstra算法(边权重非负)*
 - Bellman-Ford算法
 - SPFA算法
 - 最小生成树
 - Prim算法(扩点)
 - Kruskal算法(扩边)
 - 并查集
 - Kruskal算法实现
 - 算法 🤪
 - 数据结构学习列表 🤪
-

STL 🤪

栈stack

加入头文件`#include < stack >` 初始化与vector相同`stack< int > a;` 常用函数:

- `empty()` //判断堆栈是否为空
- `pop()` //弹出堆栈顶部的元素
- `push()` //向堆栈顶部添加元素
- `size()` //返回堆栈中元素的个数
- `top()` //返回堆栈顶部的元素

队列queue

加入头文件`#include < queue >` 初始化与vector相同`queue< int > a;` 常用函数:

- `back()` //返回队列中最后一个元素
- `empty()` //判断队列是否为空
- `front()` //返回队列中的第一个元素
- `pop()` //删除队列的第一个元素
- `push()` //在队列末尾加入一个元素
- `size()` //返回队列中元素的个数

字符串string

常用函数:

- `find()`
- `to_string()` //将基本类型的值转换为字符串
- `stoi()` //将字符串类型转换为int类型
- `substr(pos,size)` //从pos开始截取size长度的字符串

集合set

加入头文件`#include < set >` 初始化与vector相同`set< int > a;` 常用函数:

- `insert()` //插入元素
- `count()` //判断是否存在某元素
- 其余各容器相似

双向队列deque

加入头文件`#include < deque >` 初始化与vector相同`deque< int > a;` 常用函数:

- `push_back()` //在队列的尾部插入元素。
- `push_front()` //在队列的头部插入元素。
- `emplace_back()` //与`push_back()`的作用一样
- `emplace_front()` //与`push_front()`的作用一样
- `pop_back()` //删除队列尾部的元素。
- `pop_front()` //删除队列头部的元素。

- `back()` //返回队列尾部元素的引用。
- `front()` //返回队列头部元素的引用。
- `clear()` //清空队列中的所有元素。
- `empty()` //判断队列是否为空。
- `size()` //返回队列中元素的个数。
- `begin()` //返回头位置的迭代器
- `end()` //返回尾+1位置的迭代器
- `insert(pos,e)` //在指定位置pos插入元素e。 vector也能用
- `insert(pos,n,e)` //在指定位置pos插入n个元素e
- `insert(pos,a,b)` //在指定位置pos插入(a,b)区间（左闭右开）的元素
- `erase(i)` //在指定i位置删除元素
- `erase(a,b)` //在指定(a,b)区间（左闭右开）删除元素

遍历:

```
deque<int>::iterator it; //迭代器定义
for(it=d.begin();it!=d.end();it++){
    cout<<*it<<" "; //注意*t和for中的!=
}
```

包含在algorithm中的好用函数

- `max_element(a.begin(),a.end())` //返回a中最大值的迭代器
- `min_element(a.begin(),a.end())` //返回a中最小值的迭代器
- `lower_bound(a.begin(),a.end(),x)` //返回第一个 $\geq x$ 的值的值的位置(a有序)，自定义comp为找到false的值
- `lower_bound(a.begin(),a.end(),x, less< type >())` //返回第一个 $\geq x$ 的值的值的位置(a有序)
- `lower_bound(a.begin(),a.end(),x, greater< type >())` //返回第一个 $\leq x$ 的值的值的位置(a有序)

- `upper_bound(a.begin(),a.end(),x)` //返回第一个 $>x$ 的值的位罝(a有序),其余相同, 自定义comp为找到true的值

排序算法 🤪

sort排序

`sort(begin,end)`中begin至end为左开右闭, 默认升序排列 `sort(begin,end,greater<type>())`降序排列 自定义排序时返回 $a>b$ 是降序排列, 即返回true值 `stable_sort()`用法相同, 但数据相等时不进行交换, 保持稳定性 匿名函数写法:

[捕捉变量列表] (参数列表) -> 返回类型{函数体}

只有一个return时可省略返回类型

□ // 未定义变量.试图在Lambda内使用任何外部变量都是错误的. `[x, &y]` // x 按值捕获, y 按引用捕获. `[&]` // 用到的任何外部变量都隐式按引用捕获 `[=]` // 用到的任何外部变量都隐式按值捕获 `[&, x]` // x显式地按值捕获. 其它变量按引用捕获 `[=, &z]` // z按引用捕获. 其它变量按值捕获

示例

```
sort(v.begin(), v.end(), [](int a, int b) {
    return a > b;    // 降序排列
});
```

冒泡排序

时间复杂度 $O(n^2)$

交换次数等于逆序对数量

```
void BubbleSort(int a[],int left,int right){
    for(int i=left;i<=right;i++){
        for(int j=right-1;i>=i;i--){
            if(a[j]>a[j+1]){
                swap(a[j],a[j+1]);
            }
        }
    }
}
```

```
    }  
  }  
}
```

快速排序

时间复杂度平均 $O(n \cdot \log(n))$, 最坏情况 $O(n^2)$

```
//优化->三数取中法取轴点  
int Partition(int a[],int left,int right){  
    int i=left;  
    int j=right-1;  
    int p=a[right];  
    while(true){  
        while(a[i]<p){  
            i++;  
        }  
        while(a[j]>p && j>left){  
            j--;  
        }  
        if(i>=j){  
            break;  
        }  
        swap(a[i],a[j]);  
        i++;  
        j++;  
    }  
    swap(a[i],p);  
    return i;  
}  
  
void QuickSort(int a[],int left,int right){  
    if(left<right){  
        int i=Partition(a,left,right);  
        QuickSort(a,left,i-1);  
        QuickSort(a,i+1,right);  
    }  
}
```

归并排序

时间复杂度 $O(n \cdot \log(n))$

经典归并排序

```

vector<int> TwoWayMerge(int a[],int lx,int rx,int ly,int ry){
    vector<int>v;
    int i=lx;
    int j=rx;
    while(i<=lx || j<=rx){
        if(j>rx || (i<=lx && a[i]<=a[j])){
            v.push_back(a[i]);
            i++;
        }
        else{
            v.push_back(a[j]);
            j++;
        }
    }
    return v;
}

//自顶向下
vector<int> MergeSort(int a[],int left,int right){
    vector<int>v;
    if(left<right){
        int m=(left+right)/2;
        MergeSort(a,left,m);
        MergeSort(a,m+1,right);
        v=TwoWayMerge(a,left,m,m+1,right);
    }
    return v;
}

//自下向上
vector<int> MergeSortUp(int a[],int left,int right){
    int len=1;
    int n=right-left+1;
    vector<int>v;
    while(len<n){
        int lx=0;
        int rx;
        int ly;
        int ry;
        while(lx<=right-len){
            int rx=lx+len-1;
            int ly=rx+1;
            int ry=min(ly+len-1,right);
            v=TwoWayMerge(a,lx,rx,ly,ry); //这一有一点问题，不能直接用v
            lx=ry+1;
        }
    }
    return v;
}

```

感觉快排和归并很像，都像是二分,感觉快排像是自顶至下的归并

链表归并排序


```

link* LinkSort(link* list1,link* list2){
    link* p1=list1;
    link* p2=list2;
    link* pre=NULL;
    while(p2!=NULL){
        while(p1!=NULL && p1->data<p2->data){
            pre=p1;
            p1=p1->next;
        }
        link* temp=p2;
        p2=p2->next;
        temp->next=p1;
        if(pre==NULL){
            list1=temp;
        }
        else{
            pre->next=temp;
        }
        p1=temp;
    }
    return list1;
}

```

求逆序对数量

```

int TwoWayInversionCount(int a[],vector<int>v,int l,int m,int r){
    int i=l;
    int j=m+1;
    int count=0;
    while(i<=m || j<=r){
        if(j>r || (i<=m && a[i]<a[j])){
            v.push_back(a[i]);
            i++;
        }
        else{
            v.push_back(a[j]);
            j++;
            count+=(m-i+1);
        }
    }
    return count;
}

int InversionCount(int a[],int l,int r){
    int count=0;
    vector<int>v;
    if(l<r){
        int m=(l+r)/2;
        InversionCount(a,l,m);
        InversionCount(a,m+1,r);
        count+=TwoWayInversionCount(a,v,l,m,r);
    }
}

```

```
    return count;
}
```

计数排序

```
vector<int> CountSort(int a[],int n,int max){
    vector<int>v(0,n);
    int c[max];
    for(int i=0;i<max;i++){
        c[i]=0;
    }
    for(int i=0;i<n;i++){
        c[a[i]]++;
    }
    for(int i=1;i<max;i++){
        c[i]=c[i]+c[i-1];
    }
    for(int i=n-1;i>=0;i--){
        v[c[a[i]]]=a[i];
        c[a[i]]--;
    }
    return v;
}
```

树tree 🌀

二叉树

头文件1

```
#include<iostream>
#include<vector>
#include<string>
#include<queue>
using namespace std;
```

构建二叉树结构体1

```
//构建二叉树结构体
typedef struct node{
    int data;
    node*left,*right;
    node(int x):data(x),left(NULL),right(NULL){} //构造函数
}BT;
```

构建二叉树

```
//构建二叉树
BT* CreatTree(int a){
    node*tree=new node(a);
    /*tree->data=a;
    tree->left=NULL;
    tree->right=NULL; 有构造函数后可省略*/
    return tree;
}
```

前序序列反序列化

```
//前序序列反序列化
int k=-1; //全局变量k
BT* PreDESer(string s){
    k++;
    int n=s.size();
    node*tree=NULL;
    if(k<n){
        int data=int(s[k]); //to_string()使int转string, stoi()是string转int
        if(s[k]!='#'){ //此处#意为空树
            tree=new node(data);
            tree->data=data;
            tree->left=PreDESer(s);
            tree->right=PreDESer(s); //重构左右子树, k决定data
        }
    }
    return tree;
}
```

前中序列反序列化, 无"#"

```
//前中序列反序列化, 无#
BT* PreInDESer(string pre, string inorder,int preIndex,int Instart, int Inend ){
    if(Instart>Inend) return NULL;
    node*tree=new node(int(pre[preIndex]));
    preIndex++;
```

```

int Index=Instart;
for(;Index<Inend;Index++){
    if(inorder[Index]==pre[preIndex]) break;
}
tree->left=PreInDESer(pre, inorder, preIndex, Instart, Index-1 );
tree->right=PreInDESer(pre, inorder, preIndex, Index+1, Inend );
return tree;
}

```

删除树

```

//删除树
void remove(BT* tree) {
    if (tree == nullptr) return;
    remove(tree->left);
    remove(tree->right);
    delete tree;
    tree = nullptr;
}

```

前序，中序，后序遍历

```

//前序，中序，后序遍历
void PreOrder(BT*tree){
    if(tree!=NULL){
        cout<<tree->data<<" ";
        PreOrder(tree->left);
        PreOrder(tree->right);
    }
}

void InOrder(BT*tree){
    if(tree!=NULL){
        InOrder(tree->left);
        cout<<tree->data<<" ";
        InOrder(tree->right);
    }
}

void PostOrder(BT*tree){
    if(tree!=NULL){
        PostOrder(tree->left);
        PostOrder(tree->right);
        cout<<tree->data<<" ";
    }
}

```

层序遍历

```
//层序遍历
void Sequence(BT*tree){
    queue<BT*>q;
    q.push(tree);
    while(!q.empty()){
        BT*node_ptr=q.front();
        q.pop();
        if(node_ptr!=NULL){
            cout<<node_ptr->data<<" ";
            q.push(node_ptr->left);
            q.push(node_ptr->right);
        }
    }
}
```

主函数1

```
int main(){
    int a=0;
    BT*tree=CreatTree(a);
    return 0;
}
```

二叉搜索树

头文件2

```
#include<iostream>
#include<vector>
#include<string>
using namespace std;
```

构建二叉树结构体2

```
//构建二叉树结构体
typedef struct node{
    int data;
    node*left,*right;
    node(int x):data(x),left(NULL),right(NULL){} //构造函数
}BT;
```

查找

```
//查找
BT* Search(BT* bst, int key){
    if(bst==NULL || bst->data==key) return bst;
    else if(bst->data < key) return Search(bst->left, key);
    else return Search(bst->right, key);
}
```

插入1

```
//插入
BT* Insert(BT* bst, int key){
    if(bst==NULL) bst=new node(key);
    else if(key<bst->data) bst->left=Insert(bst->left, key);
    else bst->right=Insert(bst->right, key);
    return bst;
}
```

删除1

```
//删除
BT* Removal(BT* bst, int key){
    //二分查找找到节点位置
    BT* node=bst;
    BT* father=NULL;
    while(node!=NULL && node->data!=key){
        father=node;
        if(key<node->data) node=node->left;
        else node=node->right;
    }
    if(node==NULL) return bst;
    //删除结点的左右子树非空，使其变为至多有一个子节点的情况，找到后继节点替换
    if(node->left!=NULL && node->right!=NULL){
        BT* temp=node;
        father=node;
        node=node->right;
        while(node->left!=NULL){
            father=node;
            node=node->left;
        }
        temp->data=node->data;
    }
    //删除
    BT* node_ptr=NULL; //把子树先存起来
    if(node->left!=NULL) node_ptr=node->right;
    else if(node->right!=NULL) node_ptr=node->left;
```

```
if(father==NULL) bst=node_ptr;
else if(node==father->left) father->left=node_ptr;
else father->right=node_ptr;
}
```

主函数2

```
int main(){
    int a=0;
    BT*tree=new node(a);
    return 0;
}
```

AVL树

AVL头文件

```
#include<iostream>
#include<vector>
#include<string>
#include<algorithm>
using namespace std;
```

构建二叉树结构体3

```
//构建二叉树结构体
typedef struct node{
    int data;
    int height; //注意新加进结构体的属性
    node*left,*right;
    node(int x):data(x),left(NULL),right(NULL),height(1){
    } //构造函数
}BT;
```

//获取高度

时间复杂度O(1)

```
//获取高度
int GetHeight(BT*tree){
```

```
    if(tree=NULL) return 0;
    else return tree->height;
}
```

更新高度

时间复杂度 $O(1)$

```
//更新高度
//想法：在插入一个节点需要更新之前所有节点，在有维护函数后感觉不需要了
void UpdateHeight(BT*tree){
    if(tree!=NULL){
        int height_l=GetHeight(tree->left);
        int height_r=GetHeight(tree->right);
        tree->height=max(height_l,height_r)+1;
    }
}
```

维护高度属性

时间复杂度 $O(n)$

```
//维护高度属性，返回树的高度
int MaintainHeight(BT*tree){
    if(tree->left==NULL && tree->right==NULL) return 1;
    int height_l=MaintainHeight(tree->left);
    int height_r=MaintainHeight(tree->right);
    tree->height=max(height_l,height_r)+1;
    return tree->height;
}
```

左右旋

时间复杂度 $O(1)$

```
//左右旋
//先更新子树再更新树
BT* LeftRotate(BT*tree){
    BT*node=tree->right;
    tree->right=node->left;
    UpdateHeight(tree);
    node->left=tree;
    UpdateHeight(node);
    return node;
}
```



```

BT* RightRotate(BT*tree){
    BT*node=tree->left;
    tree->left=node->right;
    UpdateHeight(tree);
    node->right=tree;
    UpdateHeight(node);
    return node;
}

```

插入2

```

//插入
//与二叉平衡树相同
BT* Insert(BT* bst, int key){
    if(bst==NULL) bst=new node(key);
    else if(key<bst->data) bst->left=Insert(bst->left,key);
    else bst->right=Insert(bst->right,key);
    return bst;
}

```

删除2

```

//删除
//与二叉平衡树相同
BT* Removal(BT*bst, int key){
    //二分查找找到节点位置
    BT* node=bst;
    BT* father=NULL;
    while(node!=NULL && node->data!=key){
        father=node;
        if(key<node->data) node=node->left;
        else node=node->right;
    }
    if(node==NULL) return bst;
    //删除结点的左右子树非空，使其变为至多有一个子节点的情况，找到后继节点替换
    if(node->left!=NULL && node->right!=NULL){
        BT* temp=node;
        father=node;
        node=node->right;
        while(node->left!=NULL){
            father=node;
            node=node->left;
        }
        temp->data=node->data;
    }
    //删除
    BT*node_ptr=NULL;
    if(node->left!=NULL) node_ptr=node->right;
    else if(node->right!=NULL) node_ptr=node->left;
}

```

```

    if(father==NULL) bst=node_ptr;
    else if(node==father->left) father->left=node_ptr;
    else father->right=node_ptr;
    return bst;
}

```

自平衡

时间复杂度 $O(\log(n))$

```

//自平衡,插入删除可使用上述函数, node表示插入节点或删除结点的父节点
BT* Balance(BT*tree,BT*node){
    if(tree!=node){
        if(tree->data<node->data){
            tree->left=Balance(tree->left,node);
        }
        else{
            tree->right=Balance(tree->right,node);
        } //保存根到node的路径
    } //某一子树tree=node
    UpdateHeight(tree);

    if(GetHeight(tree->left)-GetHeight(tree->right)==2){
        if(GetHeight(tree->left)<GetHeight(tree->right)){
            tree->left=LeftRotate(tree->left);
        }
        tree=RightRotate(tree);
    }
    if(GetHeight(tree->right)-GetHeight(tree->left)==2){
        if(GetHeight(tree->right)<GetHeight(tree->left)){
            tree->right=RightRotate(tree->left);
        }
        tree=LeftRotate(tree);
    }
    return tree;
}

```

应用:树高上界

红黑树

最长路径不超过最短路径的两倍. 在结构体中增加 parents 属性

插入:

插入结点默认为红

插入结点是根结点 → 直接变黑


插入结点的叔叔是红色 → 叔父爷变色, 爷爷变插入结点 重新考虑

插入结点的叔叔是黑色 → (LL,RR,LR,RL)旋转, 然后变色 与AVL相似 爷父变色

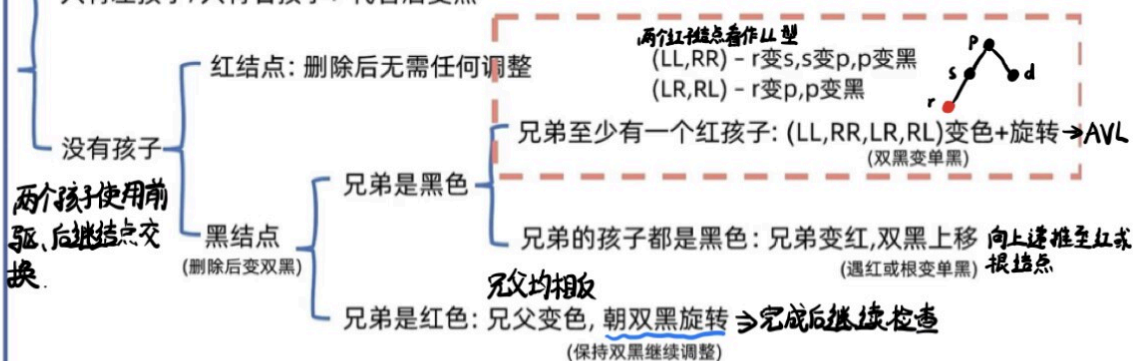
删除:

红黑树

删除

只有  这种情况, 直接使红代替黑后变黑

只有左孩子/只有右孩子: 代替后变黑



! 以上方法完成后仍需再次检查

二叉堆 BinaryHeap 🤪

最大堆, 最小堆, 最大最小堆, 对顶堆 (找到第k小的元素)

头文件及定义

```
#include<iostream>
#include<vector>
#include<string>
using namespace std;

vector<int>h; //未赋值
```

上调

时间复杂度 $O(\log(n))$

```
//上调
void SiftUp(vector<int> h,int i){ //i是起始位置
    int elem=h[i];
    while(i>1 && elem<h[i/2]){
        h[i]=h[i/2];
        i=i/2;
    }
    h[i]=elem;
}
```

下调

时间复杂度 $O(\log(n))$

```
//下调
void SiftDown(vector<int> h,int i){
    int last=h.size();
    int elem=h[i];
    int child=2*i;
    while(true){
        //找到更小的子节点
        if(child<last && h[child]>h[child+1]){
            child+=1;
        }
        else if(child>last){
            break;
        }
        //下调
        if(h[child]<elem){
            h[i]=h[child];
            i=child;
        }
        else{
            break;
        }
    }
    h[child]=elem;
}
```

插入

时间复杂度 $O(\log(n))$

```
//插入
void insert(vector<int>h,int x){
    h.push_back(x);
    SiftUp(h,h.size());
}
```

删除顶元素

时间复杂度 $O(\log(n))$

```
//删除顶元素
int DeleteMin(vector<int>h){
    int min=h[0];
    h[0]=h[h.size()-1];
    SiftDown(h,0);
    return min;
}
```

朴素建堆

时间复杂度 $O(n*\log(n))$

```
//朴素建堆
void MakeHeap(vector<int>h){
    for(int i=1;i<h.size();i++){
        SiftUp(h,i);
    }
}
```

快速建堆

时间复杂度 $O(n)$

```
//快速建堆
void MakeHeapDown(vector<int>h){
    for(int i=(h.size()/2);i>=1;i--){
        SiftDown(h,i);
    }
}
```

```
}  
}
```

静态查找Search 🤪

顺序查找

查找最大最小值

时间复杂度 $3/2 \cdot n$

```
#include<iostream>  
using namespace std;  
# define n 100  
  
int main(){  
    int a[n];  
    int max=a[0],min=a[0];  
    int k=n%2;  
    while(k<n-1){  
        if(a[k]<a[k+1]){  
            if(min>a[k]){  
                min=a[k];  
            }  
            if(max<a[k+1]){  
                max=a[k+1];  
            }  
        }  
        else{  
            if(min>a[k+1]){  
                min=a[k+1];  
            }  
            if(max<a[k]){  
                max=a[k];  
            }  
        }  
        k+=2;  
    }  
    return 0;  
}
```

查找素数

埃氏筛选法

时间复杂度 $O(n \cdot \log(\log(n)))$

```
vector<int> ESearch(int n){
    vector<int>v;
    bool prime[n+1];
    prime[0]=false;
    prime[1]=false;
    for(int i=2;i<=n;i++){
        prime[i]=true;
    }
    for(int i=2;i<=n;i++){
        if(prime[i]==true){
            v.push_back(i);
            int m=2*i;
            while(m<=n){
                prime[m]=false;
                m+=i;
            }
        }
    }
}
```

欧拉筛选法

时间复杂度 $O(n)$

```
vector<int> EulerSearch(int n){
    vector<int>v;
    bool prime[n+1];
    prime[0]=false;
    prime[1]=false;
    for(int i=2;i<=n;i++){
        prime[i]=true;
    }
    for(int i=2;i<=n;i++){
        if(prime[i]==true){
            v.push_back(i);
        }
        int k=0;
        while(i*v[k]<=n){
            prime[i*v[k]]=false;
            if(i%v[k]==0){
                break;
            }
            else{
                k++;
            }
        }
    }
}
```

二分查找

二分的思想十分重要

查找顺序表元素

时间复杂度 $O(\log(n))$

```
int BinarySearch(int a[],int left,int right,int key){
    int low=left-1;
    int high=right+1;
    while(high-low==1){
        int mid=(high+low)/2;
        if(a[mid]==key){
            return mid;
        }
        else if(a[mid]<key){
            high=mid;
        }
        else{
            low=mid;
        }
    }
    return -1;
}
```

未排序序列快速查找k小元素

先进行二分排序，再进行二分查找 时间复杂度 $O(1)$ -- $O(n^2)$

寻找最大的最小距离

时间复杂度 $O(n*\log(X[n]-X[1]))$

```
//m头牛安放进n间牛舍中，牛舍间距不同，寻找最大的最小间距
//转化为了找满足条件的最小(大)值
int MaxSpace(int a[],int n,int m){
    //a为n间牛舍的位置序列
    int min=0;
    int max=a[n-1]-a[0]+1;
    while(max-min>1){
        int mid=(max+min)/2;
        int num=1; //第num头牛
        int pre=0; //第num头牛在的位置
        for(int k=1;k<n;k++){
            if(a[k]-a[pre]>=mid){
                pre=k;
            }
        }
        if(num==m){
            return mid;
        }
        min=mid;
    }
    return max;
}
```



```

        num++;
    }
}
if(num==m){
    return mid;
}
else if(num>m){
    min=mid;
}
else{
    max=mid;
}
}
}

```

图graph 🤪

图基础功能

头文件

```

#include<iostream>
#include<vector>
#include<queue>
using namespace std;
# define maxnum 100

```

邻接矩阵的存储结构

```

struct Graph1{
    int Vex[maxnum]; //顶点表---int表示符号位置
    int Edge[maxnum][maxnum]; //边表---含权值
    int Vnum1,Enum1; //点，边数
};

```

邻接链表表示

```

struct Edge{ //相邻点链表
    int src; //表示该节点位置
    int weight; //权重

```

```

Edge *next; //指向下一节点
Edge(int x,int y):src(x),weight(y),next(NULL){}
};

struct Vex{
    int data;
    Edge *adj; //指向相邻点链表
};

struct Graph2{
    Vex vex[maxnum];
    int Vnum2,Enum2;
};

Edge* edge = new Edge(a); //构造图中线

```

用邻接矩阵法删除节点

```

Graph1 RemoveVex1(Graph1 graph,int v){
    if(v<0 &&v>graph.Vnum1){
        return graph;
    }
    //计算该节点所连边的数量
    int count=0;
    for(int u=0;u<graph.Vnum1;u++){ //删除射出的边
        if(graph.Edge[v][u]) count++;
    }
    for(int u=0;u<graph.Vnum1;u++){ //删除射入的边
        if(graph.Edge[u][v]) count++;
    }
    //改变图信息
    graph.Vex[v]=graph.Vex[graph.Vnum1-1]; //替换节点信息

    for(int u=0;u<graph.Enum1;u++){ //改变边邻接矩阵的行
        graph.Edge[v][u]=graph.Edge[graph.Vnum1][u];
    }

    for(int u=0;u<graph.Enum1;u++){ //改变边邻接矩阵的列
        graph.Edge[u][v]=graph.Edge[u][graph.Vnum1];
    }

    graph.Enum1-=count;
    graph.Vnum1-=1;

    return graph;
}

```

深度优先遍历

时间复杂度 $O(V+E)$

DFS算法实现

```
int count=0;
//邻接矩阵表示
void DFS1(Graph1 graph,int v,bool visit[]){
    visit[v]=true;
    cout<<graph.Vex[v];

    for(int u=0;u<graph.Vnum1;u++){
        if(graph.Edge[v][u]!=0 && visit[u]==false){
            DFS1(graph,u,visit);
        }
    }
}
```

```
//邻接链表表示
void DFS2(Graph2 graph,int v,bool visit[]){
    visit[v]=true;
    cout<<v;

    Edge *p=graph.vex[v].adj;
    while(p!=NULL){
        if(visit[p->src]==false){
            DFS2(graph,p->src,visit);
        }
        p=p->next;
    }
}
```

```
//总函数
void DFS(Graph1 graph){
    bool visit[graph.Vnum1];
    for(int v=0;v<graph.Vnum1;v++){
        visit[v]=false;
    }
    for(int v=0;v<graph.Vnum1;v++){
        if(visit[v]==false){
            DFS1(graph,v,visit);
            count++; //计算图中连通分量的数目
        }
    }
}
```

//扩展---输出前驱节点，发现结束时刻

```
int time=0;
int pre[maxnum];

void DFS2plus(Graph2 graph,int v,bool visit[]){
    int dfn[graph.Vnum2];
    int fin[graph.Vnum2];

    visit[v]=true;
    dfn[v]=time;
    time++;

    Edge *p=graph.vex[v].adj;
    while(p!=NULL){
        if(visit[p->src]==false){
```

```

        pre[p->src]=v;
        DFS2plus(graph,p->src,visit);
    }
    p=p->next;
}
fin[v]=time;
time++;
}

void DFSplus(Graph2 graph){
    bool visit[graph.Vnum2];
    for(int v=0;v<graph.Vnum2;v++){
        visit[v]=false;
    }
    for(int v=0;v<graph.Vnum2;v++){
        if(visit[v]==false){
            pre[v]=-1;
            DFS2plus(graph,v,visit);
        }
    }
}
}

```

应用：寻找路径

给定图，判断两点之间是否存在路径

```

bool DFS_Find(Graph2 graph,int v,int t,bool visit[],int pre[]){
    visit[v]=true;
    if(v==t){
        return true;
    }
    Edge *p=graph.vex[v].adj;
    while(p!=NULL){
        if(visit[p->src]==false){
            pre[p->src]=v;
            if(DFS_Find(graph,v,t,visit,pre)==true){
                return true;
            }
        }
        p=p->next;
    }

    return false;
}

void FindPath(Graph2 graph,int s,int t){
    int pre[graph.Vnum2];
    bool visit[graph.Vnum2];
    for(int i=0;i<graph.Vnum2;i++){
        visit[i]=false;
    }
    pre[0]=-1;
}

```

```

if(DFS_Find(graph,s,t,visit,pre)==true){
    int v=t;
    while(v>=0){
        cout<<v;
        v=pre[v];
    }
}
else{
    cout<<"No Path";
}
}

```

应用：走迷宫

```

bool Find_PathMaze(int *map[],int m,int n,int x,int y,int tx,int ty,int *pre[]){
    bool visit[m][n];
    for(int i=0;i<m;i++){
        for(int j=0;j<n;j++){
            visit[i][j]=false;
        }
    }
    visit[x][y]=true;

    int adj[4][2]={{-1,0},{1,0},{0,-1},{0,1}};

    if(x==tx && y==ty){
        return true;
    }
    for(int k=0;k<4;k++){
        int nx=x+adj[k][0];
        int ny=y+adj[k][1];
        if(nx>=0 && nx<m && ny>=0 && ny<n && map[nx][ny]!=1 && visit[nx]
[ny]==false){
            pre[nx][ny]=k;
            if(Find_PathMaze(map,m,n,nx,ny,tx,ty,pre)==true){
                return true;
            }
        }
    }
    return false;
}

```

广度优先遍历

时间复杂度 $O(V+E)$

BFS算法实现

```

int dist[maxnum]; // 距离中心节点的距离
int pre[maxnum]; // 父亲节点

void BFS2(Graph2 graph, int v, bool visit[]){
    queue<int> queue;
    queue.push(v);
    while(!queue.empty()){
        int u=queue.front();
        queue.pop();
        cout<<u;

        Edge *p=graph.vex[u].adj;
        while(p!=NULL){
            if(visit[p->src]==false){
                visit[p->src]=true;
                pre[p->src]=u;
                dist[p->src]=dist[u]+1;
                queue.push(p->src);
            }
            p=p->next;
        }
    }
}

void BFS(Graph2 graph){
    bool visit[graph.Vnum2];
    for(int v=0; v<graph.Vnum2; v++){
        visit[v]=false;
    }

    for(int v=0; v<graph.Vnum2; v++){
        visit[v]=true;
        pre[v]=-1;
        dist[v]=0;
        BFS2(graph, v, visit);
        count++; // 计算图中连通分量的数目
    }
}

```

欧拉回路

无向图的割点与桥

时间复杂度 $O(V+E)$

```

//DFS求图的割点(Tarjan算法)
int pre[maxnum];
int time=0;

void DFSvisit(Graph2 graph,int u,bool visit[]){
    visit[u]=true;
    time+=1;
    int low[graph.Vnum2];
    int dfn[graph.Vnum2];
    low[u]=time;
    dfn[u]=time;
    int children=0;
    Edge *p=graph.vex[u].adj;
    while(p!=NULL){
        if(visit[p->src]==false){
            pre[p->src]=u;
            DFSvisit(graph,p->src,visit);
            children++;
            low[u]=min(low[u],low[p->src]); //子树low更改后也要更改父节点的low
            //判断u是否为割点
            if(pre[u]==-1 && children>1){
                cout<<"u是割点"<<endl;
            }
            else if(pre[u]!=-1 && low[p->src]>=dfn[u]){
                cout<<"u是割点"<<endl;
            }
        }
        else if(pre[p->src]!=u){
            low[u]=min(low[u],dfn[p->src]);
        }
        p=p->next;
    }
}

void Tarjan(Graph2 graph){ //DFS
    bool visit[graph.Vnum2];
    for(int u=0;u< graph.Vnum2;u++){
        visit[u]=false;
    }
    for(int u=0;u<graph.Vnum2;u++){
        if(visit[u]==false){
            pre[u]=-1;
            DFSvisit(graph,u,visit);
        }
    }
}

//对于树边(u,v)如果dfn(u)<low(v),则(u,v)是桥

```

有向图的强连通分量

时间复杂度 $O(V+E)$

```

//kosaraju算法
int pre[maxnum];
int scc[maxnum];
int time=0;
int count=0;

void DFSvisit1(Graph2 graph,int u,bool visit[]){
    visit[u]=true;
    int dfn[graph.Vnum2];
    int fin[graph.Vnum2];
    time++;
    dfn[u]=time;
    Edge *p=graph.vex[u].adj;
    while(p!=NULL){
        if(visit[p->src]==false){
            pre[p->src]=u;
            DFSvisit1(graph,p->src,visit);
        }
        p=p->next;
    }
}

void DFSvisit2(Graph2 graph,int u,bool visit[]){
    visit[u]=true;
    scc[u]=count;
    Edge *p=graph.vex[u].adj;
    while(p!=NULL){
        if(visit[p->src]==false){
            DFSvisit2(graph,p->src,visit);
        }
        p=p->next;
    }
}

void Kosaraju(Graph2 graph1,Graph2 graph2){
    bool visit1[graph1.Vnum2];
    for(int u=0;u<graph1.Vnum2;u++){
        visit1[u]=false;
    }
    stack<int>s;
    for(int u=0;u<graph1.Vnum2;u++){
        if(visit1[u]==false){
            DFSvisit1(graph1,u,visit1);
        }
    }

    bool visit2[graph2.Vnum2];
    for(int u=0;u<graph2.Vnum2;u++){
        visit2[u]=false;
    }
    while(!s.empty()){
        int u=s.top();
        s.pop();
        if(visit2[u]==false){
            count++;
            DFSvisit2(graph2,u,visit2);
        }
    }
}

```



```

    }
}
//Tarjan算法...

```

拓扑排序

时间复杂度 $O(V+E)$

```

//BFS算法(顺序为BFS近似的顺序)
void CountInDegree(Graph2 graph,int n,int indegree[]){
    int indegree[n];
    for(int i=0;i<n;i++){
        indegree[i]=0;
    }
    for(int i=0;i<n;i++){
        Edge* p=graph.vex[i].adj;
        while(p!=NULL){
            indegree[p->src]++;
            p=p->next;
        }
    }
}

void TopSort_BFS(Graph2 graph,int n){
    vector<int>top_list;
    int indegree[n]; //入度
    CountInDegree(graph,n,indegree);
    queue<int>queue;
    for(int i=0;i<n;i++){
        if(indegree[i]==0){
            queue.push(i);
        }
    }
    while(!queue.empty()){
        int u=queue.front();
        queue.pop();
        top_list.push_back(u);
        Edge* p=graph.vex[u].adj; //邻接节点入度减一
        while(p!=NULL){
            indegree[p->src]--;
            if(indegree[p->src]==0){
                queue.push(p->src);
            }
            p=p->next;
        }
    }
}

//DFS算法(顺序为先输出一条路，再看其他路)
void DFS_Sort(Graph2 graph,int v,bool visit[],vector<int>top_list){
    visit[v]=true;
    Edge* p=graph.vex[v].adj;

```

```

while(p!=NULL){
    if(visit[p->src]==false){
        DFS_Sort(graph,p->src,visit,top_list);
    }
    p=p->next;
}
top_list.push_back(v);
}

void TopSort_DFS(Graph2 graph,int n,vector<int>top_list){
    bool visit[n];
    for(int i=0;i<n;i++){
        visit[i]=false;
    }
    for(int v=0;v<n;v++){
        if(visit[v]==false){
            DFS_Sort(graph,v,visit,top_list);
        }
    }
}

```

最短路径

Dijkstra算法(边权重非负)*

时间复杂度 $O(E+V\log V)$

```

struct Graph1{
    int Vex[maxnum]; //顶点表---int表示符号位置
    int Edge[maxnum][maxnum]; //边表---含权值
    int Vnum1,Enum1; //点, 边数
};

struct Edge{ //相邻点链表
    int src; //表示该节点位置
    int weight; //节点间线的权重
    Edge *next; //指向下一节点
    Edge(int x,int y):src(x),weight(y),next(NULL){}
};

struct Vex{
    int data;
    Edge *adj=NULL; //指向相邻点链表
};

struct Graph2{
    Vex vex[maxnum];
    int Vnum2,Enum2;
};

```

```

//Dij算法特有
struct heap{
    int node; //节点位置
    int dist; //距离
    heap(int x,int y):node(x),dist(y){}
};

void SiftUp(vector<heap> h,int i){ //i是起始位置
    heap elem=h[i];
    while(i>1 && elem.dist<h[i/2].dist){
        h[i]=h[i/2];
        i=i/2;
    }
    h[i]=elem;
}

void SiftDown(vector<heap> h,int i){
    int last=h.size();
    heap elem=h[i];
    int child=2*i;
    while(true){
        //找到更小的子节点
        if(child<last && h[child].dist>h[child+1].dist){
            child+=1;
        }
        else if(child>last){
            break;
        }
        //下调
        if(h[child].dist<elem.dist){
            h[i]=h[child];
            i=child;
        }
        else{
            break;
        }
    }
    h[child]=elem;
}

void insert(vector<heap>h,heap x){
    h.push_back(x);
    SiftUp(h,h.size());
}

heap DeleteMin(vector<heap>h){
    heap min=h[0];
    h[0]=h[h.size()-1];
    SiftDown(h,0);
    return min;
}

void Dijkstra(Graph2 graph,int s,int dist[],int path[]){
    bool collect[graph.Vnum2];
    for(int i=0;i<graph.Vnum2;i++){
        dist[i]=10000;
    }
}

```

```

        collect[i]=false;
    }

    vector<heap>v;
    dist[s]=0; //各点最短路径
    path[s]=-1; //父节点
    heap u(s,dist[s]); //new是用来新建指针的
    insert(v,u);
    while(!v.empty()){
        heap a=DeleteMin(v);
        if(collect[a.node]==false){ //节点未确定最短路径
            collect[a.node]=true;
            Edge* p=graph.vex[a.node].adj;
            while(p!=NULL){
                if(dist[p->src]>dist[a.node]+p->weight){
                    dist[p->src]=dist[a.node]+p->weight;
                    path[p->src]=a.node;
                    heap b(p->src,dist[p->src]);
                    insert(v,b);
                }
                p=p->next;
            }
        }
    }
}

```

Bellman-Ford算法

时间复杂度 $O(V^3)$ [最坏情况]

```

bool Bellman_Ford(Graph2 graph,int s,int dist[]){
    for(int i=0;i<graph.Vnum2;i++){
        dist[i]=10000;
    }
    dist[s]=0;
    for(int i=0;i<graph.Vnum2-1;i++){ //依据每个边来更新所有路径
        for(int u=0;u<graph.Vnum2;u++){
            Edge* p=graph.vex[u].adj;
            while(p!=NULL){
                if(dist[p->src]>dist[u]+p->weight){
                    dist[p->src]=dist[u]+p->weight;
                }
                p=p->next;
            }
        }
    }

    for(int u=0;u<graph.Vnum2;u++){ //检查是否存在环值为负值的回路
        Edge* p=graph.vex[u].adj;
        while(p!=NULL){
            if(dist[p->src]!=10000 && dist[p->src]>dist[u]+p->weight){
                return false; //存在负权环
            }
        }
    }
}

```

```

        p=p->next;
    }
}
return true;
}

```

SPFA算法

时间复杂度 $O(V \cdot E)$ [最坏情况] 空间复杂度 $O(V)$

```

void SPFA(Graph2 graph,int s,int path[],int dist[]){
    int count[graph.Vnum2]; //计数入队的次数
    bool inqueue[graph.Vnum2]; //记录是否在队列中
    for(int i=0;i<graph.Vnum2-1;i++){
        dist[i]=10000;
        count[i]=0;
        inqueue[i]=false;
    }

    dist[s]=0;
    path[s]=-1;
    queue<int>queue; //使用队列保存此轮松弛的节点，对这些节点进行下次松弛
    queue.push(s);
    inqueue[s]=true;

    while(!queue.empty()){
        int u=queue.front();
        queue.pop();
        inqueue[u]=false;
        Edge* p=graph.vex[u].adj;
        while(p!=NULL){
            if(dist[p->src]>dist[u]+p->weight){
                dist[p->src]=dist[u]+p->weight;
                path[p->src]=u;
                if(inqueue[p->src]==false){ //队列中不含重复的点
                    queue.push(p->src);
                    inqueue[p->src]=true;
                    count[p->src]++;
                }
                p=p->next;
            }
            if(count[p->src]>graph.Vnum2){
                //存在负的回路
            }
            p=p->next;
        }
    }
}

```

最小生成树

求解权值最小的生成树

Prim算法(扩点)

与Dijkstra算法完全相同，只有对临界节点的判断不同 时间复杂度 $O(V+E\log E)$ 空间复杂度 $O(V+E)$

```
struct Graph1{
    int Vex[maxnum]; //顶点表---int表示符号位置
    int Edge[maxnum][maxnum]; //边表---含权值
    int Vnum1,Enum1; //点, 边数
};

struct Edge{ //相邻点链表
    int src; //表示该节点位置
    int weight; //节点间线的权重
    Edge *next; //指向下一节点
    Edge(int x,int y):src(x),weight(y),next(NULL){}
};

struct Vex{
    int data;
    Edge *adj=NULL; //指向相邻点链表
};

struct Graph2{
    Vex vex[maxnum];
    int Vnum2,Enum2;
};

struct node{
    int index;
    int dist;
    node(int x,int y):index(x),dist(y){}
};

void SiftUp(vector<node> h,int i){ //i是起始位置
    node elem=h[i];
    while(i>1 && elem.dist<h[i/2].dist){
        h[i]=h[i/2];
        i=i/2;
    }
    h[i]=elem;
}

void SiftDown(vector<node> h,int i){
    int last=h.size();
    node elem=h[i];
    int child=2*i;
    while(true){
        //找到更小的子节点
        if(child<last && h[child].dist>h[child+1].dist){
```

```

        child+=1;
    }
    else if(child>last){
        break;
    }
    //下调
    if(h[child].dist<elem.dist){
        h[i]=h[child];
        i=child;
    }
    else{
        break;
    }
}
h[child]=elem;
}

void insert(vector<node>h,node x){
    h.push_back(x);
    SiftUp(h,h.size());
}

node DeleteMin(vector<node>h){
    node min=h[0];
    h[0]=h[h.size()-1];
    SiftDown(h,0);
    return min;
}

int Prim(Graph2 graph,int dist[],int path[]){
    bool collect[graph.Vnum2];
    for(int i=0;i<graph.Vnum2;i++){
        dist[i]=10000;
        collect[i]=false;
    }

    vector<node>v;
    dist[0]=0;
    path[0]=-1;
    node u(0,dist[0]);
    insert(v,u);
    int total_weight=0;
    while(!v.empty()){
        node a=DeleteMin(v);
        if(collect[a.index]==false){
            collect[a.index]=true;
            total_weight+=dist[a.index];
            Edge* p=graph.vex[a.index].adj;
            while(p!=NULL){
                if(collect[p->src]==false && dist[p->src]==10000){
                    dist[p->src]=p->weight;
                    path[p->src]=a.index;
                    node b(p->src,dist[p->src]);
                    insert(v,b);
                }
                p=p->next;
            }
        }
    }
}

```

```
    }  
  }  
}
```

Kruskal算法(扩边)

并查集

类似于集合合并

```
//查找元素所在集合  
int Find(int parent[],int a){  
    int root=a;  
    while(parent[root]!=root){  
        root=parent[root];  
    }  
    //路径压缩，将a放在根下  
    while(parent[a]!=root){  
        int temp=parent[a];  
        parent[a]=root;  
        a=temp;  
    }  
    return root;  
}  
  
//合并两个元素所在集合  
void Union(int parent[],int a,int b){  
    int root_a=Find(parent,a);  
    int root_b=Find(parent,b);  
    if(root_a!=root_b){  
        parent[root_b]=root_a;  
    }  
}
```

Kruskal算法实现

```
struct Edge{    //相邻点链表  
    int src;        //表示该节点位置  
    int weight;    //节点间线的权重  
    Edge *next;    //指向下一节点  
    Edge(int x,int y):src(x),weight(y),next(NULL){}  
};  
  
struct Vex{  
    int data;  
    Edge *adj=NULL; //指向相邻点链表  
};  
  
struct Graph2{
```



```

    Vex vex[maxnum];
    int Vnum2,Enum2;
};

struct edge{
    int u;
    int v;
    int weight;
    edge(int x,int y,int z):u(x),v(y),weight(z){}
};

void SiftUp(vector<edge> h,int i){ //i是起始位置
    edge elem=h[i];
    while(i>1 && elem.weight<h[i/2].weight){
        h[i]=h[i/2];
        i=i/2;
    }
    h[i]=elem;
}

void SiftDown(vector<edge> h,int i){
    int last=h.size();
    edge elem=h[i];
    int child=2*i;
    while(true){
        //找到更小的子节点
        if(child<last && h[child].weight>h[child+1].weight){
            child+=1;
        }
        else if(child>last){
            break;
        }
        //下调
        if(h[child].weight<elem.weight){
            h[i]=h[child];
            i=child;
        }
        else{
            break;
        }
    }
    h[child]=elem;
}

void insert(vector<edge>h,edge x){
    h.push_back(x);
    SiftUp(h,h.size());
}

edge DeleteMin(vector<edge>h){
    edge min=h[0];
    h[0]=h[h.size()-1];
    SiftDown(h,0);
    return min;
}

```

```

//查找元素所在集合
int Find(int parent[],int a){
    int root=a;
    while(parent[root]!=root){
        root=parent[root];
    }
    //路径压缩，将a放在根下
    while(parent[a]!=root){
        int temp=parent[a];
        parent[a]=root;
        a=temp;
    }
    return root;
}

//合并两个元素所在集合
void Union(int parent[],int a,int b){
    int root_a=Find(parent,a);
    int root_b=Find(parent,b);
    if(root_a!=root_b){
        parent[root_b]=root_a;
    }
}

int Kruskal(Graph2 graph){
    vector<edge>v; //堆
    int parent[graph.Vnum2];
    for(int i=0;i<graph.Vnum2;i++){
        parent[i]=i;
        Edge* p=graph.vex[i].adj;
        while(p!=NULL){
            edge e(i,p->src,p->weight);
            insert(v,e);
            p=p->next;
        }
    }

    int total_weight=0;
    while(!v.empty()){
        edge e=DeleteMin(v);
        if(Find(parent,e.u)!=Find(parent,e.v)){ //保证不会形成环
            total_weight+=e.weight;
            Union(parent,e.u,e.v);
        }
    }
    return total_weight;
}

```

算法 🤪

动态规划，贪心算法，回溯算法，二分法等等

数据结构学习列表

- ☐ 栈与队列
- ☐ 排序算法
- ☐ 二叉树
- ☐ 二叉堆
- ☐ 静态查找
- ☐ 图
- ☐ 算法