```
# include <iostream>
# include <stack>

# define MAXSIZE 100

using namespace std;

typedef int ElemType;

/// 线性表
//线性表的顺序存储
typedef struct {
    ElemType data[MAXSIZE];
    int length;
} SqList;

// 线性表的链式存储
typedef struct LNode {
    ElemType data;
    struct LNode *next;
} LNode, *LinkList;

//双链表
typedef struct DNode {
    ElemType data;
    struct DNode *prior, *next;
} DNode, *DLinkList;

//静态链表
typedef struct {
    ElemType data;
    int next;
} SLinkList[MAXSIZE];

//线性表的应用

/// 栈和队列
//顺序栈
typedef struct {
    ElemType data[MAXSIZE];
    int top;    //栈顶指针，初始为-1，栈顶元素S.data[S.top]
} SqStack;

//链栈
typedef struct LinkNode {
    ElemType data;
    struct LinkNode *next;
} LiStack;
```

```
//顺序队列
//存在假溢出
typedef struct {
    ElemType data[MAXSIZE];
    int front, rear;    //初始条件(队空)：Q.front == Q.rear == 0
} SqQueue;

//循环队列
//-----------------------------------------------
//将顺序队列想象为环状空间
//初始时：Q.front = Q.rear = 0;
//队首指针进1：Q.front = (Q.front + 1) % Maxsize;
//队尾指针进1：Q.rear = (Q.rear + 1) % Maxsize;
//队列长度：(Q.rear + Maxsize - Q.front) % Maxsize
//-----------------------------------------------

//链式队列
typedef struct {
    ElemType data;
    struct LinkNode *next;
} LinkNode;

typedef struct {
    LinkNode *front, *rear;
} LinkQueue;

//多维数组的存储

//特殊矩阵的压缩存储

//栈、队列和数组的应用

//栈的括号匹配
bool isValid(string s) {
    stack<char> st;
    for (char i: s) {
        if (isLeft(i)) {
            st.push(i);
        } else {
            if (st.empty())
                return false;
            if (isMatch(st.top(), i)) {
                st.pop();
            } else {
                // 不合法
                return false;
            }
        }
```

```
    }
    if (st.empty())
        return true;
    return false;
}

//串的存储
//顺序存储
typedef struct {
    char ch[MAXSIZE];
    int length;
} SString;

//堆存储
struct HString {
    char *ch;
    int length;
};

void InitString(HString &S) {
    S.length = 0;
    S.ch = NULL;
}

//生成一个其值等于串常量cahrs的串T
void StrAssign(HString &T, char *chars) {
    int i, j;
    if (T.ch)
        free(T.ch);
    i = strlen(chars);
    if (!i)
        InitString(T);
    else {
        T.ch = (char*)malloc(i * sizeof(char));
        if (!T.ch)
            exit(OVERFLOW);
        for (j = 0; j < 1; j++)
            T.ch[j] = chars[j];
        T.length = i;
    }
}

/// 树和二叉树
//顺序二叉树
//每个节点在顺序存储的结构中都有固定的位置

//链式二叉树
typedef struct BiTNode {
    ElemType data;
```

```
    struct BiTNode *lchild, *rchild;
} BiTNode, *BiTree;

//二叉树的遍历
//先序遍历
void PreOrder(BiTree T) {
    if (T != NULL) {
        visit(T);
        PreOrder(T->lchild);
        PreOrder(T->rchild);
    }
}

//中序遍历
void InOrder(BiTree T) {
    if (T != NULL) {
        InOrder(T->lchild);
        visit(T);
        InOrder(T->rchild);
    }
}

//后序遍历
void PostOrder(BiTree T) {
    if (T != NULL) {
        PostOrder(T->lchild);
        PostOrder(T->rchild);
        visit(T);
    }
}

//层次遍历
void LevelOrder(BiTree T) {
    InitQueue(Q);
    BiTree p;
    EnQueue(Q, T);                  //根节点入队
    while (!IsEmpty(Q)) {
        DeQueue(Q, p);              //队头节点出队
        visit(p);                   //访问出队节点
        if (p->lchild != NULL)
            EnQueue(Q, p->lchild);  //左子树不为空，则左子树根节点入队
        if (p->rchild != NULL)
            EnQueue(Q, p->rchild);  //右子树不为空，则右子树根节点入队
    }
}

//非递归方式的遍历
void InOrder2(BiTree T) {
    InitStack(S);
```

```
    BiTree p = T;
    while (p || !IsEmpty(S)) {
        if (p) {            //一路向左
            Push(S, p);
            p = p->lchild;
        }
        else {
            Pop(S, p);
            visit(p);
            p = p->rchild;
        }
    }
}

void PreOrder2(BiTree T) {
    InitStack(S);
    BiTree p = T;
    while (p || !IsEmpty(S)) {
        if (p) {            //一路向左
            visit(p);
            Push(S, p);
            p = p->lchild;
        }
        else {
            Pop(S, p);
            p = p->rchild;
        }
    }
}

//后序遍历的非递归实现较为复杂

//线索二叉树的基本概念和构造
//以一定的规则将二叉树中的节点排列成一个线性序列，每个节点（除了第一个和最后一个）都有一个直接前驱和直接后继。

//1. 若结点有左子树，则其 lchild 域指示其左孩子，否则令 lchild 域指示其前驱。
//2. 若结点有右子树，则其 rchild 域指示其右孩子，否则令 rchild 域指示其后继。
//3. 为了避免混淆，增加两个标志位 LTag 和 RTag 。

//二叉树的线索化是将空指针改为线索的过程，实质是遍历一次二叉树。

typedef struct ThreadNode {
    ElemType data;
    struct ThreadNode *lchild, *rchild;
    int ltag, rtag;
} ThreadNode, *ThreadTree;

//中序线索二叉树的构造
```

```
void InThread(ThreadTree &p, ThreadTree &pre) {
    if (p != NULL) {
        InThread(p->lchild, pre);    //递归，线索化左子树
        if (p->lchild == NULL) {           //左子树为空，建立前驱线索
            p->lchild = pre;
            p->ltag = 1;
        }
        if (pre != NULL && pre->rchild == NULL) {
            pre->rchild = p;     //建立前驱节点的后继线索
            pre->rtag = 1;
        }
        pre = p;     //标记当前节点为刚刚访问的节点
        InThread(p->rchild, pre);  //递归，线索化右子树
    }
}

//中序线索二叉树的遍历

//树、森林

//树的存储结构
//双亲表示法，孩子表示法，孩子兄弟表示法

//森林和二叉树的转化

//树和森林的转换

//树和森林的遍历
//树：先根遍历，后根遍历
//森林：先序遍历，中序遍历

//树与二叉树的应用
//（树形查找）

//二叉排序树（BST）
//左子树节点值 < 根节点值 < 右子树节点值

//平衡二叉树
//任意节点的左右子树高度差的绝对值不超过1
//平衡二叉树的插入

//哈夫曼树（最优二叉树）和哈夫曼编码
//路径：从树中一个结点到另一个结点之间的分支构成这两个结点之间的路径，路径上的分支数目称称做路径长度。
//树的路径长度是从树根到每一个结点的路径长度之和。
//哈夫曼编码：为了编码尽可能的短，故设计长短不等的编码，则必须是任一字符的编码都不是另一个字符的编码的前缀，这种编码称做前缀编码。

//红黑树（RBT）   重点：定义，性质，插入
//适合频繁插入和删除的场景
```

```
//①每个结点或是红色，或是黑色的
//②根节点是黑色的
//③叶结点（外部结点、NULL结点、失败结点）均是黑色的
//④不存在两个相邻的红结点（即红结点的父节点和孩子结点均是黑色）
//⑤对每个结点，从该节点到任一叶结点的简单路径上，所含黑结点的数目相同

//左根右，根叶黑，不红红，黑路同。

//结点的"黑高"：从某结点出发（不含该结点）到达任一空叶结点的路径上黑结点总数。

//红黑树的插入：！！！！

//并查集及其应用
//树的存储：双亲表示法
# define SIZE 13

//表示双亲结点的数组
int UFSets[SIZE];

void Initial(int s[]) {
    for (int i = 0; i < SIZE; i++) {
        s[i] = -1;
    }
}

int Find(int s[], int x) {
//    循环寻找x的根
    while (s[x] >= 0)
        x = s[x];
    return x;
}

//压缩路径：先找到根节点，再将查找路径上所有结点都挂到根节点下
int Find2(int s[], int x) {
    int root = x;
    while (s[root] >= 0)    //循环找到根
        root = s[root];
    //压缩路径
    while (x != root) {
        int t = s[x];   //t指向x的父节点
        s[x] = root;    //x直接挂在根节点上
        x = t;
    }
    return root;
}

void Union(int s[], int root1, int root2) {
    if (root1 == root2)
```

```
        return;
    s[root2] = root1;
}

//并操作的优化：1.根节点的绝对值表示树的结点总数。2.union操作，小树合并到大树。
void Union2(int s[], int root1, int root2) {
    if (root1 == root2)
        return;
    if (s[root2] > s[root1]) {
        s[root1] += s[root2];
        s[root2] = root1;
    }
    else {
        s[root2] += s[root1];
        s[root1] = root2;
    }
}

//应用：判断图的连通分量，判断图是否有环，Kruskal算法

/// 图
//邻接矩阵
# define MaxVertexNum 100    //顶点数目的最大值
typedef char VertexType;     //顶点的数据类型
typedef int EdgeType;        //带权图中边上权值的数据类型
typedef struct {
    VertexType Vex[MaxVertexNum];    //顶点表
    EdgeType Edge[MaxVertexNum][MaxVertexNum];  //邻接矩阵，边表
    int vexnum, arcnum; //图的当前顶点数和弧数
} MGraph;

//邻接表
# define MaxVertexNum 100
typedef struct ArcNode {     //边节点
    int adjvex;              //该弧所指向的顶点的位置
    struct ArcNode *next;    //指向下一条弧的指针
    InfoType info;           //网的边权值
} ArcNode;

typedef struct VNode {       //顶点节点
    VertexType data;         //顶点信息
    ArcNode *first;          //指向第一条依附该顶点的弧的指针
} VNode, AdjList[MaxVertexNum];

typedef struct {
    AdjList vertices;        //邻接表
    int vexnum, arcnum;      //图的顶点数和弧数
} ALGraph;
```

```
//邻接多重表
# define MAX_VERTEX_NUM 20
typedef enum {
    unvisited, visited
} VisitIf;
typedef struct EBox {
    VisitIf mark;
    int ivex, jvex;
    struct EBox *ilink, *jlink;
    InfoType *info;
} EBox;

typedef struct VexBox {
    VertexType data;
    EBox *firstedge;
} VexBox;

typedef struct {
    VexBox adjmulist[MAX_VERTEX_NUM];
    int vexnum, edgenum;
} AMLGraph;

//十字链表
# define MAX_VERTEX_NUM 20
typedef struct ArcBox {
//  该弧的尾和头顶点的位置
    int tailvex, headvex;
    struct ArcBox *hlink, *tlink;
    InfoType *info;
} ArcBox;

typedef struct VexNode {
    VertexType data;
    ArcBox *firstin, *firstout;
} VexNode;

typedef struct {
    VexNode xlist[MAX_VERTEX_NUM];
    int vexnum, arcnum;
} OLGraph;

//图的遍历

//深度优先（DFS）

//广度优先（BFS）

//图的应用
//最小生成树：Prim，Kruskal
```

```
//最短路径：Dijkstra，Floyd

//有向无环图：DAG图
//拓扑排序
//在一个表示工程的有向图中，用顶点表示活动，用弧表示活动之间的优先关系，这样的有向图为顶点表示活动的网，称为AOV网。
// AOV网中的弧表示活动之间存在的某种制约关系。

//在AOV网中，若不存在回路，则所有活动可排列成一个线性序列，使得每个活动的所有前驱活动都排在该活动的前面，
// 我们把此序列叫做拓扑序列(Topological order)。

//拓扑排序就是对一个有向图构造拓扑序列的过程。

//关键路径
//AOE网(Activity On Edge)是在AOV网的基础上，其中每一个边都具有各自的权值，是一个带权有向无环网。
//通常，其中权值表示活动持续的时间。
//
//由于在AOE网中有些活动可以并行地进行，所以完成工程的最短时间是从开始点到完成点的最长路径的长度
//(指路径上各活动持续时间之和，不是路径上弧的数目)。路径长度最长的路径叫做关键路径(Critical Path)。

/// 查找
//查找的基本概念：动态查找表，静态查找表
//顺序查找法：使用哨兵优化，从后往前找
int Sequential_Search(int *a, int n, int key) {
    int i;
    a[0] = key;
    i = n;
    while (a[i] != key)
        i--;
    return i;
}

//分块查找法
//块内元素无序，块间有序。

//折半查找法：适用于有序的线性表
int Search_bin(int *a, int n, int key) {
    int low, high, mid;
    low = 1;
    high = n;
    while (low <= high) {
        mid = (low + high) / 2;
        if (key < a[mid])
            high = mid - 1;
        else if (key > a[mid])
            low = mid + 1;
        else
            return mid;
```

```
    }
    return -1;
}

//B树（多路平衡查找树）及其基本操作，B+树的基本概念

//散列表

//字符串模式匹配
//KMP

//查找算法的分析及应用

/// 排序
//排序的基本概念

//插入排序
//直接插入排序
void InsertSort(ElemType A[], int n) {
    int i, j;
    for (i = 2; i <= n; i++) {                  //依次将A[2]~A[n]插入前面已排序序列
        if (A[i] < A[i - 1]) {                      //A[i]小于其前驱
            A[0] = A[i];                            //复制为哨兵，A[0]不存放元素
            for (j = i - 1; A[0] < A[j]; j--)       //从后往前查找待插入位置
                A[j + 1] = A[j];                    //向后挪位
            A[j + 1] = A[0];                        //复制到插入位置
        }
    }
}

//折半插入排序
void InsertSort2(ElemType A[], int n) {
    int i, j, low, high, mid;
    for (i = 2; i <= n; i++) {
        A[0] = A[i];
//        折半查找范围
        low = 1;
        high = i - 1;
        while (low <= high) {
            mid = (low + high) / 2;
            if (A[mid] > A[0])
                high = mid - 1;
            else
                low = mid + 1;
        }
        for (j = i - 1; j >= high + 1; j--)
            A[j + 1] = A[j];
        A[high + 1] = A[0]; //插入操作
    }
```

```
}

//起泡排序(bubble sort)
void BubbleSort(ElemType A[], int n) {
    for (int i = 0; i < n - 1; i++) {
        bool flag = false;
        for (int j = n - 1; j > i; j--) {
            if (A[j - 1] > A[j]) {
                swap(A[j - 1], A[j]);
                flag = true;
            }
//              本趟遍历后没有发生交换，说明表已经有序
            if (flag == false)
                return;
        }
    }
}

//简单选择排序
void SelectSort(ElemType A[], int n) {
    for (int i = 0; i < n - 1; i++) {
        int min = i;
        for (int j = i + 1; j < n; j++)
            if (A[j] < A[min])
                min = j;
        if (min != i)
            swap(A[i], A[min]);
    }
}

//希尔排序(shell sort)
void ShellSort(ElemType A[], int n) {
    int i, j, dk;
    for (dk = n / 2; dk >= 1; dk = dk / 2) {
        for (i = dk + 1; i <= n; i++) {
            if (A[i] < A[i - dk]) {
                A[0] = A[i];
                for (j = i - dk; j > 0 && A[0] > A[j]; j -= dk)
                    A[j + dk] = A[j];
                A[j + dk] = A[0];
            }
        }
    }
}

//快速排序
//确定枢轴
int Partition(ElemType A[], int low, int high) {
    ElemType pivot = A[low];    //设置第一个元素为枢轴
```

```
    while (low < high) {
        while (low < high && A[high] >= pivot)
            --high;
        A[low] = A[high];    //将比枢轴小的元素移动到左端
        while (low < high && A[low] <= pivot)
            ++low;
        A[high] = A[low];    //将比枢轴大的元素移动到右端
    }
    A[low] = pivot;
    return low;
}

void QuickSort(ElemType A[], int low, int high) {
    if (low < high) {
        int pivotpos = Partition(A, low, high);
        QuickSort(A, low, pivotpos - 1);
        QuickSort(A, pivotpos + 1, high);
    }
}

//堆排序


//二路归并排序(mergesort)


//基数排序


//外部排序


//各种排序算法的比较
//排序算法的分析和应用


int main() {
    return 0;
}
```