

- 栈(先进后出)
 - 括号匹配算法
 - 链栈
- 队列
 - 双栈实现队列
 - 数组实现循环队列
 - 优先队列
 - 链式队列
- 递归
 - 全排列
 - 非字典序输出
 - 字典序
 - 爬楼梯问题
 - 汉诺塔
 - 八皇后问题
- 搜索(数组必须为已排序)
 - Binary Search查找数据,返回下标
 - 查找一个数据在数组中第一次出现,返回下标
 - 查找一个数据在数组中最后一次出现
 - 插值查找
 - 斐波那契查找
- 排序
 - 快速排序
 - 归并排序
 - 堆排序
 - shell
 - 选择
 - 插入排序
 - 冒泡
- 哈希表
 - 哈希表
 - 简单哈希
 - 冲突解决方法
 - 开二维数组
 - 线性探测法
 - 二次探测(与线性探测一样, 就是把 $temp++$ 换成 $temp = temp * temp$)
- 树
 - 插入
 - 查找
 - 删除
 - 二叉树的遍历
 - 前序遍历

- 递归
- 非递归
- 中序遍历
 - 递归
 - 非递归
- 后序遍历
 - 递归
 - 非递归
- 二叉树层次遍历
- 创建树
 - 中序和后序建树
 - 先序后序建树
 - 二叉树层序输入建树
 - 递归版
 - 非递归
- 二叉树的销毁
- 二叉树综合
- 查找二叉树中最大最小值
- 求树的高度
- 二叉树中某一条路径节点数之和最大
- 求二叉树的宽度
- 二叉树最大宽度
- 求一颗树中有多少个叶子节点
- 哈夫曼编码
- 最小生成树
 - prim算法
 - krusal算法
- AVL树(BF是平衡因子)
 - 判断是否为AVL树
 - 左旋
 - 右旋
 - 先右旋后左旋
 - 先左旋后右旋
- 堆
 - 堆结构定义(以下均为最大堆(即为大顶堆), 最小堆和最大堆一样, 只是将>改为<)
 - 最大堆初始化
 - 插入节点
 - 删除节点
- 图
 - 拓扑排序
 - 欧拉回路的判断
 - 最短路径
 - 算法原理:
 - 使用优先队列版本

- 不使用优先队列
- floyd-warshall 任意两点间的最短路径，可以解决有向图或者负权问题
- floyd-warshall打印路径
- 两城市间最短路径，若最短路径有多个，输出花费最少的
- 两城市间最短路径，花费最多而且打印路径
- 广度优先搜索，深度优先搜索
 - bfs原型
 - dfs原型
 - 应用
 - 列出一个图中的连通集
 - 判断图是否连通
 - 走迷宫
 - dfs迷宫最短路径并输出最小步数
 - dfs打印迷宫所有路径
 - 怎么计下当前遍历为第几层(解决六度空间理论)
 - dfs实现类24点
 - dfs寻找从某一点出发最长路径
 - bfs最长路径
- 任意进制的转换
- 打素数表//求第n个素数
 - 数组实现
 - 链表实现
- 乘法转换成加法
- 表达式计算
 - 前缀表达式求值(都可以判断是否为合法输入)
 - 递归版
 - 非递归版
 - 中缀转后缀表达式
 - 表达式求值//中缀输入，转为后缀，求值
- 并查集
- STL
 - 排序
 - 各容器的删除
 - map
 - vector
 - queue
 - deque
 - set(set默认从小打到排序)
 - list
 - stack
 - multimap

栈(先进后出)

括号匹配算法

```
#include <bits/stdc++.h>
using namespace std;

int main(int argc, char const *argv[])
{
    int t = 0;
    string str = "";
    cin >> t;
    getline(cin, str);//space
    while(t--) {
        bool flag = true;
        getline(cin, str);
        stack<char> bracket;
        for(int i = 0; i < str.length(); i++) {
            switch(str[i]) {
                case '(':
                case '[':
                case '{':
                    bracket.push(str[i]);
                    break;
                case ')':
                    if(!bracket.empty() && bracket.top() == '(') {
                        bracket.pop();
                    } else {
                        flag = false;
                    }
                    break;
                case ']':
                    if(!bracket.empty() && bracket.top() == '[') {
                        bracket.pop();
                    } else {
                        flag = false;
                    }
                    break;
                case '}':
                    if(!bracket.empty() && bracket.top() == '{') {
                        bracket.pop();
                    } else {
                        flag = false;
                    }
                    break;
                default:
                    break;
            }
        }
        if(!bracket.empty()) {
            flag = false;
        }
        if(flag == true) {
            cout << "True" << endl;
        } else {
            cout << "False" << endl;
        }
    }
}
```

```
    return 0;
}
```

链栈

```
#include <bits/stdc++.h>
using namespace std;

enum ErrorCode {
    success,
    underflow,
    overflow
};

template <class StackEntry>
struct Node {
    StackEntry data;
    Node *next;
};

template <class StackEntry>
class MyStack {
public:
    MyStack() {
        pTop = NULL;
    }
    /*
    判断栈是否为空，若为空则返回true，非空则返回false
    */
    bool empty() const {
        return pTop == NULL;
    }
    int size() const {
        int size = 0;
        Node<StackEntry> * tmp = pTop;
        while(tmp != NULL) {
            size++;
            tmp = tmp -> next;
        }
        return size;
    }
    /*
    出栈操作，若正常出栈则返回success，若栈内无元素则返回underflow
    */
    ErrorCode pop() {
        if(empty()) {
            return underflow;
        } else {
            Node<StackEntry> * tmp = pTop;
            pTop = pTop -> next;
            delete tmp;
            tmp = NULL;
            return success;
        }
    }
    /*
    获取栈顶元素，若正常获取到栈顶元素则返回success，若栈内无元素则返回underflow
    */
}
```

```

元素内容保存在引用参数item中
*/
ErrorCode top(StackEntry &item) const {
    if(empty()) {
        return underflow;
    } else {
        item = pTop -> data;
        return success;
    }
}
/*
入栈操作，若正常入栈则返回success，若入栈失败则返回overFlow
*/
ErrorCode push(const StackEntry &item) {
    Node<StackEntry> * tmp = new Node<StackEntry>;
    if(tmp == NULL) {
        return overflow;
    }
    tmp -> data = item;
    tmp -> next = pTop;
    pTop = tmp;
    return success;
}
/*
清空栈
*/
void clear() {
    while(pTop != NULL) {
        pop();
    }
}
private:
    // 请不要修改数据成员。
    Node<StackEntry> *pTop;
};

```

队列

双栈实现队列

```

class Queue {
public:
    bool IsEmpty() const {
        return firstStack.size() == 0;
    }
    int& front() {
        return firstStack.top();
    }
    void push(int a) {
        if(firstStack.empty()) {
            firstStack.push(a);
        } else {
            while(!firstStack.empty()) {
                secondStack.push(firstStack.top());
                firstStack.pop();
            }
            firstStack.push(a);
        }
    }
    void pop() {
        if(secondStack.empty())
            cout << "队列为空" << endl;
        else
            cout << secondStack.top() << endl;
            secondStack.pop();
    }
    int back() {
        if(secondStack.empty())
            cout << "队列为空" << endl;
        else
            cout << secondStack.top() << endl;
            return secondStack.top();
    }
    int size() {
        return secondStack.size();
    }
};

```

```

        }
        firstStack.push(a);
        while(!secondStack.empty()) {
            firstStack.push(secondStack.top());
            secondStack.pop();
        }
    }
}

void pop() {
    firstStack.pop();
}

private:
    stack<int> firstStack;
    stack<int> secondStack;
};

```

数组实现循环队列

```

enum ErrorCode
{
    success,
    underflow,
    overflow
};

const int maxQueue = 100;

template <class QueueEntry>
class MyQueue
{
public:
    MyQueue() {
        front = 0;
        rear = 0;
        for(int i = 0; i < maxQueue; i++) {
            entry[i] = 0;
        }
    }
    // 判断队列是否为空
    bool empty() const {
        return front == rear;
    }
    // 入队操作
    ErrorCode append(const QueueEntry &item) {
        if(full()) {
            return overflow;
        } else {
            entry[rear] = item;
            rear++;
            rear = rear % maxQueue;
            return success;
        }
    }
    // 出队操作
    ErrorCode serve() {
        if(empty()) {
            return underflow;
        }
    }
};

```

```

    } else {
        front++;
        front = (front + maxQueue) % maxQueue;
        return success;
    }
}

// 获取队头元素
ErrorCode retrieve(QueueEntry &item) const {
    if(empty()) {
        return underflow;
    } else {
        item = entry[front];
        return success;
    }
}

// 判断队列是否已满
bool full() const {
    return (rear + 1) % maxQueue == front;
}

// 获取队列已有元素个数
int size() const {
    return (rear + maxQueue - front) % maxQueue;
}

// 清除队列所有元素
void clear() {
    front = 0;
    rear = 0;
    for(int i = 0; i < maxQueue; i++) {
        entry[i] = 0;
    }
}

// 获取队头元素并出队
ErrorCode retrieve_and_serve(QueueEntry &item) {
    if(empty()) {
        return underflow;
    } else {
        item = entry[front];
        front++;
        front = (front + maxQueue) % maxQueue;
        return success;
    }
}

private:
    int front;// 队头下标
    int rear;// 队尾下标
    QueueEntry entry[100];// 队列容器
};

```

优先队列

链式队列

```
//front, rear save data
enum ErrorCode {
    success,
    underflow,
    overflow
};

template <class QueueEntry>
struct Node {
    QueueEntry data;
    Node<QueueEntry> *next;
};

template <class QueueEntry>
class MyQueue {
public:
    MyQueue() {
        front = NULL;
        rear = NULL;
    }
    // 判断队列是否为空
    bool empty() const {
        return front == NULL;//zhi xiang tong yi ge jie dian
    }
    // 入队操作
    ErrorCode append(const QueueEntry &item) {
        Node<QueueEntry>* tmp = new Node<QueueEntry>;
        if(tmp == NULL) {
            return overflow;
        }
        tmp -> data = item;
        tmp -> next = NULL;
        if(front == NULL) {
            front = rear = tmp;
        } else {
            rear -> next = tmp;
            rear = tmp;//
        }
        return success;
    }
    // 出队操作
    ErrorCode serve() {
        if(empty()) {
            return underflow;
        } else {
            Node<QueueEntry>* tmp = front;
            front = tmp -> next;
            delete tmp;
            tmp = NULL;
            return success;
        }
    }
    // 获取队头元素
    ErrorCode retrieve(QueueEntry &item) const {
        if(empty()) {
            return underflow;
        } else {
            item = front -> data;
            return success;
        }
    }
}
```

```

}

// 获取队列已有元素个数
int size() const {
    int size = 0;
    Node<QueueEntry>* tmp = front;
    while(tmp != NULL) {
        size++;
        tmp = tmp -> next;
    }
    return size;
}

// 清除队列所有元素
void clear() {
    while(front != NULL) {
        serve();
    }
}

// 获取队头元素并出队
ErrorCode retrieve_and_serve(QueueEntry &item) {
    if(empty()) {
        return underflow;
    } else {
        retrieve(item);
        serve();
        return success;
    }
}

private:
    // 请不要修改数据成员.
    Node<QueueEntry> *front;           // 队头指针
    Node<QueueEntry> *rear;            // 队尾指针
};


```

递归

全排列

非字典序输出

```

perm(int arr[], 0, size-1)
void perm(int arr[], int low, int high) {
    if (low == high) {
        for (int i = 1; i <= low; i++)
            cout << arr[i];
        printf("\n");
        return;
    }
    for (int j = low; j <= high; j++) { // 每個元素和第一個交換
        swap(arr[j],arr[low]);
        perm(arr,low+1,high); // 交換后得到子序列用perm得到子序列的全排列
        swap(arr[j],arr[low]);
    }
}

```

字典序

```
int p, n, ans[11];
bool vis[11];
void arrange(int time) {
    int i ;
    if (time==0) {
        for (i = 1; i <= n; i++)
            printf("%d",ans[i]);
        printf("\n");
        return;
    }
    for (int i = 1; i <= n; ++i) {
        if (!vis[i]) {
            vis[i] = true;
            ans[++p] = i;
            arrange(time-1);
            vis[i] = false;
            p--;
        }
    }
}
int main(int argc, char const *argv[])
{
    int t;
    cin >> t;
    while (t--) {
        cin >> n;
        p = 0;
        memset(vis,0,sizeof(vis));
        memset(ans,0,sizeof(ans));
        arrange(n);
    }
    return 0;
}
```

爬楼梯问题

```
#include <bits/stdc++.h>
using namespace std;
int d[50];
void r() {
    d[0] = 1;
    d[1] = 2;
    for (int i = 2; i < 50; i++)
        d[i] = d[i-1] + d[i-2];
}
int main() {
    int size;
    cin >> size;
    r();
    for (int i = 0; i < size; i++) {
        int louti;
        cin >> louti;
        cout << d[louti-1] << endl;
```

```
    }  
}
```

汉诺塔

```
void hanota(int n, int panA, int panB, int help) {  
    if (n == 1) {  
        printf("move disk 1 from peg %d to peg %d\n", panA, panB);  
    }  
    else {  
        hanota(n-1, panA, help, panB);  
        printf("move disk %d from peg %d to peg %d\n", n, panA, panB);  
        hanota(n-1, help, panB, panA);  
    }  
}
```

八皇后问题

Solution:根据教材与上课内容，主要是放好一个皇后之后，逐个尝试下一个位置，一直到底端。当当前操作无法进行时，返回上一步的操作位置，再进行尝试，一直到底端。

Output: 1 5 8 6 3 7 2 4 1 6 8 3 7 4 2 5 1 7 4 6 8 2 5 3 (and so on...)

```
#include <iostream>  
using namespace std;  
const int max_size = 8;  
class Queens{  
private:  
    int count;  
    bool queen_square[max_size][max_size];  
public:  
    Queens(int size);  
    bool is_solved() const;  
    void print() const;  
    bool unguarded(int col) const;  
    void insert(int col);  
    void remove(int col);  
    int board_size;  
};  
  
Queens::Queens(int size){  
    board_size = size;  
    count = 0;  
    for(int i = 0; i < board_size; i++){  
        for(int j = 0; j < board_size; j++){  
            queen_square[i][j] = false;  
        }  
    }  
}  
  
bool Queens::is_solved() const{  
    return board_size == count;
```

```

}

void Queens::print()const{
    for(int i = 0; i < max_size - 1; i++){
        for(int j = 0; j < max_size; j++){
            if(queen_square[i][j]){
                cout << j + 1 << " ";
            }
        }
    }
    for(int j = 0; j < max_size; j++){
        if(queen_square[max_size - 1][j]){
            cout << j + 1;
            break;
        }
    }
    cout << endl;
}

bool Queens::unguarded(int col)const{
    int i;
    bool flag = true;
    for(i = 0; flag && i < count; i++)
        flag = !queen_square[i][col];
    for(i = 0; flag && i <= count && i <= col; i++)
        flag = !queen_square[count - i][col - i];
    for(i = 0; flag && i <= count && col + i < board_size; i++)
        flag = !queen_square[count - i][col + i];
    return flag;
}

void Queens::insert(int col){
    queen_square[count][col] = true;
    count++;
}

void Queens::remove(int col){
    queen_square[count - 1][col] = false;
    count--;
}

void solve_from(Queens &queen){
    if(queen.is_solved())queen.print();
    else{
        for(int col = 0; col < queen.board_size; col++){
            if(queen.unguarded(col)){
                queen.insert(col);
                solve_from(queen);
                queen.remove(col);
            }
        }
    }
}

int main(){
    Queens queen(8);
    solve_from(queen);
}

```

搜索(数组必须为已排序)

Binary Search查找数据,返回下标

```
int BinarySearch(const int s[], const int size, const int target) {  
    int low = 0, high = size-1;  
    while (low < high) {  
        int mid = low+(high-low)/2;  
        if (s[mid] <= target) {  
            low = mid+1;  
        } //找到第一个不等的  
        else  
            high = mid;  
    }  
    if (low == 0 && s[low] != target)  
        return -1;  
    if (s[low] == target)  
        return low;  
    if (s[low-1] != target)  
        return -1;  
    return low-1;  
}
```

查找一个数据在数组中第一次出现,返回下标

```
int first(const int s[], const int size, const int target) {  
    int low = 0, high = size-1;  
    while (low < high) {  
        int mid = low+(high-low)/2;  
        if (s[mid] <= target) {  
            low = mid+1;  
        } //找到第一个不等的  
        else  
            high = mid;  
    }  
    if (low == 0 && s[low] != target)  
        return -1;  
    if (s[low] == target)  
        return low;  
    if (s[low-1] != target)  
        return -1;  
    return low-1;  
}
```

查找一个数据在数组中最后一次出现

```
int last(const int s[], const int size, const int target) { //x第一次出现  
    int low = 0, high = size-1;  
    while (low < high) {
```

```

        int mid = low+(high-low)/2;
        if (s[mid] >= target) {
            high = mid;
        }
        else
            low = mid+1;
    }
    return low;
}

```

插值查找

```

int InsertionSearch(int a[], int value, int low, int high)//需要查找的数据, 0, size-1
{
    int mid = low+(value-a[low])/(a[high]-a[low])*(high-low);
    if(a[mid]==value)
        return mid;
    if(a[mid]>value)
        return InsertionSearch(a, value, low, mid-1);
    if(a[mid]<value)
        return InsertionSearch(a, value, mid+1, high);
}

```

斐波那契查找

```

// 斐波那契查找.cpp
const int max_size=20;//斐波那契数组的长度

/*构造一个斐波那契数组*/
void Fibonacci(int * F)
{
    F[0]=0;
    F[1]=1;
    for(int i=2;i<max_size;++i)
        F[i]=F[i-1]+F[i-2];
}

/*定义斐波那契查找法*/
int FibonacciSearch(int *a, int n, int key) //a为要查找的数组,n为要查找的数组长度,key为要查找的关键字
{
    int low=0;
    int high=n-1;

    int F[max_size];
    Fibonacci(F);//构造一个斐波那契数组F

    int k=0;
    while(n>F[k]-1)//计算n位于斐波那契数列的位置
        ++k;

    int * temp;//将数组a扩展到F[k]-1的长度
    temp=new int [F[k]-1];
    memcpy(temp,a,n*sizeof(int));

```

```

for(int i=n;i<F[k]-1;++i)
    temp[i]=a[n-1];

while(low<=high)
{
    int mid=low+F[k-1]-1;
    if(key<temp[mid])
    {
        high=mid-1;
        k-=1;
    }
    else if(key>temp[mid])
    {
        low=mid+1;
        k-=2;
    }
    else
    {
        if(mid<n)
            return mid; //若相等则说明mid即为查找到的位置
        else
            return n-1; //若mid>=n则说明是扩展的数值,返回n-1
    }
}
delete [] temp;
return -1;
}

int main()
{
    int a[] = {0,16,24,35,47,59,62,73,88,99};
    int key=100;
    int index=FibonacciSearch(a,sizeof(a)/sizeof(int),key);
    cout<<key<<" is located at:"<<index;
    return 0;
}

```

排序

复杂度：快速排序=归并排序=堆排序>shell排序>选择排序=插入排序=冒泡排序(归并排序稳定)

快速排序

调用方法: QuickSort(arr,0,arr.size()-1)

```

void QuickSort(int arr[], int low, int high) {
    int i = low;
    int j = high;
    int temp = arr[i];
    if (low < high) {
        while (i < j) {
            while(arr[j] > temp && i < j)
                j--;
            arr[i] = arr[j];

```

```

        while(arr[i] < temp && i < j)
            i++;
        arr[j] = arr[i];
    }
    arr[i] = temp;
    QuickSort(arr,low,i-1);
    QuickSort(arr,j+1,high);
}
}

```

归并排序

调用方法: MergeSort(arr,0, arr.size()-1)

```

void Merge(int arr[], int low, int mid, int high) {
    int temp[high-low+1];
    int i = low, j = mid+1;
    int index = 0;
    while(i <= mid && j <= high) {
        temp[index++] = arr[i] < arr[j] ? arr[i++] : arr[j++];
    }
    while (i <= mid)
        temp[index++] = arr[i++];
    while (j <= high)
        temp[index++] = arr[j++];
    for (index = 0, i = low; i <= high; index++,i++)
        arr[i] = temp[index];
}

void MergeSort(int* arr, int low, int high) {
    int mid;
    if (low < high) {
        mid = (low+high)/2;
        MergeSort(arr,low,mid);
        MergeSort(arr,mid+1,high);
        Merge(arr,low,mid,high);
    }
}

```

堆排序

```

HeapSort(Arr,size)
void adjust(int *Arr, int size, int parent) {
    int left = parent*2+1;
    int right = parent*2+2;
    int max = parent;
    if (left < size && Arr[max] < Arr[left])
        max = left;
    if (right < size && Arr[max] < Arr[right])
        max = right;//三者中最大值
    if (max != parent) {
        int temp = Arr[parent];
        Arr[parent] = Arr[max];
        Arr[max] = temp;//将parent设为最大值
        adjust(Arr, size, max);
    }
}

```

```
}

void HeapSort(int *Arr, int size) {
    for (int i = size/2; i >= 0; i--)
        adjust(Arr, size, i);
    for (int i = size-1; i > 0; i--) {
        int temp = Arr[0];
        Arr[0] = Arr[i];
        Arr[i] = temp;
        adjust(Arr,i,0);
    }
}
```

shell

```
ShellSort(int *Arr, int size);
void ShellSort(int *Arr, int size) {
    for (int gap = size/2; gap >= 1; gap /= 2) {
        for (int i = 0; i < gap; i++) {
            for (int j = i+gap; j < size; j += gap) {//没有排序的区间的第一个
                int temp = Arr[j];
                int k = j-gap;//已排序的最后一个
                for(; k >= 0 && Arr[k] > temp; k -= gap)
                    Arr[k+gap] = Arr[k];
                Arr[k+gap] = temp;
            }
        }
    }
}
```

选择

```
SelectSort(int size, int arr[]);
void SelectSort(int size, int arr[]) {
    int index;
    for (int i = 0; i < size-1; i++) {
        index = i;
        for (int j = i+1; j < size; j++) {
            if (arr[j] < arr[index])
                index = j;//找到最小值
        }
        if (index != i) {
            int temp = arr[i];
            arr[i] = arr[index];
            arr[index] = temp;
        }
    }
}
```

插入排序

```

void InsertSort(int *Arr, int size) {
    for (int i = 1; i < size; i++) {
        int temp = Arr[i];
        int j = i-1;
        for (; j >= 0 && Arr[j] > temp; j--)
            Arr[j+1] = Arr[j];
        Arr[j+1] = temp;
    }
}

```

冒泡

```

for (int i = 0; i < N; i++) {
    for (int j = 0; j < N-i-1; j++) {
        if (arr[j] > arr[j+1]) {
            int temp = arr[j];
            arr[j] = arr[j+1];
            arr[j+1] = temp;
        }
    }
}

```

哈希表

哈希表

```

template <typename Key, typename Value>
HashTable<Key, Value>::HashTable(const HashTable & table){
    capacity = table.capacity;
    count = table.count;
    storage = new Pair[capacity];
    for(int i = 0; i < capacity; i++){
        storage[i].isEmpty = table.storage[i].isEmpty;
        if(!storage[i].isEmpty){
            storage[i].key = table.storage[i].key;
            storage[i].value = table.storage[i].value;
        }
    }
}

template <typename Key, typename Value>
HashTable<Key, Value> &HashTable<Key, Value> :: operator =(const HashTable & table){
    capacity = table.capacity;
    count = table.count;
    clear();
    storage = new Pair[capacity];
    for(int i = 0; i < capacity; i++){
        storage[i].isEmpty = table.storage[i].isEmpty;
        if(!storage[i].isEmpty){
            storage[i].key = table.storage[i].key;
            storage[i].value = table.storage[i].value;
        }
    }
}

```

```

        }
    }
    return *this;
}

template <typename Key, typename Value>
bool HashTable<Key, Value>::hasKey(const Key & key) const{
    for(int i = 0; i < capacity; i++){
        if(storage[i].key == key && !storage[i].isEmpty) return true;
    }
    return false;
}

template <typename Key, typename Value>
const Value & HashTable<Key, Value>::get(const Key & key) const{
    for(int i = 0; i < capacity; i++){
        if(storage[i].key == key)
            return storage[i].value;
    }
}

template <typename Key, typename Value>
void HashTable<Key, Value>::put(const Key & key, const Value & value){
    if(hasKey(key)){
        for(int i = 0; i < capacity; i++){
            if(!storage[i].isEmpty && storage[i].key == key){
                storage[i].value = value;
            }
        }
    }
    if(!hasKey(key)){
        if(count == capacity){
            Pair *temp = new Pair[capacity];
            for(int i = 0; i < capacity; i++){
                temp[i].isEmpty = storage[i].isEmpty;
                temp[i].key = storage[i].key;
                temp[i].value = storage[i].value;
            }
            capacity++;
            clear();
            for(int i = 0; i < capacity - 1; i++){
                put(temp[i].key, temp[i].value);
            }
            put(key, value);
            delete []temp;
        }
        if(count != capacity){
            for(int i = 0; i < capacity; i++){
                if(storage[i].isEmpty){
                    storage[i].isEmpty = false;
                    storage[i].key = key;
                    storage[i].value = value;
                    count++;
                    break;
                }
            }
        }
    }
}
}

```

简单哈希

```
#include <bits/stdc++.h>
using namespace std;

#define mod 13

struct hashNode{
    int data;
    hashNode * next;
    hashNode(int d) : data(d), next(NULL) {}
};

int main(int argc, char const *argv[])
{
    int n = 0, key = 0;
    hashNode * arr[mod];
    while(scanf("%d", &n) && n != 0) {
        for(int i = 0; i < mod; i++) {
            arr[i] = new hashNode(i);
        }
        for(int i = 0; i < n; i++) {
            cin >> key;
            hashNode * temp = arr[key % mod];
            if(arr[key % mod] -> next == NULL) {
                arr[key % mod] -> next = new hashNode(key);
            } else {
                while(temp -> next != NULL) {
                    temp = temp -> next;
                }
                temp -> next = new hashNode(key);
            }
        }
        for(int i = 0; i < mod; i++) {
            cout << i << "#";
            hashNode * temp = arr[i];
            if(temp -> next == NULL) {
                cout << "NULL";
            } else {
                while(temp -> next != NULL) {
                    temp = temp -> next;
                    cout << temp -> data;
                    if(temp -> next != NULL) {
                        cout << " ";
                    }
                }
            }
            delete temp;
            temp = NULL;
            cout << endl;
        }
    }
    return 0;
}
```

冲突解决方法

①开二维数组，②线性探测法③二次探测法，同②， $temp = temp * temp^*$

开二维数组

例：0, 26, 13的存储

```
#include <bits/stdc++.h>
using namespace std;
int main(int argc, char const *argv[])
{
    int size;
    while(cin >> size && size) {
        int HashTable[13][200] = {0};
        while (size--) {
            int num;
            scanf("%d", &num);
            int pos = ++HashTable[num%13][0];//第一个位置用来存储num%13的数量
            HashTable[num%13][pos] = num;
        }
        for (int i = 0; i < 13; i++) {
            printf("%d#", i);
            if (HashTable[i][0] == 0)
                printf("NULL\n");
            else {
                for (int j = 1; j <= HashTable[i][0]; j++) {
                    if (j == 1)
                        printf("%d", HashTable[i][j]);
                    else
                        printf(" %d", HashTable[i][j]);
                }
                printf("\n");
            }
        }
    }
    return 0;
}
```

线性探测法

```
#include <iostream>
#include <cstdio>
using namespace std;
int main(int argc, char const *argv[])
{
    int P, N;
    scanf("%d%d", &N, &P);
    int *arr = new int[N];
    int *HashTable = new int[P];
    for (int i = 0; i < P; i++)
        HashTable[i] = -1;
    for (int i = 0 ; i < N; i++) {
        scanf("%d", &arr[i]);
        int temp = arr[i]%P;
        while (HashTable[temp%P] != -1 && HashTable[temp%P] != arr[i])
            temp++;
    }
}
```

```

HashTable[temp%P] = arr[i];
if (i)
    printf(" %d", temp%P);
else
    printf("%d", temp%P);
}
printf("\n");
return 0;
}

```

二次探测(与线性探测一样，就是把temp++换成temp = temp*temp)

树

插入

```

pnode insert(pnode root, int x)
{
    pnode p = new pnode;
    p->val = x;
    p->lchild = NULL;
    p->rchild = NULL;
    if(root == NULL){
        root = p;
    }
    else if(x < root->val){
        root->lchild = insert(root->lchild, x);
    }
    else{
        root->rchild = insert(root->rchild, x);
    }
    return root;
}

```

查找

```

### 非递归版本
pnode search_BST(pnode p, int x)
{
    bool solve = false;
    while(p && !solve){
        if(x == p->val){
            solve = true;
        }
        else if(x < p->val){
            p = p->lchild;
        }
        else{
            p = p->rchild;
        }
    }
}

```

```

if(p == NULL){
    cout << "没有找到" << x << endl;
}
return p;
}

### 递归版本
AVLTreeNode<T>* AVLTree<T>::search_recurse(AVLTreeNode<T>* pnode, T key) const
{
    if (pnode != nullptr)
    {
        if (key == pnode->key)
            return pnode;
        if (key > pnode->key)
            return search_recurse(pnode->rchild, key);
        else
            return search_recurse(pnode->lchild, key);
    }
    return nullptr;
};

```

删除

```

bool delete_BST(pnode p, int x) //返回一个标志, 表示是否找到被删元素
{
    bool find = false;
    pnode q;
    p = BT;
    while(p && !find){ //寻找被删元素
        if(x == p->val){ //找到被删元素
            find = true;
        }
        else if(x < p->val){ //沿左子树找
            q = p;
            p = p->lchild;
        }
        else{ //沿右子树找
            q = p;
            p = p->rchild;
        }
    }
    if(p == NULL){ //没找到
        cout << "没有找到" << x << endl;
    }

    if(p->lchild == NULL && p->rchild == NULL){ //p为叶子节点
        if(p == BT){ //p为根节点
            BT = NULL;
        }
        else if(q->lchild == p){
            q->lchild = NULL;
        }
        else{
            q->rchild = NULL;
        }
        free(p); //释放节点p
    }
    else if(p->lchild == NULL || p->rchild == NULL){ //p为单支子树

```

```

if(p == BT){ //p为根节点
    if(p->lchild == NULL){
        BT = p->rchild;
    }
    else{
        BT = p->lchild;
    }
}
else{
    if(q->lchild == p && p->lchild){ //p是q的左子树且p有左子树
        q->lchild = p->lchild; //将p的左子树链接到q的左指针上
    }
    else if(q->lchild == p && p->rchild){
        q->lchild = p->rchild;
    }
    else if(q->rchild == p && p->lchild){
        q->rchild = p->lchild;
    }
    else{
        q->rchild = p->rchild;
    }
}
free(p);
}

else{ //p的左右子树均不为空
pnode t = p;
pnode s = p->lchild; //从p的左子节点开始
while(s->rchild){ //找到p的前驱, 即p左子树中值最大的节点
    t = s;
    s = s->rchild;
}
p->val = s->val; //把节点s的值赋给p
if(t == p){
    p->lchild = s->lchild;
}
else{
    t->rchild = s->lchild;
}
free(s);
}
return find;
}

```

二叉树的遍历

前序遍历

递归

```

void preOrder1(BinTree *root)      //递归前序遍历
{
    if(root!=NULL)
    {
        cout<<root->data<<" ";
        preOrder1(root->lchild);
    }
}

```

```

        preOrder1(root->rchild);
    }
}

```

非递归

对于任一结点P:

- 1)访问结点P，并将结点P入栈；
- 2)判断结点P的左孩子是否为空，若为空，则取栈顶结点并进行出栈操作，并将栈顶结点的右孩子置为当前的结点P，循环至1)；若不为空，则将P的左孩子置为当前的结点P；
- 3)直到P为NULL并且栈为空，则遍历结束。

```

void preOrder2(BinTree *root)      //非递归前序遍历
{
    stack<BinTree*> s;
    BinTree *p=root;
    while(p!=NULL || !s.empty())
    {
        while(p!=NULL)
        {
            cout<<p->data<<" ";
            s.push(p);
            p=p->lchild;
        }
        if(!s.empty())
        {
            p=s.top();
            s.pop();
            p=p->rchild;
        }
    }
}

```

中序遍历

递归

```

void inOrder1(BinTree *root)      //递归中序遍历
{
    if(root!=NULL)
    {
        inOrder1(root->lchild);
        cout<<root->data<<" ";
        inOrder1(root->rchild);
    }
}

```

非递归

对于任一结点P,

- 1)若其左孩子不为空，则将P入栈并将P的左孩子置为当前的P，然后对当前结点P再进行相同的处理；

- 2)若其左孩子为空，则取栈顶元素并进行出栈操作，访问该栈顶结点，然后将当前的P置为栈顶结点的右孩子；
 3)直到P为NULL并且栈为空则遍历结束

```
void inOrder2(BinTree *root)      //非递归中序遍历
{
    stack<BinTree*> s;
    BinTree *p=root;
    while(p!=NULL || !s.empty())
    {
        while(p!=NULL)
        {
            s.push(p);
            p=p->lchild;
        }
        if(!s.empty())
        {
            p=s.top();
            cout<<p->data<<" ";
            s.pop();
            p=p->rchild;
        }
    }
}
```

后序遍历

递归

```
void postOrder1(BinTree *root)    //递归后序遍历
{
    if(root!=NULL)
    {
        postOrder1(root->lchild);
        postOrder1(root->rchild);
        cout<<root->data<<" ";
    }
}
```

非递归

第一种思路：

对于任一结点P，将其入栈，然后沿其左子树一直往下搜索，直到搜索到没有左孩子的结点，此时该结点出现在栈顶，但是此时不能将其出栈并访问，因此其右孩子还未被访问。所以接下来按照相同的规则对其右子树进行相同的处理，当访问完其右孩子时，该结点又出现在栈顶，此时可以将其出栈并访问。这样就保证了正确的访问顺序。可以看出，在这个过程中，每个结点都两次出现在栈顶，只有在第二次出现在栈顶时，才能访问它。因此需要多设置一个变量标识该结点是否是第一次出现在栈顶。

```
void postOrder2(BinTree *root)    //非递归后序遍历
{
    stack<BTNode*> s;
    BinTree *p=root;
```

```

BTNode *temp;
while(p!=NULL||!s.empty())
{
    while(p!=NULL)                                //沿左子树一直往下搜索，直至出现没有左子树的结点
    {
        BTNode *btn=(BTNode *)malloc(sizeof(BTNode));
        btn->btnode=p;
        btn->isFirst=true;
        s.push(btn);
        p=p->lchild;
    }
    if(!s.empty())
    {
        temp=s.top();
        s.pop();
        if(temp->isFirst==true)          //表示是第一次出现在栈顶
        {
            temp->isFirst=false;
            s.push(temp);
            p=temp->btnode->rchild;
        }
        else                            //第二次出现在栈顶
        {
            cout<<temp->btnode->data<<" ";
            p=NULL;
        }
    }
}
}

```

第二种思路：

要保证根结点在左孩子和右孩子访问之后才能访问，因此对于任一结点P，先将其入栈。如果P不存在左孩子和右孩子，则可以直接访问它；或者P存在左孩子或者右孩子，但是其左孩子和右孩子都已被访问过了，则同样可以直接访问该结点。若非上述两种情况，则将P的右孩子和左孩子依次入栈，这样就保证了每次取栈顶元素的时候，左孩子在右孩子前面被访问，左孩子和右孩子都在根结点前面被访问。

```

void postOrder3(BinTree *root)      //非递归后序遍历
{
    stack<BinTree*> s;
    BinTree *cur;           //当前结点
    BinTree *pre=NULL;      //前一次访问的结点
    s.push(root);
    while(!s.empty())
    {
        cur=s.top();
        if((cur->lchild==NULL&&cur->rchild==NULL)|||  

            (pre!=NULL&&(pre==cur->lchild||pre==cur->rchild)))
        {
            cout<<cur->data<< " "; //如果当前结点没有孩子结点或者孩子节点都已被访问过
            s.pop();
            pre=cur;
        }
        else
        {
            if(cur->rchild!=NULL)
                s.push(cur->rchild);
            if(cur->lchild!=NULL)

```

```

        s.push(cur->lchild);
    }
}
}

```

二叉树层次遍历

核心思想：

首先，将二叉树的最祖先节点入队列

然后，循环执行以下步骤，知道队列空。

一：节点出队列

二：该节点如果有左孩子节点，左孩子节点入队列

三：该节点如果有右孩子节点，右孩子节点入队列

```

void levelTraversal(BinaryNode<T>* root, void (*visit)(T &x)) {
    if(root == NULL) {
        return;
    }
    queue<BinaryNode<T>*> treeQueue;
    treeQueue.push(root);
    while(!treeQueue.empty()) {
        if(treeQueue.front() -> left) {
            treeQueue.push(treeQueue.front() -> left);
        }
        if(treeQueue.front() -> right) {
            treeQueue.push(treeQueue.front() -> right);
        }
        visit(treeQueue.front() -> elem);
        treeQueue.pop();
    }
}

```

创建树

中序和后序建树

```

TreeNode * build(vector<int>& inorder,int ileft,int iright,vector<int>& postorder,int
pleft,int pright)
{
    if(pleft>pright || ileft>iright) return NULL;
    int i,j,k;
    for(i=ileft;i<=iright;i++) if(postorder[pright]==inorder[i]) break;
    TreeNode * node=new TreeNode(postorder[pright]);
    int size=i-ileft;
    node->left=build(inorder,ileft,i-1,postorder,pleft,pleft+size-1);
    node->right=build(inorder,i+1,iright,postorder,pleft+size,pright-1);
    return node;
}

```

```

TreeNode* buildTree(vector<int>& inorder, vector<int>& postorder) {
    return build(inorder, 0, inorder.size() - 1, postorder, 0, postorder.size() - 1);
}

```

先序后序建树

```

TreeNode * build(vector<int>& preorder, int pleft, int pright, vector<int>& inorder, int
ileft, int iright)
{
    if(pleft>pright || ileft>iright) return NULL;
    int i,j,k;
    for(i=ileft;i<=iright;i++) if(preorder[pleft]==inorder[i]) break;
    TreeNode * node=new TreeNode(preorder[pleft]);
    int size=i-ileft;
    node->left=build(preorder, pleft+1, pleft+size, inorder, ileft, i-1);
    node->right=build(preorder, pleft+size+1, pright, inorder, i+1, iright);
    return node;
}
TreeNode* buildTree(vector<int>& preorder, vector<int>& inorder) {
    return build(preorder, 0, preorder.size() - 1, inorder, 0, inorder.size() - 1);
}

```

二叉树层序输入建树

递归版

```

void mid_order(ptree &root)
{
    if (root != NULL)
    {
        mid_order(root->lchild);
        printf ("%c ",root->data);
        mid_order(root->rchild);
    }
    else
        return;
}

```

非递归

```

void buildTree(BN** rootptr) { //当num == -1, 表示该节点为空
    int num;
    BN* temp[40] = {0}; //temp数组用来暂时存储节点
    int index = 1, head = 1;
    scanf ("%d", &num);
    if (num != -1)
        *rootptr = (BN*)calloc(1, sizeof(BN));
    else
        (*rootptr) = NULL;
    temp[1] = *rootptr;
    temp[1]->x = num;
    temp[1]->left = NULL;
    temp[1]->right = NULL;
}

```

```

while(num != -1) {
    scanf("%d",&num);
    if (num != -1) {
        index++;
        BN* _left = (BN*)calloc(1,sizeof(BN));
        _left->x = num;
        _left->left = NULL;
        _left->right = NULL;
        temp[head]->left = _left;
        temp[index] = _left;
        scanf("%d",&num); //一次while循环输入两个num
    }
    if (num != -1) {
        index++;
        BN* _right = (BN*)calloc(1,sizeof(BN));
        _right->x = num;
        _right->right = NULL;
        _right->left = NULL;
        temp[index] = _right;
        temp[head]->right = _right;
    }
    head++;
}
}

```

二叉树的销毁

```

template<typename T>
void AVLTree<T>::destory(AVLTreeNode<T>* & pnode)
{
    if (pnode != nullptr)
    {
        destory(pnode->lchild); //递归销毁左子树
        destory(pnode->rchild); //递归销毁右子树
        delete pnode; //销毁根节点
        pnode = nullptr;
    }
};

template<typename T>
void AVLTree<T>::destory()
{
    destory(root);
}

```

二叉树综合

```

//.hpp
#ifndef BT_TREE
#define BT_TREE
#include <iostream>
using namespace std;
struct node {
    int ele;

```

```

node* left;
node* right;
node(int e):left(0), right(0){
    ele = e;
}
};

class BinaryTree {
private:
    node* root;
    // 此处四个函数均为辅助函数
    static void MemoryDelete(node* p); // 内存处理的辅助函数
    static void BuildTree(const node* Source_Root, node* &Target_Root); // 拷贝构造函数的辅助函数
    static void BuildTree(const int* arr,int len, node* &root); // 数组构建树的辅助函数
    static void preorder(const node* p); // 输出的辅助函数
public:
    BinaryTree();
    BinaryTree(const BinaryTree&);
    BinaryTree(const int* arr, int len);
    void ResetTree(const int* arr, int len); // 清空当前树并重置
    ~BinaryTree();
    void clear(); // 清空
    void insert(int ele); // 按值插入
    void Delete(int ele); // 暗值删除
    void print();
};

#endif

```

```

#include "BinaryTree.hpp"

void BinaryTree::preorder(const node* p) {
    if (p == NULL) return;
    cout << p->ele << ' ';
    preorder(p->left);
    preorder(p->right);
}

void BinaryTree::ResetTree(const int* arr, int len) {
    clear();
    BuildTree(arr, len, root);
}

void BinaryTree::print() {
    preorder(root);
    cout << endl;
}

BinaryTree::BinaryTree() {
    root = NULL;
}

void BinaryTree::insert(int ele) {
    node* p = root;
    if (p == NULL) {
        root = new node(ele);
        return;
    }
    while (p->ele != ele) {
        if (ele > p->ele) {

```

```

    if (p->right) {
        p = p->right;
    } else {
        p->right = new node(ele);
        break;
    }
} else {
    if (p->left) {
        p = p->left;
    } else {
        p->left = new node(ele);
        break;
    }
}
}

void BinaryTree::Delete(int ele) {
    node* p = root, *q = NULL;
    bool flag = true; // if flag == true, p is the left child of q.
    while (p && p->ele != ele) {
        q = p;
        if (ele > p->ele) {
            p = p->right;
            flag = false;
        } else {
            p = p->left;
            flag = true;
        }
    }
    if (p) {
        if (p->left == NULL) {
            if (q == NULL) root = p->right;
            else if (flag) q->left = p->right;
            else q->right = p->right;
        } else {
            node* tmp = p->left;
            node* tmp_par = p;
            while (tmp->right) {
                tmp_par = tmp;
                tmp = tmp->right;
            }
            tmp->right = p->right;
            tmp_par->right = tmp->left;
            if (p->left != tmp)
                tmp->left = p->left;
            if (q == NULL) root = tmp;
            else if (flag) q->left = tmp;
            else q->right = tmp;
        }
        delete p;
    }
}
BinaryTree::BinaryTree(const BinaryTree& oth) {
    BuildTree(oth.root, this->root);
}
BinaryTree::~BinaryTree() {
    clear();
}
void BinaryTree::clear() {
    MemoryDelete(root);
}

```

```

root = NULL;
}
void BinaryTree::MemoryDelete(node* t) {
    if (t == NULL) return;
    MemoryDelete(t->left);
    MemoryDelete(t->right);
    delete t;
}
BinaryTree::BinaryTree(const int* arr, int len) {
    BuildTree(arr, len, root);
}
void BinaryTree::BuildTree(const node* Source_Root, node* &Target_Root) {
    if (Source_Root == NULL) {
        Target_Root = NULL;
        return;
    }
    Target_Root = new node(Source_Root->ele);
    BuildTree(Source_Root->left, Target_Root->left);
    BuildTree(Source_Root->right, Target_Root->right);
}

void BinaryTree::BuildTree(const int* arr, int len, node* &root) {
    int index = 0;
    root = NULL;
    while (len--) {
        if (root == NULL) {
            root = new node(arr[index++]);
            continue;
        }
        node* p = root;
        int tmp = arr[index++];
        while (p->ele != tmp) {
            if (tmp > p->ele) {
                if (p->right) {
                    p = p->right;
                } else {
                    p->right = new node(tmp);
                    break;
                }
            } else {
                if (p->left) {
                    p = p->left;
                } else {
                    p->left = new node(tmp);
                    break;
                }
            }
        }
    }
}
}

```

查找二叉树中最大最小值

```

template <typename T>
AVLTreeNode<T>* AVLTree<T>::maximum(AVLTreeNode<T>* pnode) const
{
    if (pnode != nullptr)
    {

```

```

        while (pnode->rchild != nullptr)
            pnode = pnode->rchild;
        return pnode;
    }
    return nullptr;
};

template<typename T>
T AVLTree<T>::maximum()
{
    AVLTreeNode<T>* presult = maximum(root);
    if (presult != nullptr)
        return presult->key;
};

/*返回树中最小节点值*/
template <typename T>
AVLTreeNode<T>* AVLTree<T>::minimum(AVLTreeNode<T>* pnode) const
{
    if (pnode != nullptr)
    {
        while (pnode->lchild != nullptr)
            pnode = pnode->lchild;
        return pnode;
    }
    return nullptr;
};

template<typename T>
T AVLTree<T>::minimum()
{
    AVLTreeNode<T>* presult = minimum(root);
    if (presult != nullptr)
        return presult->key;
};

```

求树的高度

```

int height(node* root) {
    if (root == NULL)
        return 0;
    else {
        int L = height(root->left);
        int R = height(root->right);
        return (L > R ? L : R)+1;
    }
}

```

二叉树中某一条路径节点数之和最大

```

int max_sum;
void find_max_path(tree* root, int sum) {
    if (root) {
        sum += root->val;

```

```

    if (sum > max_sum)
        max_sum = sum;
    find_max_path(root->left, sum);
    find_max_path(root->right, sum);
    sum -= root->val;
}
}

```

求二叉树的宽度

```

int width(const treeNode * root){
    if(root == NULL) return 0;
    queue<const treeNode *> my;
    my.push(root);
    int max = 1;
    while(true){
        int num = my.size();
        if(num == 0)break;
        while(num){
            const treeNode * temp = my.front();
            my.pop();
            num--;
            if(temp->left)my.push(temp->left);
            if(temp->right)my.push(temp->right);
        }
        max = (max > my.size()) ? max : my.size();
    }
    return max;
}

```

二叉树最大宽度

此处宽度为

```

    4
    4
4 NULL NULL 4宽度为4

```

```

public class Solution
{
    public int widthOfBinaryTree(TreeNode root)
    {
        if(root == null)
            return 0;
        Queue<TreeNode> queue = new LinkedList<>(); //保存当前层的树节点
        Queue<Integer> queuePos = new LinkedList<>(); //保存上面队列中树节点对应的位置标号
        queue.add(root);
        queuePos.add(1); //在顶层根节点位置为1

        int countCurrent = 1; //记录当前正在遍历层的剩余数量
        int countTmp = 0; //记录下一层的节点数量
        int max = countCurrent;
        int start = 1;
        int end = 1; //每一层中最大孩子的位置下标
        while(!queue.isEmpty()) //首先判断此队列是否为空

```

```

{
    TreeNode current = queue.poll();
    end = queuePos.poll();
    if(current.left != null)
    {
        queue.add(current.left);
        queuePos.add(2*end);      //左孩子的下标
        countTmp++;
    }
    if(current.right != null)
    {
        queue.add(current.right);
        queuePos.add(2*end+1);   //右孩子的下标
        countTmp++;
    }
    if(--countCurrent == 0)    //判断是否将当前这一层的所有节点都遍历完
    {
        max = max > (end - start + 1) ? max : (end - start + 1);    //start要么为1，要么就为下一层中开头的那个节点的下标
        countCurrent = countTmp;    //这里将新的下一层的节点个数赋给这个变量
        countTmp = 0;              //将下一层的节点计数赋值为0
        start = queuePos.isEmpty() ? 1 : queuePos.peek();           //此start值为下一层开头的第一个节点的下标
    }
    return max;
}

```

求一颗树中有多少个叶子节点

```

int leaves(const BinaryNode* root){
    int count = 0;
    if(root == NULL) return 0;
    else if(root->left == NULL && root->right == NULL) return 1;
    else{
        count = leaves(root->left) + leaves(root->right);
    }
    return count;
}

```

哈夫曼编码

```

#include<stdio.h>
#include<stdlib.h>
#include<string.h>
#include<queue>
using namespace std;

typedef struct node{
    char ch;                      //存储该节点表示的字符，只有叶子节点用的到
    int val;                      //记录该节点的权值
    struct node *self,*left,*right; //三个指针，分别用于记录自己的地址，左孩子的地址和右孩子的地址
    friend bool operator <(const node &a,const node &b) //运算符重载，定义优先队列的比较结构
};

```

```

{
    return a.val>b.val;           //这里是权值小的优先出队列
}
}node;

priority_queue<node> p;          //定义优先队列
char res[30];                    //用于记录哈夫曼编码
void dfs(node *root,int level)   //打印字符和对应的哈夫曼编码
{
    if(root->left==root->right) //叶子节点的左孩子地址一定等于右孩子地址，且一定都为NULL；叶子节点记录有字符
    {
        if(level==0)           // "AAAAA" 这种只有一字符的情况
        {
            res[0]='0';
            level++;
        }
        res[level]='\0';        //字符数组以'\0'结束
        printf("%c=>%s\n",root->ch,res);
    }
    else
    {
        res[level]='0';        //左分支为0
        dfs(root->left,level+1);
        res[level]='1';        //右分支为1
        dfs(root->right,level+1);
    }
}
void huffman(int *hash)           //构建哈夫曼树,hash为哈希映射
{
    node *root,fir,sec;
    for(int i=0;i<26;i++)       //程序只能处理全为大写英文字母的信息串，故哈希也只有26个
    {
        if(!hash[i])           //对应字母在text中未出现
            continue;
        root=(node *)malloc(sizeof(node)); //开辟节点
        root->self=root;           //记录自己的地址，方便父节点连接自己
        root->left=root->right=NULL; //该节点是叶子节点，左右孩子地址均为NULL
        root->ch='A'+i;           //记录该节点表示的字符
        root->val=hash[i];         //记录该字符的权值
        p.push(*root);             //将该节点压入优先队列
    }
    //下面循环模拟建树过程，每次取出两个最小的节点合并后重新压入队列
    //当队列中剩余节点数量为1时，哈夫曼树构建完成
    while(p.size()>1)
    {
        fir=p.top();p.pop();    //取出最小的节点
        sec=p.top();p.pop();    //取出次小的节点
        root=(node *)malloc(sizeof(node)); //构建新节点，将其作为fir, sec的父节点
        root->self=root;         //记录自己的地址，方便该节点的父节点连接
        root->left=fir.self;    //记录左孩子节点地址
        root->right=sec.self;   //记录右孩子节点地址
        root->val=fir.val+sec.val;//该节点权值为两孩子权值之和
        p.push(*root);           //将新节点压入队列
    }
    fir=p.top();p.pop();        //弹出哈夫曼树的根节点
    dfs(fir.self,0);           //输出叶子节点记录的字符和对应的哈夫曼编码
}
int main()

```

```

{
    char text[100];
    int hash[30];
    memset(hash, 0, sizeof(hash)); //哈希数组初始化全为0
    scanf("%s", text); //读入信息串text
    for(int i=0; text[i]!='\0'; i++) //通过哈希求每个字符的出现次数
    {
        hash[text[i]-'A']++; //程序假设运行的全为英文大写字母
    }
    huffman(hash);
    return 0;
}

```

最小生成树

应用：要在n个城市之间铺设光缆，主要目标是要使这 n 个城市的任意两个之间都可以通信，但铺设光缆的费用很高，且各个城市之间铺设光缆的费用不同，因此另一个目标是要使铺设光缆的总费用最低

prim算法

```

struct node {
    int v, len;
    node(int v, int len = 0) : v(v), len(len) {}
    bool operator < (const node& a) const {
        return len > a.len;
    }
};

std::vector<node> G[maxn]; //记录所有距离
int vis[maxn];
int dis[maxn];

void init() {
    for (int i = 0; i < maxn; i++) {
        G[i].clear();
        dis[i] = INF; //记录与当前集合里节点距离的最小值
        vis[i] = false;
    }
}

int Prim(int s) {
    priority_queue<node> Q;
    int ans = 0;
    Q.push(node(s, 0));
    while (!Q.empty()) {
        node now = Q.top();
        Q.pop();
        int v = now.v;
        if (vis[v])
            continue;
        vis[v] = true;
        ans += now.len;
        for (int i = 0; i < G[v].size(); i++) {
            int v2 = G[v][i].v;
            int len = G[v][i].len;
            if (!vis[v2] && dis[v2] > len) {

```

```

        dis[v2] = len;
        Q.push(node(v2,dis[v2]));
    }
}
return ans;
}

```

krusal算法

```

//每次都是加最小边
struct edge
{
    int u,v;
    int cost;
    edge(int x, int y, int c) : u(x), v(y), cost(c) {};
};

bool cmp(edge a, edge b) {
    return a.cost < b.cost;
}

int findFather(int f[], int k) {
    int temp = k;
    while (k != f[k]) {
        k = f[k];//找到根节点
    }
    while (a != f[a]) {
        int z = a;
        a = f[a];
        f[z] = k;//统一将该集合的父亲定为根节点
    }
}

int Krusal(int m, int n, vector<edge> E) {//m为边数, n为顶点数
    for (int i = 0; i < n; i++)
        f[i] = i;//初始化集合
    sort(E.begin(), E.end(), cmp);
    for (int i = 0; i < m; i++) {
        int Fu = findFather(E[i].u);
        int Fv = findFather(E[i].v);
        if (Fu != Fv) {//是否为同一个集合
            f[Fu] = Fv;
            total_cost += cost;
            edge++;
            if (edge == n-1)
                break;
        }
    }
    if (edge != n-1)
        return -1;
    else
        return total_cost;
}

```

AVL树(BF是平衡因子)

判断是否为AVL树

```
template <class Entry>
struct AvlNode{
    Entry entry;
    AvlNode<Entry> *left;
    AvlNode<Entry> *right;
    int bf;// balance factor 均衡因子
};

template <class Entry>
bool judge(AvlNode<Entry> *root, int *n){
    if(root == NULL){
        *n=0;
        return true;
    }
    int left, right;
    if(judge(root->left, &left) && judge(root->right, &right)){
        int cha = left - right;
        if(cha <=1 && cha >= -1){
            *n = 1 + (left > right ? left : right);
            return true;
        }
    }
    return false;
}

template <class Entry>
bool is_AVL_Tree(AvlNode<Entry> *&root){
    int n = 0;
    return judge(root, &n);
}
```

左旋

```
template<typename T>
AVLTreeNode<T>* AVLTree<T>::leftRotation(AVLTreeNode<T>* proot)
{
    AVLTreeNode<T>* prchild = proot->rchild;
    proot->rchild = prchild->lchild;
    prchild->lchild = proot;

    proot->height = max(height(proot->lchild),height(proot->rchild))+1; //更新节点的高度值
    prchild->height = max(height(prchild->lchild), height(prchild->rchild)) + 1; //更新节点的高度值

    return prchild;
};
```

右旋

```
template <typename T>
AVLTreeNode<T>* AVLTree<T>::rightRotation(AVLTreeNode<T>*proot)
```

```

{
    AVLTreeNode<T>* plchild = proot->lchild;
    proot->lchild = plchild->rchild;
    plchild->rchild = proot;

    proot->height = max(height(proot->lchild), height(proot->rchild)) + 1;      //更新节点的高度值
    plchild->height = max(height(plchild->lchild), height(plchild->rchild)) + 1; //更新节点的高度值

    return plchild;
};

```

先右旋后左旋

```

template<typename T>
AVLTreeNode<T>* AVLTree<T>::rightLeftRotation(AVLTreeNode<T>* proot)
{
    proot->rchild = rightRotation(proot->rchild);
    return leftRotation(proot);
};

```

先左旋后右旋

```

template <typename T>
AVLTreeNode<T>* AVLTree<T>::leftRightRotation(AVLTreeNode<T>* proot)
{
    proot->lchild= leftRotation(proot->lchild);
    return rightRotation(proot);
};

```

堆

堆结构定义(以下均为最大堆(即为大顶堆), 最小堆和最大堆一样, 只是将>改为<)

```

struct MaxHeap
{
    Etype *heap;
    int HeapSize;
    int MaxSize;
};

MaxHeap H;

```

最大堆初始化

```

void MaxHeapInit (MaxHeap &H)//最大堆从Heap[1]开始存放数据，Heap[0]只是用来暂时存储数据，在调用
MaxHeapInit 前，应该将MaxHeap初始化，记录数据
{
    for(int i = H.HeapSize/2; i>=1; i--)
    {
        H.heap[0] = H.heap[i];
        int son = i*2;
        while(son <= H.HeapSize)
        {
            if(son < H.HeapSize && H.heap[son] < H.heap[son+1])
                son++;
            if(H.heap[0] >= H.heap[son])
                break;
            else
            {
                H.heap[son/2] = H.heap[son];
                son *= 2;
            }
        }
        H.heap[son/2] = H.heap[0];
    }
}

```

插入节点

```

void MaxHeapInsert (MaxHeap &H, EType &x)
{
    if(H.HeapSize == H.MaxValue)
        return false;
    int i = ++H.HeapSize;
    while(i!=1 && x>H.heap[i/2])
    {
        H.heap[i] = H.heap[i/2];
        i = i/2;
    }
    H.heap[i] = x;
    return true;
}

```

删除节点

```

void MaxHeapDelete (MaxHeap &H, EType &x)
{
    if(H.HeapSize == 0)
        return false;
    x = H.heap[1];
    H.heap[0] = H.heap[H.HeapSize--];
    int i = 1, son = i*2;

    while(son <= H.HeapSize)
    {
        if(son <= H.HeapSize && H.heap[0] < H.heap[son+1])
            son++;
    }
}

```

```

        if(H.heap[0] >= H.heap[son])
            break;
        H.heap[i] = H.heap[son];
        i = son;
        son = son*2;
    }
H.heap[i] = H.heap[0];
return true;
}

```



拓扑排序

拓扑排序应用于子任务的调动，例如任务A、B、C、D....Z，A必须在B前完成(此时B的indegree[B])，称为B依赖与A，D必须在C前完成(此时C的indegree[C])，而C又依赖于A，求这个任务怎么安排，依赖关系(大学课程安排中，学习数据结构必须先学习C语言，则数据结构依赖于C语言，A的奖金必须比B的高，B的比C的高...)

```

#include <iostream>
#include <cstdio>
#include <queue>
#include <vector>
using namespace std;

int main(int argc, char const *argv[])
{
    vector<int> zirenwu[110];
    int indegree[110] = {0};
    queue<int> q;//q存储没有依赖关系了的节点，即可以完成的任务
    int size;
    scanf("%d", &size);
    for (int i = 1; i <= size; i++) {
        scanf("%d", &indegree[i]); //表示一个任务i需要依赖于多少个别的任务
        for (int j = 0; j < indegree[i]; j++) {
            int temp;
            scanf("%d", &temp);
            zirenwu[temp].push_back(i); //i依赖于temp
        }
    }
    for (int i = 1; i <= size; i++) {
        if (indegree[i] == 0) //将没有依赖关系的放进队列
            q.push(i);
    }
    int count = 0;
    while (!q.empty()) {
        count++;
        int temp = q.front(); //temp任务已经完成
        q.pop();
        for (int i = 0; i < zirenwu[temp].size(); i++) {
            int ver = zirenwu[temp][i];
            if (--indegree[ver] == 0) //由于temp已经完成，所以ver的依赖的子任务可以减少1
                q.push(ver);
        }
    }
}

```

```

    }
    if (count == size)
        printf("1\n");
    else
        printf("0\n");
    return 0;
}

```

核心代码

```

queue<int> q;
for (int i = 0; i < n; i++) {
    if (indegree[i] == 0)
        q.push(i);
}
while (q.size()) {
    int temp = q.front();
    q.pop();
    for (int i = 0; i < n; i++) {
        if (matrix[temp][i]) {
            indegree[i]--;
            if (indegree[i] == 0)
                q.push(i);
        }
    }
}

```

欧拉回路的判断

```

#include <iostream>
#include <cstring>
using namespace std;
int map[1100][1100];
bool isVisited[1100];
void dfs(int size, int ver);
bool judge(int size, int ver);
int main(int argc, char const *argv[])
{
    memset(isVisited, false, sizeof(isVisited));
    int size, side, ver;
    scanf("%d%d", &size, &side);
    for (int i = 1; i <= size; i++) {
        for (int j = 1; j <= size; j++)
            map[i][j] = 0;
    }
    for (int i = 0; i < side; i++) {
        int v1, v2;
        scanf("%d%d", &v1, &v2);
        map[v1][v2] = 1;
        map[v2][v1] = 1;
        ver = v1;
    }
    int flag = 1;
    dfs(size, 1);
    for (int i = 1; i <= size; i++) {
        if (!isVisited[i] || judge(size, i)) {

```

```

        flag = 0;
        break;
    }
}
printf("%d\n", flag);
return 0;
}

void dfs(int size, int ver) {
    isVisited[ver] = true;
    for (int i = 1; i <= size; i++) {
        if (map[ver][i] && !isVisited[i])
            dfs(size, i);
    }
}

bool judge(int size, int ver) { // 欧拉回路的度必定为偶数
    int count = 0;
    for (int i = 1; i <= size; i++)
        if (map[ver][i])
            count++;
    return count % 2;
}

```

最短路径

Dijkstra算法 一个节点到其他所有节点最短路径，权不可以为负值，无向图，比floyd快

算法原理：

算法步骤如下：

$G = \{V, E\}$

1. 初始时令 $S = \{v_0\}$, $T = V - S = \{\text{其余顶点}\}$, T 中顶点对应的距离值

若存在 $\langle v_0, v_i \rangle$, $d(v_0, v_i)$ 为 $\langle v_0, v_i \rangle$ 弧上的权值

若不存在 $\langle v_0, v_i \rangle$, $d(v_0, v_i)$ 为 ∞

2. 从 T 中选取一个与 S 中顶点有关联边且权值最小的顶点 w , 加入到 S 中

3. 对其余 T 中顶点的距离值进行修改：若加进 w 作中间顶点，从 v_0 到 v_i 的距离值缩短，则修改此距离值

重复上述步骤 2、3，直到 S 中包含所有顶点，即 $w = v_i$ 为止

使用优先队列版本

```

#define INF 999999
struct node {
    int v, len;
    node(int a, int b) : v(a), len(b){};
    bool operator<(const node& a) const {
        return len > a.len;
    }
};

vector<node> G[max];
bool vis[max];
int dis[max]; // dis 存储集合 T 中各点到 S 中最短路径

```

```

void init() {
    for (int i = 0; i < max; i++) {
        G[i].clear();
        vis[i] = false;
        dis[max] = INF;
    }
}

int dijkstra(int s, int e) { //广度优先
    priority_queue<node> Q;
    Q.push(node(s, 0));
    dis[s] = 0;
    while (!Q.empty()) {
        node now = Q.top();
        Q.pop();
        int v = now.v;
        if (vis[v])
            continue;
        vis[v] = true;
        for (int i = 0; i < G.size(); i++) {
            int v2 = G[v][i].v;
            int len = G[v][i].len;
            if (!vis[v2] && dis[v2] > dis[v]+len) {
                dis[v2] = dis[v]+len;
                Q.push(node(v2, dis[v2]));
            }
        }
    }
    return dis[e];
}

```

不使用优先队列

```

#define INF 100000000
#define maxn 1001
bool vis[maxn];
int adj[maxn][maxn],dis[maxn],pre[maxn];//pre[]记录前驱, adj应该初始化为INF
int n, m;
void dijkstra(int v)
{
    int i, j, u , min;
    for(i=0;i<=n;i++)
    {
        dis[i]=adj[v][i];
        vis[i]=0;
    }
    vis[v]=1;dis[v]=0;
    for(i=1;i<n;i++)
    {
        min = INF;
        for(j=1;j<=n;j++)
        {
            if(!vis[j]&&min > dis[j])
            {
                min = dis[j];
                u = j;
            }
        }

```

```

        }
        if(min == INF)break;
        vis[u]=1;
        for(j=1;j<=n;j++)
        {
            if(!vis[j]&&adj[u][j]!=INF&&dis[u]+adj[u][j]<dis[j])
            {
                dis[j] = adj[u][j] + dis[u];
            }
        }
    }
}

```

floyd-warshall 任意两点间的最短路径，可以解决有向图或者负权问题

```

for (int i = 1; i <= n; i++) {
    for (int j = 1; j <= n; j++) {
        for (int k = 1; k <= n; k++) {
            if (m[j][k] > m[j][i]+m[i][k])
                m[j][k] = m[j][i]+m[i][k];
        }
    }
}

```

floyd-warshall打印路径

```

void floyd(){
    int i,j,k;
    for(i=1;i<=n;i++)
        for(j=1;j<=n;j++)
            dist[i][j]=map[i][j],path[i][j]=0;//需要先将dist[i][j]初始化为INF
    for(k=1;k<=n;k++)
        for(i=1;i<=n;i++)
            for(j=1;j<=n;j++)
                if(dist[i][k]+dist[k][j]<dist[i][j])
                    dist[i][j]=dist[i][k]+dist[k][j],path[i][j]=k;
}
void output(int i,int j){
    if(i==j) return;
    if(path[i][j]==0) cout<<j<<' ';
    else{
        output(i,path[i][j]);
        output(path[i][j],j);
    }
}
int main() {
    floyd();
    output(start,end);
}

```

两城市间最短路径，若最短路径有多个，输出花费最少的

```

void Floyd(int size) {
    for (int i = 0; i < size; i++) {

```

```

        for (int j = 0; j < size; j++) {
            for (int k = 0; k < size; k++) {
                if (m[j][k] > m[j][i]+m[i][k]) {
                    m[j][k] = m[j][i]+m[i][k];
                    spend[j][k] = spend[j][i]+spend[i][k];
                }
                else if (m[j][k] == m[j][i]+m[i][k]) {
                    if (spend[j][k] > spend[j][i]+spend[i][k])
                        spend[j][k] = spend[j][i] + spend[i][k];
                }
            }
        }
    }
}

```

两城市间最短路径，花费最多而且打印路径

```

#define INFINITY 9999
#include <iostream>
#include <cstdio>
using namespace std;
int N,M,S,D;
int map[600][600];
int val[600],dis[600],vis[600],cnt[600],sum[600],pre[600];
void path(int d) {
    if (pre[d] != -1) {
        path(pre[d]);
        printf("%d ", pre[d]);
    }
}

int main(int argc, char const *argv[])
{
    cin >> N >> M >> S >> D;
    for (int i = 0; i < N; i++) {
        cin >> val[i];//val记录花费
        for (int j = 0; j < N; j++) {
            if (i == j)
                map[i][j] = 0;
            else
                map[i][j] = INFINITY;
        }
        vis[i] = cnt[i] = sum[i] = 0;
        dis[i] = INFINITY;
        pre[i] = -1;
    }
    for (int i = 0; i < M; i++) {
        int v1, v2, d;
        cin >> v1 >> v2 >> d;
        map[v1][v2] = map[v2][v1] = d;
    }
    cnt[S] = 1;
    dis[S] = 0;
    vis[S] = 1;
    sum[S] = val[S];
    for (int i = 0; i < N; i++) {
        int min_dis = INFINITY;
        int min_ver = S;

```

```

        for (int j = 0; j < N; j++) {
            if (!vis[j] && dis[j] < min_dis) {
                min_dis = dis[j];
                min_ver = j;
            }
        }
        vis[min_ver] = 1;
        for (int j = 0; j < N; j++) {
            if (vis[j] == 0) {
                if (dis[j] > map[j][min_ver]+dis[min_ver]) {
                    dis[j] = map[j][min_ver]+dis[min_ver];
                    pre[j] = min_ver;
                    sum[j] = sum[min_ver]+val[j];
                    cnt[j] = cnt[min_ver];
                }
                else if (dis[j] == map[j][min_ver]+dis[min_ver]) {
                    cnt[j] += cnt[min_ver];
                    if (sum[j] < val[j]+sum[min_ver]) {
                        sum[j] = val[j]+sum[min_ver];
                        pre[j] = min_ver;
                    }
                }
            }
        }
    }
    printf("%d %d\n", cnt[D], sum[D]);
    path(D);
    printf("%d\n", D);
    return 0;
}

```

广度优先搜索，深度优先搜索

bfs原型

思想：从图中某顶点v出发，在访问了v之后依次访问v的各个未曾访问过的邻接点，然后分别从这些邻接点出发依次访问它们的邻接点，并使得“先被访问的顶点的邻接点先于后被访问的顶点的邻接点被访问”（通过进栈方式），直至图中所有已被访问的顶点的邻接点都被访问到。如果此时图中尚有顶点未被访问，则需要另选一个未曾被访问过的顶点作为新的起始点，重复上述过程，直至图中所有顶点都被访问到为止。

```

memset(visited,0,sizeof(visited));
mmeset(matrix,0,sizeof(matrix));
若连通，则matrix设置为1
memset(visited,0,sizeof(visited));
mmeset(matrix,0,sizeof(matrix));
若连通，则matrix设置为1
void bfs(int dim, int ver) {
    queue<int> q;
    q.push(ver);
    visited[ver] = 1;
    while (!q.empty()) {
        int temp = q.front();
        cout << temp << ' ';
        q.pop();
        for (int i = 0; i < dim; i++) {
            if (!visited[i] && matrix[temp][i]) {

```

```
        visited[i] = 1;
        q.push(i);
    }
}
}
```

dfs原型

思想:假设初始状态是图中所有顶点均未被访问，则从某个顶点 v 出发，首先访问该顶点，然后依次从它的各个未被访问的邻接点出发深度优先搜索遍历图，直至图中所有和 v 有路径相通的顶点都被访问到。若此时尚有其他顶点未被访问到，则另选一个未被访问的顶点作起始点，重复上述过程，直至图中所有顶点都被访问到为止。

```
void dfs(int dim, int ver) {  
    visited[ver] = 1;  
    cout << ver << ' ';  
    for (int i = 0; i < dim; i++) {  
        if (!visited[i] && matrix[ver][i])  
            dfs(dim,i);  
    }  
}
```

应用

列出一个图中的连通集

```
#include <bits/stdc++.h>
using namespace std;

int matrix[10][10] = {0};
int visit[10] = {0};

void dfs(int dim, int ver) {
    visit[ver] = 1;
    cout << ver << ' ';
    for (int i = 0; i < dim; i++) {
        if (!visit[i] && matrix[ver][i]) {
            dfs(dim, i);
        }
    }
}

void bfs(int dim, int ver) {
    queue<int> q;
    q.push(ver);
    visit[ver] = 1;
    while (!q.empty()) {
        int temp = q.front();
        cout << temp << ' ';
        q.pop();
        for (int i = 0; i < dim; i++) {
            if (!visit[i] && matrix[temp][i]) {
                visit[i] = 1;
                q.push(i);
            }
        }
    }
}
```

```

        if (!visit[i] && matrix[temp][i]) {
            q.push(i);
            visit[i] = 1;
        }

    }
}

int main() {
    int N, E;
    cin >> N >> E;
    memset(matrix, 0, sizeof(matrix));
    memset(visit, 0, sizeof(visit));
    for (int i = 0; i < E; i++) {
        int ver1, ver2;
        cin >> ver1 >> ver2;
        matrix[ver1][ver2] = 1;
        matrix[ver2][ver1] = 1;
    }
    for (int i = 0; i < N; i++) {
        if (visit[i]) continue;
        cout << '{' << ' ';
        dfs(N, i);
        cout << '}' << endl;//每个{}则为一个集合
    }
    memset(visit, 0, sizeof(visit));
    for (int i = 0; i < N; i++) {
        if (visit[i]) continue;
        cout << '{' << ' ';
        bfs(N, i);
        cout << '}' << endl;
    }
    return 0;
}

```

判断图是否连通

dfs, 若连通打印路径, 不连通输出0,一般使用这种方法, 也可以用floyd算法, 再遍历, 若存在matrix[i][j] == MAX, 则不连通

```

void dfs(int size, int ver);
int main() {
    dfs(size,v);
    for (int i = 1; i <= size; i++) {
        if (!Visited[i])
            flag = 1;
    }
    if (flag)
        printf(" 0\n");
    else
        printf("\n");
}

```

走迷宫

bfs, 迷宫是否可以走通, 输出最短路径

```
int maze[10][10];//迷宫为5*5的迷宫, 输出最短路径, 从(0,0)走到(4,4)
int vis[10][10],dist[10][10];
int dr[]={-1,1,0,0}//上,下,左,右
int dc[]={0,0,-1,1};
struct Node
{
    int r,c;//也可以在Node中加一个int pre属性, 然后做一个全局的nodes, 就不用pre[][]数组了.
    Node(int r,int c):r(r),c(c){}
    Node(){}
}pre[10][10];
queue<Node> Q;
bool BFS()
{
    while(!Q.empty()) Q.pop();
    memset(vis,0,sizeof(vis));
    dist[0][0]=0;
    vis[0][0]=1;
    Q.push(Node(0,0));
    while(!Q.empty())
    {
        Node node=Q.front();Q.pop();
        int r=node.r,c=node.c;
        for(int d=0;d<4;d++)
        {
            int nr=r+dr[d];
            int nc=c+dc[d];
            if(nr>=0&&nr<5&&nc>=0&&nc<5&&vis[nr][nc]==0&&maze[nr][nc]==0)
            {
                vis[nr][nc]=1;
                Q.push(Node(nr,nc));
                dist[nr][nc]=1+dist[r][c];
                pre[nr][nc]=Node(r,c);
                if(nr==4&&nc==4) return true;//一旦走到(4,4)立马return true则为最短路径, 表示该迷宫可以走
            }
        }
    }
    return 0;迷宫无法走通
}
int main()
{
    for(int i=0;i<5;i++)
        for(int j=0;j<5;j++)
            scanf("%d",&maze[i][j]);
    BFS();
    stack<Node> S;
    int cur_r=4,cur_c=4;
    while(true)
    {
        S.push(Node(cur_r,cur_c));
        if(cur_r==0&&cur_c==0) break;
        int r=cur_r,c=cur_c;
        cur_r=pre[r][c].r;
        cur_c=pre[r][c].c;
    }
    while(!S.empty())
    {
        Node node=S.top(); S.pop();
    }
}
```

```

    printf("(%d, %d)\n", node.r, node.c);
}
return 0;
}

```

dfs迷宫最短路径并输出最小步数

```

typedef pair<int,int> P;
const int MAX_N=1000;
const int INF=10000;
int N,M;
char maze[MAX_N][MAX_N];
int d[MAX_N][MAX_N];

int sx,sy;
int gx,gy;

int dx[4]={1,0,-1,0};
int dy[4]={0,1,0,-1};

void dfs(int x,int y)
{
    for(int i=0;i<4;i++)
    {
        for(int j=0;j<4;j++)
        {
            int nx=x+dx[i];
            int ny=y+dy[i];

            if(0<=nx&&nx<N&&0<=ny&&ny<M&&maze[nx][ny]!='#'&&d[nx][ny]==INF)//#表示该点有障碍
            {
                d[nx][ny]=d[x][y]+1;//记录步数

                dfs(nx,ny);
            }
        }
    }
}

void solve()
{
    for(int i=0;i<N;i++)
    {
        for(int j=0;j<M;j++)
        {
            if(maze[i][j]=='S') sx=i,sy=j;//S即开始位置
            if(maze[i][j]=='G') gx=i,gy=j;//G即目标位置
            d[i][j]=INF;
        }
    }

    d[sx][sy]=0;
    dfs(sx,sy);
    cout<<d[gx][gy]<<endl;
}

int main(int argc, char const *argv[])

```

```

{
    cin>>N>>M;
    for(int i=0;i<N;i++)
    {
        for(int j=0;j<M;j++)
        {
            cin>>maze[i][j];
        }
    }
    solve();
    return 0;
}

```

dfs打印迷宫所有路径

```

int maze[10][10];//迷宫
int vis[10][10];//记录迷宫中的某个位置是否访问过
int n,m;

int dir[4][2] = {{0,1},{0,-1},{1,0}, {-1,0}};//四个方向

struct point//位置
{
    int x,y;
} p;

stack<point> path,temp;//记录路径, temp是一个临时变量, 和path一起处理路径

int count;//路径条数

void dfs(int x,int y)//x,y:当前位置
{
    if(x==n-1 && y==m-1)//成功--下面处理路径问题
    {
        cout << "*****路径" << ++count << "*****" << endl;
        while(!path.empty())//将path里面的点取出来, 放在temp里面
        {//path从栈顶-栈底的方向, 路径是从终点-起点的顺序
            point p1 = path.top();
            path.pop();
            temp.push(p1);
        }
        while(!temp.empty())
        {//输出temp里面的路径, 这样刚好是从起点到终点的顺序
            point p1 = temp.top();
            temp.pop();
            path.push(p1);//将路径放回path里面, 因为后面还要回溯!!!
            cout << "(" << p1.x << "," << p1.y << ")" << endl;
        }
        return;
    }

    if(x<0 || x>=n || y<0 || y>=m)//越界
        return;

    //如果到了这一步, 说明还没有成功, 没有出界
    for(int i=0;i<4;i++)//从4个方向探测
    {
        int nx = x + dir[i][0];

```

```

int ny = y + dir[i][1];//nx,ny: 选择一个方向, 前进一步之后, 新的坐标
if(0<=nx && nx<n && 0<=ny && ny<m && maze[nx][ny]==0 && vis[nx][ny]==0)
    {//条件: nx,ny没有出界, maze[nx][ny]=0这个点不是障碍可以走, vis[nx][ny]=0说明(nx,ny)没有访问过, 可以访问

        vis[nx][ny]=1;//设为访问过
        p.x = nx;
        p.y = ny;
        path.push(p);//让当前点进栈

        dfs(nx,ny);//进一步探测

        vis[nx][ny]=0;//回溯
        path.pop();//由于是回溯, 所以当前点属于退回去的点, 需要出栈
    }
}
}

int main()
{
    count = 0;
    freopen("in.txt","r",stdin);//读取.cpp文件同目录下的名为in.txt的文件

    p.x = 0;
    p.y = 0;
    path.push(p);//起点先入栈

    cin >> n >> m;
    for(int i=0;i<n;i++)
    {
        for(int j=0;j<m;j++)
        {
            vis[i][j] = 0;
            cin >> maze[i][j];
        }
    }
    dfs(0,0);

    return 0;
}

```

怎么计下当前遍历为第几层(解决六度空间理论)

```

typedef struct listNode
{
    int data;
    listNode *next;
} *plist, nlist;
typedef struct
{
    plist *listArray;
    int GraghSize;
    bool *visited;
    //float *sixDegreePERSIST;
} *pGragh, nGragh;
pGragh CreateGragh(int size);
void ConnectGraghVertex(pGragh pG, int vStart, int vEnd);
void InsertListNode(plist pL, int vertex);

```

```

int BFS(pGragh pG, int vertex);
void DestoyGragh(pGragh pG);
int main(int argc, char const *argv[])
{
    int vertex, sides;
    cin >> vertex >> sides;
    pGragh pG = CreateGragh(vertex+1);
    for (int i = 0; i < sides; i++) {
        int ver1, ver2;
        cin >> ver1 >> ver2;
        ConnectGraghVertex(pG, ver1, ver2);
    }
    float percentage;
    int count;
    for (int i = 1; i <= vertex; i++) {
        count = BFS(pG, i);
        percentage = (float)count/vertex*100;
        cout << fixed << setprecision(2);
        cout << i << ":" << percentage << '%' << endl;
    }
    DestoyGragh(pG);
    return 0;
}

pGragh CreateGragh(int size) {
    pGragh pG = new nGragh();
    pG->listArray = new plist[size];
    pG->GraghSize = size;
    pG->visited = new bool[size];
    //pG->sixDegreePERSIST = new float[size];
    for (int i = 0; i < size; i++) {
        pG->listArray[i] = new nlist();
        pG->listArray[i]->data = i;
        pG->listArray[i]->next = NULL;
        //sixDegreePERSIST[i] = 0;
        pG->visited[i] = false;
    }
    return pG;
}

void ConnectGraghVertex(pGragh pG, int vStart, int vEnd) {
    if (pG == NULL || vStart < 0 || vEnd < 0 || vEnd == vStart)
        return;
    InsertListNode(pG->listArray[vStart], vEnd);
    InsertListNode(pG->listArray[vEnd], vStart);
}

void InsertListNode(plist pL, int vertex) {
    plist temp = new nlist();
    temp->data = vertex;
    temp->next = pL->next;
    pL->next = temp;
}

int BFS(pGragh pG, int vertex) { //用lastest记录一层的最后一个元素，当当前元素source若等于tail(此时tail用来记录真正的

```

最上面的一层的最后一个元素时，level++，表示遍历到一层)

```

    int source, lastest, level = 0, tail = vertex, sum = 1, maxLayer = 6;
    queue<int> que;
    if (pG->visited[vertex] == false) {
        que.push(vertex);

```

```

    pG->visited[vertex] = true;
}
while (!que.empty() && level < maxLayer) {
    source = que.front();
    que.pop();
    plist listlter = pG->listArray[source]->next;
    while (listlter) {
        if (pG->visited[listlter->data] == false) {
            pG->visited[listlter->data] = true;
            sum++;
            //cout << listlter->data << ' ';
            que.push(listlter->data);
            lastest = listlter->data;
        }
        listlter = listlter->next;
    }
    if (tail == source) {
        //cout << endl;
        level++;
        tail = lastest;
    }
}
for (int i = 0; i < pG->GraphSize; i++)
    pG->visited[i] = false;
return sum;
}
void DestroyGraph(pGraph pG) {
    delete []pG->visited;
    //delete []sixDegreePerset;
    for (int i = 0; i < pG->GraphSize; i++) {
        plist listlter = pG->listArray[i];
        while(listlter != NULL) {
            plist temp = listlter;
            listlter = listlter->next;
            delete temp;
        }
    }
    delete []pG->listArray;
    delete pG;
}

```

dfs实现类24点

```

#include <bits/stdc++.h>
using namespace std;
int arr[5];
int ans;
int result;
void dfs(int n) {//n表示四个运算符每次随机选一个
    for (int i = 0; i < n; i++)
        if (arr[i] > ans && arr[i] <= result)
            ans = arr[i];
    if (ans == result)
        return;
    if (n == 1)
        return;
    for (int i = 0; i < n; i++) {
        for (int j = i+1; j < n; j++) {
            int a = arr[i];
            int b = arr[j];
            int c = arr[i+j];
            int d = arr[i+j+1];
            if (a+b+c+d == result)
                ans = a+b+c+d;
        }
    }
}

```

```

        int b = arr[j];//此处i和j不可以互换
        arr[j] = arr[n-1];
        arr[i] = a+b;
        dfs(n-1);
        arr[i] = a-b;
        dfs(n-1);
        arr[i] = b-a;
        dfs(n-1);
        arr[i] = a*b;
        dfs(n-1);
        if (a != 0 && b%a == 0) {
            arr[i] = b/a;
            dfs(n-1);
        }
        if (b != 0 && a%b == 0) {
            arr[i] = a/b;
            dfs(n-1);
        }
        arr[i] = a;
        arr[j] = b;
    }
}
}

int main(int argc, char const *argv[])
{
    int ca;
    cin >> ca;
    while (ca--) {
        for (int i = 0; i < 5; i++)
            cin >> arr[i];
        cin >> result;
        ans = -2147483648;
        dfs(5);
        cout << ans << endl;
    }
    return 0;
}

```

dfs寻找从某一点出发最长路径

```

struct node {
    int next;//与该点连接的下一点
    int dis;//两点间距离
    node(int next = 0, int dis = 0) : next(next), dis(dis) {};
};

bool vis[10100];
vector<node> m[10100];
int answer;

void dfs(int start, int dis) {
    if (dis > answer)
        answer = dis;
    vis[start] = true;
    for (int i = 0; i < m[start].size(); i++) {
        if (!vis[m[start][i].next]) {
            dfs(m[start][i].next, dis+m[start][i].dis);//不可以写为l += m[start][i].dis
        }
    }
}

```

```

    }

}

int main(int argc, char const *argv[])
{
    int size,start;
    while (cin >> size >> start) {
        memset(m,0,sizeof(m));
        answer = 0;
        for (int i = 0; i < size-1; i++) {
            int s,t,dis;
            cin >> s >> t >> dis;
            m[s].push_back(node(t,dis));
            m[t].push_back(node(s,dis));
        }
        memset(vis,0,sizeof(vis));
        dfs(start,0);
        cout << answer << endl;
    }
    return 0;
}

```

bfs最长路径

```

void BFS(int i, int num){
    queue<int> p, q;
    visited[i] = true;
    q.push(i);
    p = q;
    while(q.size()){
        while(p.size()){
            int temp = q.front();
            p.pop();
            q.pop();
            for(int j = 0; j < num; j++){
                if (!visited[j] && adj[temp][j]){
                    visited[j] = true;
                    q.push(j);
                }
            }
        }
        p = q;
        count++;
    }
    if(count > result)result = count;
}

```

答案输出应该为result-1(也可能不是, 不清楚hhh)

任意进制的转换

```

int str_num(string str, int base) {//string类型的base进制转换为10进制
    int result = 0;
    for (int i = 0; i < str.length(); i++) {
        if (str[i] <= '9')

```

```

        result += (str[i]-'0')*pow(base,str.length()-i-1);
    else if (str[i] <= 'Z' && str[i] >= 'A')
        result += (str[i]-'A'+10)*pow(base,str.length()-i-1);
    else
        result += (str[i]-'a'+10)*pow(base,str.length()-i-1);
    }
    return result;
}

string num_str(int num, int base) { //10进制转换为base进制的string
    int temp[100];
    int index = 0;
    if (num==0)
        return (string)("0");
    while(num) {
        temp[index++] = num%base;
        num /= base;
    }
    string result("");
    for (int i = index-1; i >= 0; i--) {
        if (temp[i] >= 10)
            result += temp[i]-10+'A';
        else
            result += temp[i]+'0';
    }
    return result;
}

```

打素数表//求第n个素数

```

int prime[MAXN];
bool isprime[MAXN];
void sieve() {
    int p = 0;
    for (int i = 0; i < MAXN; i++)
        isprime[i] = true;
    isprime[0] = isprime[1] = false;
    for (int i = 2; i < MAXN; i++) {
        if (isprime[i]) {
            prime[p++] = i;
            for (int j = i+i; j < MAXN; j += i)
                isprime[j] = false;
        }
    }
}

```

#多项式加法

数组实现

```

#include <bits/stdc++.h>
int main(int argc, char const *argv[])

```

```

{
    int test;
    scanf("%d", &test);
    while (test--) {
        int size1, size2;
        scanf("%d", &size1);
        int cof1[size1], power1[size1];
        for (int i = 0; i < size1; i++)
            scanf("%d%d", &cof1[i], &power1[i]);
        scanf("%d", &size2);
        int cof2[size2], power2[size2];
        for (int i = 0; i < size2; i++)
            scanf("%d%d", &cof2[i], &power2[i]);
        int re_cof[size1+size2], re_power[size1+size2];
        int i = 0, j = 0;
        int index = 0;
        while(i < size1 && j < size2) {
            if (power1[i] == power2[j]) {
                if (cof1[i]+cof2[j]) { //幂相同且系数相加不为0
                    re_cof[index] = cof1[i]+cof2[j];
                    re_power[index++] = power1[i];
                }
                i++;
                j++;
            }
            else if (power1[i] > power2[j]) { //取幂较大者
                if (cof1[i]) {
                    re_cof[index] = cof1[i];
                    re_power[index++] = power1[i];
                }
                i++;
            }
            else { //取幂较大者
                if (cof2[j]) {
                    re_cof[index] = cof2[j];
                    re_power[index++] = power2[j];
                }
                j++;
            }
        }
        while(i < size1) {
            if (cof1[i]) {
                re_cof[index] = cof1[i];
                re_power[index++] = power1[i];
            }
            i++;
        }
        while(j < size2) {
            if (cof2[j]) {
                re_cof[index] = cof2[j];
                re_power[index++] = power2[j];
            }
            j++;
        }
        printf("%d\n", index);
        for (int k = 0; k < index; k++)
            printf("%d %d\n", re_cof[k], re_power[k]);
    }
    return 0;
}

```

链表实现

```
struct point
{
    int coe;
    int pow;
    point* next;
    point() {
        coe = pow = 0;
        next = NULL;
    }
    point(int c, int p) : coe(c), pow(p), next(NULL) {};
    point(point& copy) {
        coe = copy.coe;
        pow = copy.pow;
        next = NULL;
    }
};

typedef point* node;

void add(node addend, node summand, node* result) { //addend为多项式a, summand为多项式b, pow为幂, coe为系数, 此处result为指针的指针
    node temp = new point();
    *result = temp;
    while (addend && summand) {
        if (addend->pow == summand->pow) {
            if (addend->coe+summand->coe != 0) {
                temp->next = new point(addend->coe+summand->coe, addend->pow);
                temp = temp->next;
            }
            addend = addend->next;
            summand = summand->next;
        }
        else if (addend->pow > summand->pow) {
            if (addend->coe != 0) {
                temp->next = new point(addend->coe, addend->pow);
                temp = temp->next;
            }
            addend = addend->next;
        }
        else {
            if (summand->coe != 0) {
                temp->next = new point(summand->coe, summand->pow);
                temp = temp->next;
            }
            summand = summand->next;
        }
    }
    while (addend) {
        if (addend->coe) {
            temp->next = new point(addend->coe, addend->pow);
            temp = temp->next;
            addend = addend->next;
        }
    }
    while (summand) {
        if (summand->coe) {
            temp->next = new point(summand->coe, summand->pow);
            temp = temp->next;
            summand = summand->next;
        }
    }
}
```

```

        temp = temp->next;
        summand = summand->next;
    }
}

temp = *result;
*result = (*result)->next;
delete temp;
temp = NULL;
}

```

乘法转换成加法

下面的答案中，每个链表开头都有一个空的多余的节点，是为了方便下文相加时直接写root->next =，而不需要判断是否为头节点而单独写root=...，只需要输出时跳过头节点输出即可，为什么这里的代码那么烦，因为我懒得弄了啊略略略

```

struct node{
    int coef; //系数
    int expn; //指数
    node* next;
    node() : coef(0),expn(0),next(NULL){};
};

node* creat_list(int n){ //读入链表
    node *head, *r;
    head = new node;
    r = head;
    int coef , expn;
    while(n--){
        scanf("%d%d", &coef, &expn);
        node* tmp = new node; //创建临时结点
        tmp->coef = coef;
        tmp->expn = expn;
        r->next = tmp; //临时结点接到链表中
        r = tmp;
    }
    r->next = NULL; //结尾设为 NULL
    return head;
}

node* add_list(node* a,node* b){
    node *r,*fans, *ans;
    node *ha,*hb; //为了防止修改指针本身的值，使用代理指针来完成操作，也就是游标。
    fans = new node;
    ans = fans; //ans 作为fans 的“游标”
    ha = a->next;
    hb = b->next;
    while(ha && hb){
        node* tmp = new node; //建立一次即可
        if(ha->expn > hb->expn){ //每次把指数 (exponent) 较大的加入链表fans
            tmp->coef = ha->coef;
            tmp->expn = ha->expn;
            ans->next = tmp;
            ans = tmp;
            ha = ha->next;
        }
        else if(ha->expn < hb->expn){
            tmp->coef = hb->coef;
            tmp->expn = hb->expn;
            ans->next = tmp;
            ans = tmp;
            hb = hb->next;
        }
    }
}

```

```

        tmp->expn = hb->expn;
        ans->next = tmp;
        ans = tmp;
        hb = hb->next;
    }
    else{
        int mulOfcoef = (ha->coef)+(hb->coef); //如果指数相同，就把系数求和。
        if(mulOfcoef!=0){
            tmp->coef = mulOfcoef;
            tmp->expn = ha->expn;
            ans->next = tmp;
            ans = tmp;
        }
        ha = ha->next; //注意这里 即使和为0，也要移动“游标”
        hb = hb->next;
    }
}

while(ha){
    node* tmp = new node;
    tmp->coef = ha->coef;
    tmp->expn = ha->expn;
    ans->next = tmp;
    ans = tmp;
    ha = ha->next;
}
while(hb){
    node* tmp = new node;
    tmp->coef = hb->coef;
    tmp->expn = hb->expn;
    ans->next = tmp;
    ans = tmp;
    hb = hb->next;
}
ans->next = NULL; //结尾设为NULL
return fans;
}

node* multi_list(node* a,node* b){
    node* ha, *hb;
    node* ans,*fans;
    ha = a->next;
    hb = b->next;
    fans = creat_list(0);
    if(ha == NULL || hb == NULL){
        return fans;
    }
    node* tmp;
    while(ha != NULL){
        tmp = new node;
        ans = tmp;
        hb = b->next; //每次都是从 b 的第一项开始乘;
        while(hb != NULL){
            node* ltmp = new node;
            ltmp->expn = ha->expn + hb->expn; //指数相加，系数相乘
            ltmp->coef = ha->coef * hb->coef;
            hb = hb->next;
            ans->next= ltmp;
            ans = ltmp;
        }
        ans->next = NULL;
        fans = add_list(fans,tmp); //将乘法 分解成一次次的加法
    }
}

```

```

    ha = ha->next;
}
return fans;
}

void print_list(node* l){
    node *hc;
    int flag = 0;
    hc = l->next; //指针操作常用，用新创立的节点代替源节点操作
    if(hc == NULL){ //格式控制。。。真坑！
        printf("0 0");
    }
    while(hc != NULL){
        if(flag)
            printf(" ");
        else
            flag = 1;
        printf("%d %d",hc->coef,hc->expn);
        hc = hc->next;
    }
}

int main(){
    int n;
    scanf("%d",&n);
    node *a = creat_list(n);
    int m;
    scanf("%d",&m);
    node* b = creat_list(m);
    node* c = add_list(a,b);
    node* d = multi_list(a,b);
    print_list(d);
    printf("\n");
    print_list(c);
    printf("\n");
    return 0;
}

```

表达式计算

前缀表达式求值(都可以判断是否为合法输入)

递归版

```

float f() {
    char A[10] = {0};
    cin >> A;
    if (A[1] == 0) {
        switch(A[0]) {
            case '-' : return f()-f();
            case '+' : return f()+f();
            case '/' : {
                float fenmu = f();
                float fenzi = f();
                if (fenzi==0) {
                    cout << "ERROR\n";

```

```

        exit(0);
    }
    else return fenmu/fenzi;
}
case '*' : return f()*f();
default : return atof(A);
}
}
else {
if (A[0] == '+' || A[0] == '-') {
    char flag = A[0];
    int i = 0;
    while (A[i]) {
        A[i] = A[i+1];
        i++;
    }
    if (flag == '-')
        return 0-atof(A);
    else
        return atof(A);
}
else
    return atof(A);
}
}

```

非递归版

最后得到stack[0]为答案，若top==1，则输出答案stack[0]，否则，该表达式不合法

```

for (int i = prefix.length()-1; i >= 0; i--) {
switch(prefix[i]) {
    case ('-') : {
        double data1 = stack[--top];
        double data2 = stack[--top];
        stack[top++] = data1-data2;
        break;
    }
    case ('+') : {
        double data1 = stack[--top];
        double data2 = stack[--top];
        stack[top++] = data2+data1;
        break;
    }
    case ('*') : {
        double data1 = stack[--top];
        double data2 = stack[--top];
        stack[top++] = data1*data2;
        break;
    }
    case ('/') : {
        double data1 = stack[--top];
        double data2 = stack[--top];
        if (data2 == 0) {
            cout << "ERROR\n";
            exit(0);
        }
        stack[top++] = data1/data2;
    }
}
}

```

```

        break;
    }
    case (' '): {
        break;
    }
    default : {
        char num[40] = {0};
        int index = 0, j = i;
        while (j >= 0 && prefix[j] != ' ')
            j--;
        int a = j;
        while(j <= i) {
            num[index++] = prefix[++j];
        }
        num[index] = 0;
        stack[top++] = atof(num);
        i = a;
    }
}
}
}

```

中缀转后缀表达式

此处表达式以string类型输入，由于-可以为负号也可以为减号，比一般情况复杂，故下文在-号前加@表示此处-为负号，若不加则为减号，还有以vector<string>方式输入的，处理比较简单(不需要考虑-为负号还是减号)，在下面表达式求值

```

stack<char> res;
void Inexp_To_Post(string exp);
int main(int argc, char const *argv[])
{
    string exp;
    cin >> exp;
    Inexp_To_Post(exp);
    stack<char> post;
    while(!res.empty()) {
        char c = res.top();
        res.pop();
        post.push(c);
    }
    while(!post.empty()) {
        char c = post.top();
        post.pop();
        if (c == '#') {
            if (!post.empty())
                cout << ' ';
        }
        else if (c == '@') {
            c = post.top();
            post.pop();
            if (c == '-')
                cout << c;
        }
        else if ((c < '0' || c > '9') && c != '.') {
            cout << c;
            if (!post.empty())
                cout << ' ';
        }
    }
}

```

```

    }
    else
        cout << c;
    }
    cout << endl;
}

void Inexp_To_Post(string exp) {
    int top = -1;
    char *st = new char[exp.length()+1];
    int i = 0;
    char c = exp[0];
    while (i < exp.size()) {
        //cout << c << ' ' << i << endl;
        switch(c) { //只有当前操作符的优先级高于操作符栈栈顶的操作符的优先级,
            //才入栈, 否则弹出操作符以及操作数进行计算直至栈顶操作符的优先级低于当前操作符,
            //然后将当前操作符压栈
            case '(' : {
                top++;
                st[top] = c;
                break;
            }
            case ')' : {
                while(st[top] != '(') {
                    res.push(st[top]);
                    top--;
                }
                top--;
                break;
            }
            case '-' :
            case '+' : {
                if (i==0 || (exp[i-1] != ')' && (exp[i-1] < '0' || exp[i-1] > '9'))) {//满足则为负
号
                    res.push('@');
                    res.push(c);
                }
                else {
                    while (top > -1 && st[top] != '(') {
                        res.push(st[top]);
                        top--;
                    }//栈内运算符优先级较高时直接进栈
                    top++;
                    st[top] = c;
                }
                break;
            }
            case '*' :
            case '/' : {
                while (top > -1 && st[top] != '(' && (st[top] == '\\\\' || st[top] == '**')) {
                    res.push(st[top]);
                    top--;
                }//将优先级比自己高的抛出
                top++;
                st[top] = c;
                break;
            }
            default : {
                while ((c >= '0' && c <= '9') || c == '.') {
                    res.push(c);
                    i++;
                    c = exp[i];
                }
            }
        }
    }
}

```

```

        }
        res.push('#');
        i--;
    }
}
i++;
c = exp[i];
}
while (top > -1)
    res.push(st[top--]);
delete []st;
}

```

表达式求值//中缀输入，转为后缀，求值

```

//只有当前操作符的优先级高于操作符栈栈顶的操作符的优先级，才入栈，否则弹出操作符以及操作数进行计算直至栈顶
操作符的优先级低于当前操作符，然后将当前操作符压栈
// int get_priority(char ch) {//得到优先级,return值越大，优先级越高
//     if (ch == '(')
//         return 0;
//     if (ch == '-' || ch == '+')
//         return 1;
//     if (ch == '*' || ch == '/')
//         return 2;
//     if (ch == '#')
//         return 3;#代表负号
// }

//只有当前操作符的优先级高于操作符栈栈顶的操作符的优先级，才入栈，否则弹出操作符以及操作数进行计算直至栈顶
操作符的优先级低于当前操作符，然后将当前操作符压栈
void infix_to_post(vector<string>& infix, vector<string>& post) {
    stack<string> st;
    for (int i = 0; i < infix.size(); i++) {
        switch(infix[i][0]) {
            case '(' : {
                st.push(infix[i]);
                break;
            }
            case ')' : {
                while (st.top() != "(") {
                    post.push_back(st.top());
                    st.pop();
                }
                st.pop();
                break;
            }
            case '-' : {
            case '+' : {
                while (!st.empty() && st.top() != "(") {
                    post.push_back(st.top());
                    st.pop();
                }
                st.push(infix[i]);
                break;
            }
            case '*' : {
            case '/' : {
                while (!st.empty() && (st.top() == "*" || st.top() == "/")) {

```

```

        post.push_back(st.top());
        st.pop();
    }
    st.push(infix[i]);
    break;
}
default :
    post.push_back(infix[i]);
}
}

while (!st.empty()) {
    post.push_back(st.top());
    st.pop();
}
}

int evaluateExpression(vector<string> &expression) {
    stack<double> arr;
    vector<string> post;
    infix_to_post(expression,post);
    for (int i = 0; i < post.size(); i++) {
        switch(post[i][0]) {
            case ('-') : {
                double num1 = arr.top();
                arr.pop();
                double num2 = arr.top();
                arr.pop();
                arr.push(num2-num1);
                break;
            }
            case ('+') : {
                double num1 = arr.top();
                arr.pop();
                double num2 = arr.top();
                arr.pop();
                arr.push(num1+num2);
                break;
            }
            case ('*') : {
                double num1 = arr.top();
                arr.pop();
                double num2 = arr.top();
                arr.pop();
                arr.push(num1*num2);
                break;
            }
            case ('/') : {
                double num1 = arr.top();
                arr.pop();
                double num2 = arr.top();
                arr.pop();
                arr.push(num2/num1);
                break;
            }
        default : {
            int num;
            stringstream stream;
            stream.clear();
            stream << post[i];
            stream >> num;
            arr.push(num);
        }
    }
}

```

```

        }
    }
    if (arr.empty())
        return 0;
    int num = arr.top();
    return num;
}

```

并查集

```

int father[maxn]; // 储存i的father父节点

void makeSet() {
    for (int i = 0; i < maxn; i++)
        father[i] = i;
}

int findRoot(int x) { // 迭代找根节点
    int root = x; // 根节点
    while (root != father[root]) { // 寻找根节点
        root = father[root];
    }
    while (x != root) {
        int tmp = father[x];
        father[x] = root; // 根节点赋值
        x = tmp;
    }
    return root;
}

void Union(int x, int y) { // 将x所在的集合和y所在的集合整合起来形成一个集合。
    int a, b;
    a = findRoot(x);
    b = findRoot(y);
    father[a] = b; // y连在x的根节点上 或者 father[b] = a为x连在y的根节点上;
}

```

STL

排序

```

sort(s.begin(), s.end());
自定义类型
struct ss
{
    int a, b;
};
bool comp(const ss &a, const ss &b)
{
    return a.a < b.a;
}

```

```
}
```

```
vector<ss> s;
```

```
sort(s.begin(), s.end(), comp);
```

各容器的删除

```
连续性序列容器(vector, string, deque)
```

```
vector<int> c;
```

```
for (vector<int>::iterator it = c.begin(); it != c.end(); ) {
```

```
    if (need_delete())
```

```
        it = c.erase(it);
```

```
    else
```

```
        it++;
```

```
}
```

```
关联容器(set, multiset, map, multimap)
```

```
map<int, int> m;
```

```
for (map<int, int>::iterator::it = m.begin(); it != m.end(); ) {
```

```
    if(need_delete())
```

```
        m.erase(it++);
```

```
    else
```

```
        it++;
```

```
}
```

```
非连续序列容器list
```

```
以上两种方法均适用
```

map

构造

```
map<int, string> mapStudent;
```

插入

```
① mapStudent.insert(pair<int, string>(3, "student_three"));  
② mapStudent.insert(map<int, string>::value_type (1, "student_one"));  
③ mapStudent[1] = "student_one"; //若key已经存在，会覆盖数据
```

是否插入成功

```
map<int, string> mapStudent;  
Pair<map<int, string>::iterator, bool> Insert_Pair;  
Insert_Pair = mapStudent.insert(pair<int, string>(1, "student_one"));  
If(Insert_Pair.second == true)  
{  
    cout<<"Insert Successfully"<<endl;  
}  
Else  
{  
    cout<<"Insert Failure"<<endl;  
}
```

查看size

```
int nSize = mapStudent.size();
```

数据查找

```
① 用count函数来判定关键字是否出现，其缺点是无法定位数据出现位置，由于map的特性  
，一对一的映射关系，就决定了count函数的返回值只有两个，要么是0，要么是1，  
出现的情况，当然是返回1了
```

```
② find  
    mapStudent.insert(pair<int, string>(1, "student_one"));
```

```

mapStudent.insert(pair<int, string>(2, "student_two"));
map<int, string>::iterator iter;
iter = mapStudent.find(1);
if(iter != mapStudent.end())
{
    cout<<"Find, the value is "<<iter->second<<endl;
}
Else
{
    cout<<"Do not Find"<<endl;
}

```

数据清空与判断是否为空

清空map中的数据可以用clear()函数，判定map中是否有数据可以用empty()函数，它返回true则说明是空map
删除

```

iter = mapStudent.find(1);
mapStudent.erase(iter);
//或者
//如果要删除1，用关键字删除
int n = mapStudent.erase(1); //如果删除了会返回1，否则返回0

```

交换

```
a.swap(b);
```

vector

1. 构造函数

```

vector(): 创建一个空vector
vector(int nSize): 创建一个vector，元素个数为nSize
vector(int nSize, const T& t): 创建一个vector，元素个数为nSize，且值均为t
vector(const vector&): 复制构造函数
vector(begin,end): 复制[begin,end)区间内另一个数组的元素到vector中

```

2. 增加函数

```

void push_back(const T& x): 向量尾部增加一个元素x
iterator insert(iterator it,const T& x): 向量中迭代器指向元素前增加一个元素x
iterator insert(iterator it,int n,const T& x): 向量中迭代器指向元素前增加n个相同的元素x
iterator insert(iterator it,const_iterator first,const_iterator last): 向量中迭代器指向元素前插入
另一个相同类型的[first,last)间的数据

```

3. 删除函数

```

iterator erase(iterator it): 删除向量中迭代器指向元素
iterator erase(iterator first,iterator last): 删除向量中[first, last)中元素
void pop_back(): 删除向量中最后一个元素
void clear(): 清空向量中所有元素

```

4. 遍历函数

```

reference at(int pos): 返回pos位置元素的引用
reference front(): 返回首元素的引用
reference back(): 返回尾元素的引用
iterator begin(): 返回向量头指针，指向第一个元素
iterator end(): 返回向量尾指针，指向向量最后一个元素的下一个位置
reverse_iterator rbegin(): 反向迭代器，指向最后一个元素
reverse_iterator rend(): 反向迭代器，指向第一个元素之前的位置

```

5. 判断函数

bool empty() const: 判断向量是否为空，若为空，则向量中无元素

6. 大小函数

```

int size() const: 返回向量中元素的个数
int capacity() const: 返回当前向量能容纳的最大元素值
int max_size() const: 返回最大可允许的vector元素数量值

```

7. 其他函数

```
void swap(vector&): 交换两个同类型向量的数据  
void assign(int n, const T& x): 设置向量中第n个元素的值为x  
void assign(const_iterator first, const_iterator last): 向量中[first, last)中元素设置成当前向量元素
```

queue

```
push(x) 将x压入队列的末端  
pop() 弹出队列的第一个元素(队顶元素), 注意此函数并不返回任何值  
front() 返回第一个元素(队顶元素)  
back() 返回最后被压入的元素(队尾元素)  
empty() 当队列为空时, 返回true  
size() 返回队列的长度
```

dequeue

STL里面容器默认用的是 `vector`. 比较方式默认用 `operator<`, 所以如果你把后面俩个参数 缺省的话, 优先队列就是大顶堆, 队头元素最大。

```
priority_queue<int, vector<int>, less<int> >q; // 使用priority_queue<int> q1; 一样  
priority_queue<int, vector<int>, greater<int> >q;
```

自定义类型需要自己重载`operator<`

```
struct Node{  
    int x, y;  
}node;  
bool operator<( Node a, Node b){  
    if(a.x==b.x) return a.y>b.y;  
    return a.x>b.x;  
}  
priority_queue<Node>q;  
empty() 如果队列为空返回真  
pop() 删除对顶元素  
push(entry) 加入一个元素  
size() 返回优先队列中拥有的元素个数  
top() 返回优先队列堆顶元素
```

set(set默认从小打到排序)

1. 元素插入: `insert()`

2. 元素检索: `find()`, 若找到, 返回该键值迭代器的位置, 否则, 返回最后一个元素后面一个位置。

```
it = s.find(5);  
if (it == s.end())  
    cout << "找到"  
else  
    cout << "未找到"
```

3. 删除

4. 自定义比较函数

(1) 元素不是结构体:

例:

```
// 自定义比较函数myComp, 重载“()”操作符
```

```

struct myComp
{
    bool operator()(const your_type &a,const your_type &b)
    [
        return a.data-b.data>0;
    ]
}
set<int,myComp>s;
.....
set<int,myComp>::iterator it;

```

(2)如果元素是结构体，可以直接将比较函数写在结构体内。

例：

```

struct Info
{
    string name;
    float score;
    //重载“<”操作符，自定义排序规则
    bool operator <(const Info &a) const
    {
        //按score从大到小排列
        return a.score<score;
    }
}
set<Info> s;
.....
set<Info>::iterator it;

```

list

2.1 list中的构造函数：

list() 声明一个空列表；

list(n) 声明一个有n个元素的列表，每个元素都是由其默认构造函数**T()**构造出来的

list(n,val) 声明一个由n个元素的列表，每个元素都是由其复制构造函数**T(val)**得来的

list(n,val) 声明一个和上面一样的列表

list(first,last) 声明一个列表，其元素的初始值来源于由区间所指定的序列中的元素

2.2 begin()和end()：通过调用**list**容器的成员函数**begin()**得到一个指向容器起始位置的**iterator**，可以调用**list**容器的 **end()** 函数来得到**list**末端下一位置，相当于：**int a[n]**中的第n+1个位置a[n]，实际上是不存在的，不能访问，经常作为循环结束判断结束条件使用。

2.3 push_back() 和push_front()：使用**list**的成员函数**push_back**和**push_front**插入一个元素到**list**中。其中 **push_back()**从**list**的末端插入，而 **push_front()**实现的从**list**的头部插入。

2.4 empty()：利用**empty()** 判断**list**是否为空。

2.5 resize()：如果调用**resize(n)**将**list**的长度改为只容纳n个元素，超出的元素将被删除，如果需要扩展那么调用默
认构造函数**T()**将元素加到**list**末端。如果调用**resize(n,val)**，则扩展元素要调用构造函数**T(val)**函数进行元素构造，
其余部分相同。

2.6 clear()：清空**list**中的所有元素。

2.7 front()和back()：通过**front()**可以获得**list**容器中的头部元素，通过**back()**可以获得**list**容器的最后一个元
素。但是有一点要注意，就是**list**中元素是空的时候，这时候调用**front()**和**back()**会发生什么呢？实际上会发生不能正
常读取数据的情况，但是这并不报错，那我们编程序时就要注意了，个人觉得在使用之前最好先调用**empty()**函数判断
list是否为空。

2.8 pop_back和pop_front(): 通过删除最后一个元素，通过`pop_front()`删除第一个元素；序列必须不为空，如果当`list`为空的时候调用`pop_back()`和`pop_front()`会使程序崩掉。

2.9 assign(): 具体和vector中的操作类似，也是有两种情况，第一种是：`l1.assign(n, val)`将 l1中元素变为n个`T(val)`。第二种情况是：`l1.assign(l2.begin(), l2.end())`将l2中的从`l2.begin()`到`l2.end()`之间的数值赋值给l1。

2.10 swap(): 交换两个链表(两个重载)，一个是`l1.swap(l2)`；另外一个是`swap(l1, l2)`，都可能完成连个链表的交换。

2.11 reverse(): 通过`reverse()`完成`list`的逆置。

2.12 merge(): 合并两个链表并使之默认升序(也可改)，`l1.merge(l2, greater<int>())`；调用结束后l2变为空，l1中元素包含原来l1 和 l2中的元素，并且排好序，升序。其实默认是升序，`greater<int>()`可以省略，另外`greater<int>()`是可以变的，也可以不按升序排列。

stack

- 1.入栈：如`s.push(x);`
- 2.出栈：如 `s.pop()`.注意：出栈操作只是删除栈顶的元素，并不返回该元素。
- 3.访问栈顶：如`s.top();`
- 4.判断栈空：如`s.empty()`.当栈空时返回`true`。
- 5.访问栈中的元素个数，如`s.size ()`；

multimap

`begin()`返回指向第一个元素的迭代器
`clear()`删除所有元素
`count()`返回一个元素出现的次数
`empty()`如果`multimap`为空则返回真
`end()`返回一个指向`multimap`末尾的迭代器
`equal_range()`返回指向元素的key为指定值的迭代器对
`erase()`删除元素
`find()`查找元素
`get_allocator()`返回`multimap`的配置器
`insert()`插入元素
`key_comp()`返回比较key的函数
`lower_bound()`返回键值 \geq 给定元素的第一个位置
`max_size()`返回可以容纳的最大元素个数
`rbegin()`返回一个指向`multimap`尾部的逆向迭代器
`rend()`返回一个指向`multimap`头部的逆向迭代器
`size()`返回`multimap`中元素的个数
`swap()`交换两个`multimaps`
`upper_bound()`返回键值 $>$ 给定元素的第一个位置
`value_comp()`返回比较元素value的函数
遍历
遍历和删除

C++的STL的关联容器`multimap`允许一个key值对应多个值，当对整个`multimap`进行遍历时可以使用迭代器，迭代器会指向多个键值对，而不是一个键值后的多个值，当希望输出`key:value1 value2...`形式时，需要使用`count`函数计算一个key值包含多少个值，然后使用循环对迭代器自增输出这些值。当需要删除多个值中的某一个时，使用`equal_range`函数保存某一key值对应所有值的起始位置，

这两个位置保存在一个pair对中，用first值遍历这些值，找到符合条件的删除，否则自增，由于删除操作会使迭代器失效，故将erase函数的返回值赋值给first迭代器，令其指向删除元素的下一个元素，保证first迭代器不会失效。

```
int main ()
{
    multimap<string,int> authors;
    for (int i=0;i<10;i++)//初始化一个multimap
    {
        for (int j=0;j<=i;j++)
        {
            string s;
            s+= 'a'+j;
            authors.insert(make_pair(s,i));
            authors.insert(make_pair(s,i-1));
        }
    }
    multimap<string,int>::iterator map_it=authors.begin();
    typedef multimap<string,int>::iterator authors_it;
    cout<<"原multimap容器: "<<endl;
    while (map_it!=authors.end())//遍历multimap容器，由于map_it一次指向包含一个键值对，例如，当前map_it指向(a,0)，自增之后指向(a,1)
        //故仅在第一次输出first值，使用count函数计算该键对应的值个数，然后循环输出各second值，同时增加map_it
    {
        cout<<map_it->first<<":";
        typedef multimap<string,int>::size_type sz_type;//multimap数量类型
        sz_type cnt=authors.count(map_it->first);//返回一个键对应的值个数
        for (sz_type i=0;i!=cnt;++map_it,++i)//循环输出各值，同时自增map_it
        {
            cout<<map_it->second<< " ";
        }
        cout<<endl;
    }

    map_it=authors.find("c");//删除(c,5)
    pair<authors_it,authors_it> pos=authors.equal_range(map_it->first);//利用一对multimap<string,int>指向第一个出现(c,5)的位置和最后一个出现(c,5)的位置
    while (pos.first!=pos.second)
    {
        if (pos.first->second==5)//当pos指向5时
        {
            pos.first=authors.erase(pos.first); //删除后会改变pos迭代器，故赋值给自身，指向删除后的下一个键值对
        }
        else
            ++pos.first;//不进行删除操作则自增
    }
    cout<<"删除 (c,5) 之后的multimap容器: "<<endl;//输出删除(c,5)之后的multimap
    map_it=authors.begin();
    while (map_it!=authors.end())//遍历multimap容器，由于map_it一次指向包含一个键值对，例如，当前map_it指向(a,0)，自增之后指向(a,1)
        //故仅在第一次输出first值，使用count函数计算该键对应的值个数，然后循环输出各second值，同时增加map_it
    {
        cout<<map_it->first<<":";
    }
}
```

```
typedef multimap<string,int>::size_type sz_type;//multimap数量类型
sz_type cnt=authors.count(map_it->first);//返回一个键对应的值个数
for (sz_type i=0;i!=cnt;++map_it,++i)//循环输出各值，同时自增map_it
{
    cout<<map_it->second<<" ";
}
cout<<endl;
}
system("pause");
return 0;
}
```