

# 南京林业大学

## 实验报告

学 院： 信息科学技术学院、人工智能学院

专 业： 计算机科学与技术

课程名称： 数据结构

学 号： 2351610105

学生姓名： 方泽宇

指导教师： 张黎宁

二〇二四年三月——二〇二四年六月

## 实验一：设计实现抽象数据类型“有理数”

### 一. 实验内容：

设计实现抽象数据类型“有理数”

设计并上机实现抽象数据类型“有理数”，有理数的基本操作包括：两个有理数的加、减、乘、除等（包括有理数的创建和输出等）。

### 二. 实验要求：

- (1) 要有能根据用户输入选择不同运算的菜单选择界面。
- (2) 有理数的类型，我们可以构造成一个结构体类型，这个结构体由两个整数构成，分别表示有理数的分子和分母。
- (3) 在初始化或创建一个有理数时，可以给出有理数的分子和分母来创建一个有理数；也可以给出一个小数形式的有理数，来计算对应的分子分母来创建一个有理数（可设置一个允许的计算误差）。
- (4) 以分数形式创建有理数时，要处理分母为零的异常情况。
- (5) 输出不能有类似于 4/4、3/6 这样的结果数据。

### 三. 源程序：

```
#include<stdio.h>
typedef struct
{
    int num,deno;
}RatNum;
double fabs(double x)
{
    return (x>0)?x:(-x);
}
int gcd(int a,int b)
{
    return b?gcd(b,a%b):a;
}
int lcm(int a,int b)
{
    return a/gcd(a,b)*b;
}
void create(RatNum *p)
{
    printf("input a num:");
    scanf("%d %d",&(p->num),&(p->deno));
}
void print(RatNum a)
{
    if(a.deno==1) printf("%d",a.num);
```

```
    else printf("%d/%d", a.num, a.deno);
}
RatNum approximation(RatNum a)
{
    int factor=gcd(a.num,a.deno);
    a.deno/=factor;
    a.num/=factor;
    if((a.deno<0&&a.num>0) || (a.deno<0&&a.num<0))
    {
        a.num=-a.num;
        a.deno=-a.deno;
    }
    return a;
}
RatNum add(RatNum a,RatNum b)
{
    RatNum res;
    res.num=a.num*b.deno+b.num*a.deno;
    res.deno=a.deno*b.deno;
    return approximation(res);
}
RatNum sub(RatNum a,RatNum b)
{
    RatNum res;
    res.num=a.num*b.deno-b.num*a.deno;
    res.deno=a.deno*b.deno;
    return approximation(res);
}
RatNum muti(RatNum a,RatNum b)
{
    RatNum res;
    res.num=a.num*b.num;
    res.deno=a.deno*b.deno;
    return approximation(res);
}
RatNum div(RatNum a,RatNum b)
{
    RatNum res={0,0};
    if(a.deno==0 || b.deno==0)
    {
        printf("error\n");
        return res;
    }
    res.num=a.num*b.deno;
    res.deno=a.deno*b.num;
```

```
    return approximation(res);
}

RatNum Create_RatNum(double x, double error) {
    int sign=(x<0)?-1:1;
    x=(x>0)?x:(-x);
    double lower_n=0, lower_d=1;
    double upper_n=1, upper_d=0;
    int num=1, denom=1;
    while (1)
    {
        num=lower_n+upper_n;
        denom=lower_d+upper_d;
        double approx = (double)num / denom;
        if (fabs(approx - x) < error)
        {
            RatNum result = {sign * num, denom};
            return result;
        }
        if (approx < x)
        {
            lower_n = num;
            lower_d = denom;
        } else
        {
            upper_n = num;
            upper_d = denom;
        }
    }
}

int main()
{
    int ch;
    RatNum a,b,res;
    printf("1.Create from a decimal\n2.clac\n");
    scanf("%d",&ch);
    if(ch==1)
    {
        printf("Input the num and the maxmize error\n");
        double input,error;
        scanf("%lf",&input);
        RatNum x=Create_RatNum(input,error);
        print(x);
    }
    else
    {

```

```
printf("1.add\n2.sub\n3.muti\n4.div\n");
scanf("%d",&ch);
printf("give the first number\n");
create(&a);
printf("give the second number\n");
create(&b);
switch (ch)
{
    case 1:res=add(a,b);break;
    case 2:res=sub(a,b);break;
    case 3:res=muti(a,b);break;
    case 4:res=div(a,b);break;
    default:break;
}
print(res);
}
return 0;
}
```

#### 四. 运行结果:

##### (1) 加法运算

```
1.add
2.sub
3.muti
4.div
1
give the first number
input a num:2 3
give the second number
input a num:4 5
22/15%
○ fang50253@MacBook-Pro 课程作业 %
```

##### (2) 减法运算

```
1.add
2.sub
3.muti
4.div
2
give the first number
input a num:1 2
give the second number
input a num:1 2
0%
```

fang50253@MacBook-Pro 课程作业 %

## (3) 乘法运算

```
1.add
2.sub
3.muti
4.div
3
give the first number
input a num:1 3
give the second number
input a num:7 2
7/6%
```

fang50253@MacBook-Pro 课程作业 %

## (4) 除法运算

```
2.sub
3.muti
4.div
4
give the first number
input a num:3 7
give the second number
input a num:7 0
error
```

## (5) 从小数创建有理数

```
1.Create from a decimal
2.clac
1
Input the num and the maxmize error
2.667 0.01
2667/1000%
```

## 五. 实验小结:

- (1) 在本次实习过程中, 我们要通过多种运行的数据来寻找程序可能存在的问题, 并通过修改程序来解决他们。比如对于一个有理数(分数), 其分母不能为 0, 一个被除数也不可以为 0。
- (2) 本次实验采用一个结构体类型来保存有理数, 保证了有理数的存储精度不会因为数据类型的精度而丢失。
- (3) 代码实现了有理数的基本运算, 包括加法、减法、乘法和除法, 这些操作都是基于有理数的分子和分母的基本运算。使用了 `approximation` 函数来简化结果, 确保结果为最简分数形式。

## 实验二：顺序表基本操作的实现

### 一. 实验内容:

实现顺序表的初始化, 输出顺序表中各元素的值, 在顺序表中插入数据元素, 删除数据元素, 求顺序表的长度, 顺序表的逆置, 顺序表的按值从小到大排序, 合并有序的两个顺序表等操作。

### 二. 实验要求:

- (1) 要有能根据用户输入选择不同运算的菜单选择界面。
- (2) 顺序表中数据元素的类型统一抽象表示为 ElemType 类型, 具体类型不限, 可以是整型、实型、字符型、或者是自己构造的一种结构体类型。
- (3) 实现课后习题有关顺序表运算算法

### 三. 源程序:

```
#include<stdio.h>
#define ElemType int
#define ERROR 1
#define maxsize 110
typedef struct
{
    ElemType elem[maxsize]; // 静态申请顺序表所需要的空间
    int last;
}SeqList;
SeqList List;
void Init_List(SeqList *L);
void Visit(SeqList *L);
void Create(SeqList *L,int n);
void Insert_List(SeqList *L,int arr,ElemType x);
int Seq_Length(SeqList *L);
void Reserve_List(SeqList *L,int l,int r);
void Sort_List(SeqList *L,int l,int r);
void Merge_List(SeqList *a,SeqList *b,SeqList *c);
void swap(ElemType *a,ElemType *b);
void Merge_List(SeqList *a,SeqList *b,SeqList *c);
void Delete_List(SeqList *L,int arr);
void Merge_Sort_List(SeqList *a,SeqList *b,SeqList *c);
void swap(ElemType *a,ElemType *b)
{
    ElemType tmp=*a;
    *a=*b;
```



```
    *b=tmp;
}

void Init_List(SeqList *L)// 初始化操作
{
    L->last=-1;
}

void Visit(SeqList *L)// 按顺序输出顺序表
{
    if(L->last==-1)
    {
        printf("Empty_array\n");
        return;
    }
    for(int i=0;i<=L->last;++i) printf("%d ",L->elem[i]);
    printf("\n");
}

void Create(SeqList *L,int n)// 创建一个单链表, 并使用尾插法输入 n 个数字
{
    if(n>maxsize)
    {
        printf("Failed to create the array,the num is out of the size\n");
    }
    Init_List(L);
    // 初始化链表L
    for(int i=0;i<n;++i)
    {
        // 依次输入数据, 并将其添加至顺序表尾部
        ElemType input;
        scanf("%d",&input);
        Insert_List(L,i-1,input);
    }
}

void Insert_List(SeqList *L,int arr,ElemType x)// 在顺序表L 中, 索引号arr 的数字之后插入数据元素x
{
    // 第一步, 移动索引号为arr+1 到last 到元素, 使其索引号+1
    if(L->last==maxsize)
    {
        // 处理静态顺序表满的异常
        printf("Error,the array is full\n");
        return;
    }
    else ++L->last;
    for(int i=L->last;i>=arr+1;--i) L->elem[i+1]=L->elem[i];
    // 第二步, 将x 添加到顺序表的arr 结点当中
```

```
L->elem[arr+1]=x;
}
int Seq_Length(SeqList *L)// 返回顺序表L 的长度
{
    return L->last+1;
}
void Reserve_List(SeqList *L,int l,int r)// 对顺序表L 中, 索引号为l 到r 的项进行逆序操作
{
    if(l>r) printf("Error,the num L is larger than R\n");
    while(l<r)
    {
        swap(&(L->elem[l]),&(L->elem[r]));
        ++l,--r;
    }
}
void Sort_List(SeqList *L,int l,int r)// 将顺序表中, 对索引号l 到r 的项从小到大排序
{
    if(l>=r) return;
    int arr=L->elem[l+r>>1];
    int i=l-1,j=r+1;
    while(i<j)
    {
        do ++i; while(L->elem[i]<arr);
        do --j; while(L->elem[j]>arr);
        if(i<j) swap(&(L->elem[i]),&(L->elem[j]));
    }
    Sort_List(L,l,j);
    Sort_List(L,j+1,r);
}
void Merge_List(SeqList *a,SeqList *b,SeqList *c)// 将a 和b 两个顺序表合并, 并将答案存入 c
中
{
    int i=0,j=0,k=0;
    for(int i=0;i<=a->last;++i,++k) Insert_List(c,k-1,a->elem[i]);
    for(int i=0;i<=b->last;++i,++k) Insert_List(c,k-1,b->elem[i]);
}
void Merge_Sort_List(SeqList *a,SeqList *b,SeqList *c)// 合并两个有序的顺序表
{
    int i=0,j=0,k=0;
    while(i<=a->last&& j<=b->last)
    {
        if(a->elem[i]<b->elem[j])
        {
            Insert_List(c,k-1,a->elem[i]);
            ++i;
        }
    }
}
```

```
}
else
{
    Insert_List(c,k-1,b->elem[j]);
    ++j;
}
++k;
}
while(i<=a->last)
{
    Insert_List(c,k-1,a->elem[i]);
    ++i,++k;
}
while(j<=b->last)
{
    Insert_List(c,k-1,b->elem[j]);
    ++j,++k;
}
}

void Delete_List(SeqList *L,int arr)//删除顺序表L 中索引号为arr 的元素
{
    for(int i=arr+1;i<=L->last;++i) L->elem[i-1]=L->elem[i];
    --L->last;
}

void fun()
{
    //system(CLS);
    printf("1.在顺序表中插入数组元素\n");
    printf("2.删除数组中的元素\n");
    printf("3.求顺序表的长度\n");
    printf("4.顺序表的逆序\n");
    printf("5.将顺序表中的值从小到大排序\n");
    printf("6.合并两个顺序表\n");
    printf("7.合并两个有序的顺序表\n");
    printf("8.输出 L\n");
    printf("9.使用尾插法一次性向 List 输入数据\n");
    int ch;
    scanf("%d",&ch);
    if(ch==1)
    {
        printf("2 个参数: 在顺序表 L 中, 索引号 arr 的数字之后插入数据元素 x\n");
        int arr;
        ElemType x;
        scanf("%d%d",&arr,&x);
        Insert_List(&List,arr,x);
    }
}
```

```
}
else if(ch==2)
{
    printf("1个参数: 在顺序表 L 中, 删除索引号为 arr 的元素\n");
    int arr;
    scanf("%d",&arr);
    Delete_List(&List,arr);
}
else if(ch==3)
{
    printf("0 个参数\n");
    printf("顺序表 L 的长度为: %d\n",Seq_Length(&List));
}
else if(ch==4)
{
    printf("2 个参数: 需要逆序部分索引号的左边界和右边界\n");
    int l,r;
    scanf("%d%d",&l,&r);
    Reserve_List(&List,l,r);
}
else if(ch==5)
{
    printf("2 个参数: 需要排序的索引号左边界和右边界\n");
    int l,r;
    scanf("%d%d",&l,&r);
    Sort_List(&List,l,r);
}
else if(ch==6)
{
    SeqList a,b;
    printf("4 个参数: 请先输入顺序表 a 的长度, 再依次输入 a 中的值; 再输入顺序表 b 的长度, 再依次输入 b 中的值\n");
    int len;
    scanf("%d",&len);
    Create(&a,len);
    scanf("%d",&len);
    Create(&b,len);
    Init_List(&List);
    Merge_List(&a,&b,&List);
}
else if(ch==7)
{
    SeqList a,b;
    printf("4 个参数: 请先输入顺序表 a 的长度, 再依次输入 a 中的值; 再输入顺序表 b 的长度, 再依次输入 b 中的值\n");
```

```
    int len;
    scanf("%d",&len);
    Create(&a,len);
    scanf("%d",&len);
    Create(&b,len);
    Init_List(&List);
    Merge_Sort_List(&a,&b,&List);
}
else if(ch==8)
{
    Visit(&List);
}
else if(ch==9)
{
    printf("2 个参数: 输入需要创建顺序表的表长, 以及顺序表的每个数字\n");
    int n;
    scanf("%d",&n);
    Create(&List,n);
}
}
int main()
{
    Init_List(&List);
    while(1) fun();
    return 0;
}
```

#### 四. 运行结果:

##### 1. 使用尾插法建表(一次性输入数据)

```
1 warning generated.
1.在顺序表中插入数组元素
2.删除数组中的元素
3.求顺序表的长度
4.顺序表的逆序
5.将顺序表中的值从小到大排序
6.合并两个顺序表
7.合并两个有序的顺序表
8.输出L
9.使用尾插法一次性向List输入数据
9
2 个参数: 输入需要创建顺序表的表长, 以及顺序表的每个数字
5 1 2 3 4 5
1.在顺序表中插入数组元素
2.删除数组中的元素
3.求顺序表的长度
4.顺序表的逆序
5.将顺序表中的值从小到大排序
6.合并两个顺序表
7.合并两个有序的顺序表
8.输出L
9.使用尾插法一次性向List输入数据
8
1 2 3 4 5
```

## 2. 删除指定索引号的元素

```
1 2 3 4 5
1.在顺序表中插入数组元素
2.删除数组中的元素
3.求顺序表的长度
4.顺序表的逆序
5.将顺序表中的值从小到大排序
6.合并两个顺序表
7.合并两个有序的顺序表
8.输出L
9.使用尾插法一次性向List输入数据
2
1个参数：在顺序表L中，删除索引号为arr的元素
2
1.在顺序表中插入数组元素
2.删除数组中的元素
3.求顺序表的长度
4.顺序表的逆序
5.将顺序表中的值从小到大排序
6.合并两个顺序表
7.合并两个有序的顺序表
8.输出L
9.使用尾插法一次性向List输入数据
8
1 2 4 5
1.在顺序表中插入数组元素
```

## 3. 求顺序表的表长

```
1 2 4 5
1.在顺序表中插入数组元素
2.删除数组中的元素
3.求顺序表的长度
4.顺序表的逆序
5.将顺序表中的值从小到大排序
6.合并两个顺序表
7.合并两个有序的顺序表
8.输出L
9.使用尾插法一次性向List输入数据
3
0个参数
顺序表L的长度为：4
1.在顺序表中插入数组元素
```

## 5. 将顺序表中的值从小到大排序

```
2417 327 2389 2389 291
1.在顺序表中插入数组元素
2.删除数组中的元素
3.求顺序表的长度
4.顺序表的逆序
5.将顺序表中的值从小到大排序
6.合并两个顺序表
7.合并两个有序的顺序表
8.输出L
9.使用尾插法一次性向List输入数据
5
2个参数：需要排序的索引号左边界和右边界
0 4
1.在顺序表中插入数组元素
2.删除数组中的元素
3.求顺序表的长度
4.顺序表的逆序
5.将顺序表中的值从小到大排序
6.合并两个顺序表
7.合并两个有序的顺序表
8.输出L
9.使用尾插法一次性向List输入数据
8
291 327 2389 2389 2417
1.在顺序表中插入数组元素
```

## 6. 合并两个有序顺序表

```

7
4个参数：请先输入顺序表a的长度，再依次输入a中的值；再输入顺序表b的长度，再依次输入b中的值
4 2 4 6 8
5 3 7 9 12 15
1.在顺序表中插入数组元素
2.删除数组中的元素
3.求顺序表的长度
4.顺序表的逆序
5.将顺序表中的值从小到大排序
6.合并两个顺序表
7.合并两个有序的顺序表
8.输出L
9.使用尾插法一次性向List输入数据
8
2 3 4 6 7 8 9 12 15

```

## 7. 顺序表的逆置

```

2 3 4 6 7 8 9 12 15
1.在顺序表中插入数组元素
2.删除数组中的元素
3.求顺序表的长度
4.顺序表的逆序
5.将顺序表中的值从小到大排序
6.合并两个顺序表
7.合并两个有序的顺序表
8.输出L
9.使用尾插法一次性向List输入数据
4
2个参数：需要逆序部分索引号的左边界和右边界
0 8
1.在顺序表中插入数组元素
2.删除数组中的元素
3.求顺序表的长度
4.顺序表的逆序
5.将顺序表中的值从小到大排序
6.合并两个顺序表
7.合并两个有序的顺序表
8.输出L
9.使用尾插法一次性向List输入数据
8
15 12 9 8 7 6 4 3 2

```

## 五. 实验小结:

- (1) 顺序表结构体：SeqList 包含一个静态数组 elem 和一个指向最后一个元素的 last 指针。
- (2) 初始化顺序表：Init\_List 函数将顺序表的 last 指针设置为 -1，表示顺序表为空。
- (3) 插入元素：Insert\_List 在指定位置插入元素，先右移元素再插入新元素。
- (4) 删除元素：Delete\_List 删除指定位置的元素，并将后续元素左移以填补空缺。
- (5) 获取长度：Seq\_Length 返回顺序表的当前长度，即 last + 1。
- (6) 逆序操作：Reserve\_List 将顺序表中指定区间的元素顺序反转。
- (7) 排序操作：Sort\_List 使用快速排序算法对指定区间的元素从小到大排序。
- (8) 顺序表合并：Merge\_List 将两个顺序表合并为一个顺序表，不要求输入顺序表有序。
- (9) 合并有序顺序表：Merge\_Sort\_List 合并两个有序顺序表，保证合并后的顺序表仍然有序。

- (10) 遍历输出: `Visit` 按顺序输出顺序表中的所有元素, 如果为空则提示。
- (11) 顺序表创建: `Create` 函数使用尾插法根据输入的元素个数创建顺序表。
- (12) 主菜单功能: `fun` 函数提供一个用户交互的菜单, 执行各种顺序表操作。



## 实验三：单链表基本操作的实现

### 一. 实验内容：

构建线性表的链式存储结构，采用动态分配方式实现单链表的初始化，数据的插入，删除，输出单链表内中各元素，求单链表的长度，实现单链表中数据结点的按值排序，实现单链表的逆置，合并两个有序的单链表（有序的 a 表和有序的 b 表合并之后的结果保存在 a 表中）等操作。

### 二. 实验要求：

- (1) 要有能根据用户的输入来选择不同运算的菜单界面。
- (2) 单链表中数据元素的类型统一抽象表示为 ElemType 类型，具体类型不限，可以是整型、实型、字符型、或者是自己构造的一种结构体类型。
- (3) 实现课后习题有关单链表运算算法

### 三. 源程序：

```
#include<stdio.h>
#include<stdlib.h>
#define ElemType int
//#define CLS "cls"//win
#define CLS "clear"//macos/linux
typedef struct Node
{
    ElemType data;
    struct Node *next;
}Node,*LinkList;
Node *List;
void Init_List(Node *L);
void Insert_List(Node *L,int arr,ElemType x);
void Delete_List(Node *L,int arr);
void Visit(Node *L);
int List_Length(Node *L);
void List_Bubble_Sort(Node *L);
void List_Reserve(Node *L);
void Merge_List(Node *a,Node *b);
void swap(ElemType *a,ElemType *b);
void swap(ElemType *a,ElemType *b)
{
    ElemType tmp=*a;
    *a=*b;
    *b=tmp;
}
void Init_List(Node *L)//单链表的初始化
```

```
{
    L->next=NULL;
}

void Insert_List(Node *L,int arr,ElemType x)//向单链表L的第arr位后面插入x
{
    Node *p=L;
    int cnt=-1;
    while(p->next!=NULL&&arr!=-1)
    {
        p=p->next;
        ++cnt;
        if(cnt==arr) break;
    }
    if(cnt<arr)
    {
        printf("Insert_Out_of_Array\n");
        return;
    }
    Node *new_node=(Node*)malloc(sizeof(Node));
    new_node->data=x;
    new_node->next=p->next;
    p->next=new_node;
}

void Delete_List(Node *L,int arr)//删除单链表L中索引号为arr的结点
{
    Node *p=L;
    int cnt=0;
    while(p->next!=NULL&&arr!=0)
    {
        p=p->next;
        ++cnt;
        if(cnt==arr) break;
    }
    if(p->next==NULL)
    {
        printf("Delete_Out_of_Array\n");
        return;
    }
    Node *tmp=p->next;
    p->next=tmp->next;
    free(tmp);
}

void Visit(Node *L)//输出单链表中的所有元素
{
    Node *p=L->next;
```

```
while(p!=NULL)
{
    printf("%d ",p->data);
    p=p->next;
}
printf("\n");
}

int List_Length(Node *L)// 返回1 以L 为头节点的链表的长度
{
    Node *p=L;
    int cnt=0;
    while(p->next!=NULL)
    {
        p=p->next;
        ++cnt;
    }
    return cnt;
}

void List_Bubble_Sort(Node *L)// 对链表L 进行冒泡排序
{
    Node *p=L;
    int change=1;
    while(change)
    {
        change=0;
        p=L;
        while(p->next->next!=NULL)
        {
            p=p->next;
            if(p->data>p->next->data)
            {
                swap(&(p->data),&(p->next->data));
                change=1;
            }
        }
    }
}

void List_Reserve(Node *L)// 实现单链表的逆置
{
    if(!L->next||!(L->next->next)) return;
    Node *p=L->next->next; L->next->next=NULL;
    while(p)
    {
        Node *temp=p->next;
        p->next=L->next; L->next=p;
    }
}
```

```
        p=temp;
    }
}

void Merge_List(Node *a,Node *b)// 合并两个有序的单链表（有序的a 表和有序的b 表合并之后的结果保存在a 表中）
{
    Node *pre=a,*p=a->next,*q=b->next;
    while(p&&q)
    {
        if(p->data<q->data) {pre->next=p;pre=p;p=p->next;}
        else {pre->next=q;pre=q;q=q->next;}
    }
    pre->next=p?p:q;
    free(b);
}

void Init_Insert(Node *L,int n)
{
    Init_List(L);
    for(int i=1;i<=n;++i)
    {
        int x;
        scanf("%d",&x);
        Insert_List(L,i-2,x);
    }
}

void fun()
{
    printf("1.初始化链表 L，并使用尾插法添加 n 个元素\n");
    printf("2.在链表 L 的索引号 arr 后面插入一个数字 x\n");
    printf("3.删除链表 L 中索引号为 arr 的项\n");
    printf("4.输出单链表 L 中所有的项\n");
    printf("5.输出单链表 L 的长度\n");
    printf("6.将单链表 L 中的值按照升序排序\n");
    printf("7.将单链表中的值逆序\n");
    printf("8.合并两个有序的单链表\n");
    int ch;
    scanf("%d",&ch);
    if(ch==1)
    {
        printf("2 个参数：需要添加的元素个数，所有元素\n");
        int n;
        scanf("%d",&n);
        Init_Insert(List,n);
    }
    else if(ch==2)
```

```
{
    printf("2个参数: 在索引号 arr 后面添加元素 x\n");
    int arr,x;
    scanf("%d%d",&arr,&x);
    Insert_List(List,arr,x);
}
else if(ch==3)
{
    printf("1个参数: 删除元素的索引号 arr\n");
    int arr;
    scanf("%d",&arr);
    Delete_List(List,arr);
}
else if(ch==4)
{
    printf("无参数\n");
    Visit(List);
}
else if(ch==5)
{
    printf("无参数\n");
    printf("List 的长度为%d\n",List_Length(List));
}
else if(ch==6)
{
    printf("无参数\n");
    List_Bubble_Sort(List);
}
else if(ch==7)
{
    printf("无参数\n");
    List_Reserve(List);
}
else if(ch==8)
{
    printf("2个参数: 需要和 List 归并的升序数组 b 的长度, 以及他的所有元素\n");
    int n;
    scanf("%d",&n);
    Node *b=malloc(sizeof(Node));
    Init_Insert(b,n);
    Merge_List(List,b);
}
}
int main()
{
```

```

List=malloc(sizeof(Node));
while(1) fun();
return 0;
}

```

#### 四. 运行结果:

##### 1. 创建链表, 并添加元素 1 15 6 9 10

```

1
2个参数: 需要添加的元素个数, 所有元素
5 1 15 6 9 10
1.初始化链表L, 并使用尾插法添加n个元素
2.在链表L的索引号arr后面插入一个数字x
3.删除链表L中索引号为arr的项
4.输出单链表L中所有的项
5.输出单链表L的长度
6.将单链表L中的值按照升序排序
7.将单链表中的值逆序
8.合并两个有序的单链表
4
无参数
1 15 6 9 10

```

##### 2. 在 arr=2 的后面添加一个元素 18

```

2 2 18
2个参数: 在索引号arr后面添加元素x
1.初始化链表L, 并使用尾插法添加n个元素
2.在链表L的索引号arr后面插入一个数字x
3.删除链表L中索引号为arr的项
4.输出单链表L中所有的项
5.输出单链表L的长度
6.将单链表L中的值按照升序排序
7.将单链表中的值逆序
8.合并两个有序的单链表
4
无参数
1 15 6 18 9 10

```

##### 3. 删除 arr=4 的项

```

3
1个参数: 删除元素的索引号arr
4
1.初始化链表L, 并使用尾插法添加n个元素
2.在链表L的索引号arr后面插入一个数字x
3.删除链表L中索引号为arr的项
4.输出单链表L中所有的项
5.输出单链表L的长度
6.将单链表L中的值按照升序排序
7.将单链表中的值逆序
8.合并两个有序的单链表
4
无参数
1 15 6 18 10

```

## 4. 输出单链表 L 现在的长度

```
5
无参数
List 的长度为 5
```

## 5. 将单链表中的值逆序

```
7
无参数
1. 初始化链表 L, 并使用尾插法添加 n 个元素
2. 在链表 L 的索引号 arr 后面插入一个数字 x
3. 删除链表 L 中索引号为 arr 的项
4. 输出单链表 L 中所有的项
5. 输出单链表 L 的长度
6. 将单链表 L 中的值按照升序排序
7. 将单链表中的值逆序
8. 合并两个有序的单链表
4
无参数
10 18 6 15 1
```

## 6. 将单链表中的值, 按照升序排序

```
6
无参数
1. 初始化链表 L, 并使用尾插法添加 n 个元素
2. 在链表 L 的索引号 arr 后面插入一个数字 x
3. 删除链表 L 中索引号为 arr 的项
4. 输出单链表 L 中所有的项
5. 输出单链表 L 的长度
6. 将单链表 L 中的值按照升序排序
7. 将单链表中的值逆序
8. 合并两个有序的单链表
4
无参数
1 6 10 15 18
```

## 7. 合并两个有序的单链表, 并将合并结果存储在 List 中

```
8
2 个参数: 需要和 List 归并的升序数组 b 的长度, 以及他的所有元素
5 2 7 9 20 21
1. 初始化链表 L, 并使用尾插法添加 n 个元素
2. 在链表 L 的索引号 arr 后面插入一个数字 x
3. 删除链表 L 中索引号为 arr 的项
4. 输出单链表 L 中所有的项
5. 输出单链表 L 的长度
6. 将单链表 L 中的值按照升序排序
7. 将单链表中的值逆序
8. 合并两个有序的单链表
4
无参数
1 2 6 7 9 10 15 18 20 21
```

## 五. 实验小结:

- (1) 节点结构定义与全局变量: 定义了存储整数的链表节点结构 Node 和一个全局链表指针 List。
- (2) 基本操作函数: 提供了链表的初始化、插入、删除、访问、长度计算、排序、逆置、

合并等操作函数。

- (3) swap 函数：用于交换两个元素的值，辅助排序操作。
- (4) 链表初始化 (Init\_List)：将链表的头节点的 next 指针初始化为 NULL。
- (5) 元素插入 (Insert\_List)：在链表指定位置后插入一个新的元素，如果位置超出范围则打印错误信息。
- (6) 元素删除 (Delete\_List)：删除链表中指定索引的节点，如果索引无效则打印错误信息。
- (7) 访问链表 (Visit)：遍历并输出链表中的所有元素。
- (8) 链表长度计算 (List\_Length)：返回链表的长度。
- (9) 冒泡排序 (List\_Bubble\_Sort)：对链表元素进行升序排序，使用冒泡排序算法。
- (10) 链表逆置 (List\_Reserve)：将链表中的元素顺序反转。
- (11) 链表合并 (Merge\_List)：将两个有序链表合并成一个新的有序链表，并将结果存放在第一个链表中。
- (12) 初始化插入 (Init\_Insert)：初始化链表并按顺序插入多个元素。
- (13) 用户交互函数 (fun)：提供一个文本菜单，用户可以选择对链表进行的操作，输入相应参数以执行功能。



## 实验四：顺序栈和循环队列的基本运算

### 一. 实验内容:

构建顺序栈类型，实现顺序栈的初始化、判满、判空、入栈、出栈、读取栈顶元素运算，基于顺序栈实现表达式或文本中括号是否匹配的检验；构建循环队列类型，实现循环队列的初始化、判满、判空、入队、出队、读取队头元素、读取队尾元素的运算，基于循环队列实现杨辉三角形 N 行数据的输出。

### 二. 实验要求:

(1) 栈中数据元素的类型统一抽象表示为 SElemType 类型，在程序中将 SElemType 类型具体化为 char 类型

(2) 队列中数据元素的类型统一抽象表示为 QElemType 类型，在程序中将 QElemType 类型具体化为 int 类型

(3) 将栈和队列的基本运算分别封装在 stack.h 和 queue.h 文件中，源程序直接 #include “stack.h” 和 #include “queue.h”

(4) 提交 1 个源程序文件，2 个头文件 (stack.h 和 queue.h)。

### 注意.h 文件的生成:

(1) 先设计一程序，假设文件名为 “XXX.cpp”，包含：顺序栈类型的构建，顺序栈的基本运算，主函数，在主函数中调用测试顺序栈的基本运算，保证顺序栈的基本运算正确。

(2) 将 “XXX.cpp” 另存为 “stack.h”，另存的时候选保存类型为 “Header files”，然后删除文件里的 #include <stdio.h>, typedef int SElemType; , main() 函数等；只留下顺序栈基本运算涉及的内容（参考下图框架）后再次保存。

```
#define Stack_Size 20
#define TRUE 1
#define FALSE 0
typedef struct{
    SElemType elem[Stack_Size];
    int top;
}SeqStack;
void InitStack(SeqStack *s){
int IsEmpty(SeqStack *s){
int IsFull(SeqStack *s){
int Push(SeqStack *S,SElemType e){
int Pop(SeqStack *S,SElemType *e){
int GetTop(SeqStack *S,SElemType *e){
```

(3) 文件 “queue.h” 的生成类似上述两步。

(4) 在基于顺序栈、基于队列的应用问题解决的源程序里就可以使用 stack.h 和

queue.h 文件，参考下图：

```
#include <stdio.h>
typedef int QElemType;
#include "queue.h"
typedef int SElemType;
#include "stack.h"
void Conversion(int n,int base){
void YangHuiTriangle(int column){
int main(){
    Conversion(31,2);
    Conversion(31,8);
    Conversion(31,16);
    YangHuiTriangle(5);
    YangHuiTriangle(10);
    return 0;
}
```

十进制数31对应的二进制数为：11111

十进制数31对应的八进制数为：37

十进制数31对应的十六进制数为：1F

杨辉三角形的前5行数据为：

```

      1
     1 1
    1 2 1
   1 3 3 1
  1 4 6 4 1
```

杨辉三角形的前10行数据为：

```

      1
     1 1
    1 2 1
   1 3 3 1
  1 4 6 4 1
 1 5 10 10 5 1
1 6 15 20 15 6 1
1 7 21 35 35 21 7 1
1 8 28 56 70 56 28 8 1
1 9 36 84 126 126 84 36 9 1
```

### 三．源程序：

文件 1:Stack.h

```
typedef struct
{
    StackElementType elem[maxsize];
    int top;
}SeqStack;
void InitStack(SeqStack *s)// 顺序栈的初始化操作
{
    s->top=-1;
}
int Stack_IsEmpty(SeqStack *s)// 顺序栈的判空
{
    if(s->top==-1) return 1;
    else return 0;
```

```
}
int Stack_IsFull(SeqStack *s)// 顺序栈的判满
{
    if(s->top==maxsize-1) return 1;
    else return 0;
}
int Stack_Push(SeqStack *s,StackElementType e)// 顺序栈的入栈操作
{
    if(Stack_IsFull(s)) return 0;
    ++s->top;
    s->elem[s->top]=e;
    return 1;
}
int Stack_Pop(SeqStack *s,StackElementType *e)// 顺序栈的出栈操作
{
    if(Stack_IsEmpty(s)) return 0;
    *e=s->elem[s->top--];
    return 1;
}
int Stack_GetTop(SeqStack *s,StackElementType *e)// 读取栈顶元素
{
    if(Stack_IsEmpty(s)) return 0;
    *e=s->elem[s->top];
    return 1;
}
```

文件 2. DebugStack. c

```
#include<stdio.h>
#define maxsize 100
#define StackElementType int
#include"Stack.h"
SeqStack s;
void menu()
{
    printf("1. 实现顺序栈的初始化\n2. 顺序栈的判空\n3. 顺序栈的入栈\n4. 顺序栈的出栈\n5. 读取栈顶元素\n6. 基于顺序栈实现表达式或者文本括号匹配的检验运算\n");
    int op;
    scanf("%d",&op);
    if(op==1) InitStack(&s);
    else if(op==2)
    {
        if(Stack_IsEmpty(&s)) printf("The stack is empty\n");
        else printf("It is not an empty stack\n");
    }
    else if(op==3)
    {
```

```
    if(Stack_IsFull(&s))
    {
        printf("You cannot push number into the stack\n");
        return;
    }
    printf("Input the number that you want to push\n");
    int x;
    scanf("%d",&x);
    Stack_Push(&s,x);
}
else if(op==4)
{
    int x;
    if(Stack_Pop(&s,&x)) printf("OK\n");
    else printf("Oops!The stack is empty\n");
}
else if(op==5)
{
    int x;
    if(Stack_GetTop(&s,&x)) printf("The top of the stack is:%d\n",x);
    else printf("Oops!The stack is empty\n");
}
}
int main()
{
    while(1) menu();
    return 0;
}
```

文件 3. Queue. h

```
typedef struct
{
    QueueElementType elem[maxsize];
    int rear;
    int front;
}SeqQueue;
void InitQueue(SeqQueue *q)// 循环链表的初始化操作
{
    q->front=0;
    q->rear=0;
}
int Queue_IsEmpty(SeqQueue *q)// 循环队列的判空
{
    if(q->front==q->rear) return 1;
    else return 0;
}
```

```
int Queue_IsFull(SeqQueue *q)
{
    return (q->rear+1)%maxsize==q->rear;
}
int Queue_Push(SeqQueue *q, QueueElementType e)
{
    if(Queue_IsFull(q)) return 0;
    q->elem[q->rear]=e;
    q->rear=(q->rear+1)%maxsize;
    return 1;
}
int Queue_Pop(SeqQueue *q, QueueElementType *e)
{
    if(Queue_IsEmpty(q)) return 0;
    *e=q->elem[q->front];
    q->front=(q->front+1)%maxsize;
    return 1;
}
int Queue_Front(SeqQueue *q, QueueElementType *e)
{
    if(Queue_IsEmpty(q)) return 0;
    *e=q->elem[q->front];
    return 1;
}
int Queue_Back(SeqQueue *q, QueueElementType *e)
{
    if(Queue_IsEmpty(q)) return 0;
    *e=q->elem[(q->rear-1+maxsize)%maxsize];
    return 1;
}
```

文件 4. DebugQueue. c

```
#include<stdio.h>
#define maxsize 100
#define QueueElementType int
#include"Queue.h"
SeqQueue s;
void menu()
{
    printf("1. 循环队列的初始化\n2. 循环队列的判空\n3. 循环队列的判满\n4. 循环队列的入队\n5. 循环队列的出队\n");
    printf("6. 读取队头元素\n7. 读取队尾元素内容\n");
    int op;
    scanf("%d",&op);
    if(op==1) InitQueue(&s);
    else if(op==2)
```

```
{
    if(Queue_IsEmpty(&s)) printf("The queue is empty\n");
    else printf("It is not an empty queue\n");
}
else if(op==3)
{
    if(Queue_IsFull(&s)) printf("The queue is full\n");
    else printf("The queue is not full\n");
}
else if(op==4)
{
    if(Queue_IsFull(&s))
    {
        printf("You cannot push number into the queue\n");
        return;
    }
    printf("Input the number that you want to push\n");
    int x;
    scanf("%d",&x);
    Queue_Push(&s,x);
}
else if(op==5)
{
    int x;
    if(Queue_Pop(&s,&x)) printf("OK\n");
    else printf("The queue is empty\n");
}
else if(op==6)
{
    int x;
    if(Queue_Front(&s,&x)) printf("The front of the queue is:%d\n",x);
    else printf("Oops!The queue is empty\n");
}
else if(op==7)
{
    int x;
    if(Queue_Back(&s,&x)) printf("The back of the queue is:%d\n",x);
    else printf("Oops!The queue is empty\n");
}
}
int main()
{
    while(1) menu();
    return 0;
}
```

## 文件 5. Parentheses.c

```
#include<stdio.h>
#define StackElementType char
#define maxsize 100
#include"Stack.h"
int match(char str1,char str2)
{
    if(str1=='(' && str2==')') return 1;
    if(str1=='[' && str2==']') return 1;
    if(str1=='{' && str2=='}') return 1;
    return 0;
}
void BracketMatch(char str[])
{
    SeqStack s;
    int i;
    char ch;
    InitStack(&s);
    for(i=0;str[i]!='\0';++i)
    {
        switch(str[i])
        {
            case '(':
            case '[':
            case '{':
                Stack_Push(&s,str[i]);break;
            case ')':
            case ']':
            case '}':
                if(Stack_IsEmpty(&s))
                    {printf("右括号多余\n");return;}
                else
                {
                    Stack_GetTop(&s,&ch);
                    if(match(ch,str[i])) Stack_Pop(&s,&ch);
                    else {printf("括号类型不匹配\n");return;}
                }
                break;
        }
    }
    if (Stack_IsEmpty(&s)) printf("括号匹配成功\n");
    else printf("左括号多余\n");
}
int main()
{
```

```
    char str[100];
    scanf("%s",str);
    BracketMatch(str);
    return 0;
}
```

文件 6. Yang\_Hui\_Triangle.c

```
#include<stdio.h>
#define maxsize 100
#define QueueElementType int
#include"Queue.h"
void Yang_Hui_Triangle(int N)
{
    SeqQueue q;
    InitQueue(&q);
    Queue_Push(&q,1); // 第一行元素入队
    for(int n=2;n<=N;++n)
    {
        Queue_Push(&q,1); // 第n 行的第一个元素入队
        for(int i=1;i<=n-2;++i)
        {
            QueueElementType tmp1,tmp2;
            Queue_Pop(&q,&tmp1);
            printf("%d ",tmp1);
            Queue_Front(&q,&tmp2);
            tmp2+=tmp1;
            Queue_Push(&q,tmp2);
        }
        QueueElementType x;
        Queue_Pop(&q,&x);
        printf("%d ",x);
        Queue_Push(&q,1);
        printf("\n");
    }
    while(!Queue_IsEmpty(&q))
    {
        QueueElementType x;
        Queue_Pop(&q,&x);
        printf("%d ",x);
    }
}

int main()
{
    Yang_Hui_Triangle(5);
    return 0;
}
```



## 四. 运行结果:

对文件 DebugStack.c 而言:

当原顺序栈为空

```
1
1.实现顺序栈的初始化
2.顺序栈的判空
3.顺序栈的入栈
4.顺序栈的出栈
5.读取栈顶元素
6.基于顺序栈实现表达式或者文本括号匹配的检验运算
2
The stack is empty
```

将 1 入栈, 判断栈是否为空, 并读取栈顶元素

```
1.实现顺序栈的初始化
2.顺序栈的判空
3.顺序栈的入栈
4.顺序栈的出栈
5.读取栈顶元素
6.基于顺序栈实现表达式或者文本括号匹配的检验运算
3
Input the number that you want to push
1
1.实现顺序栈的初始化
2.顺序栈的判空
3.顺序栈的入栈
4.顺序栈的出栈
5.读取栈顶元素
6.基于顺序栈实现表达式或者文本括号匹配的检验运算
2
It is not an empty stack
1.实现顺序栈的初始化
2.顺序栈的判空
3.顺序栈的入栈
4.顺序栈的出栈
5.读取栈顶元素
6.基于顺序栈实现表达式或者文本括号匹配的检验运算
5
The top of the stack is:1
```

让 4 入栈, 再读取栈顶元素

```
1.实现顺序栈的初始化
2.顺序栈的判空
3.顺序栈的入栈
4.顺序栈的出栈
5.读取栈顶元素
6.基于顺序栈实现表达式或者文本括号匹配的检验运算
3
Input the number that you want to push
4
1.实现顺序栈的初始化
2.顺序栈的判空
3.顺序栈的入栈
4.顺序栈的出栈
5.读取栈顶元素
6.基于顺序栈实现表达式或者文本括号匹配的检验运算
5
The top of the stack is:4
```

让栈顶元素 4 出栈，再读取栈顶元素

```

1.实现顺序栈的初始化
2.顺序栈的判空
3.顺序栈的入栈
4.顺序栈的出栈
5.读取栈顶元素
6.基于顺序栈实现表达式或者文本括号匹配的检验运算
4
OK
1.实现顺序栈的初始化
2.顺序栈的判空
3.顺序栈的入栈
4.顺序栈的出栈
5.读取栈顶元素
6.基于顺序栈实现表达式或者文本括号匹配的检验运算
5
The top of the stack is:1

```

对于文件 DebugQueue.c 而言：

当原队列为空：

```

1.循环队列的初始化
2.循环队列的判空
3.循环队列的判满
4.循环队列的入队
5.循环队列的出队
6.读取队头元素
7.读取队尾元素内容
1
1.循环队列的初始化
2.循环队列的判空
3.循环队列的判满
4.循环队列的入队
5.循环队列的出队
6.读取队头元素
7.读取队尾元素内容
2
The queue is empty
1.循环队列的初始化
2.循环队列的判空
3.循环队列的判满
4.循环队列的入队
5.循环队列的出队
6.读取队头元素
7.读取队尾元素内容
3
The queue is not full
1.循环队列的初始化
2.循环队列的判空
3.循环队列的判满
4.循环队列的入队
5.循环队列的出队
6.读取队头元素
7.读取队尾元素内容

```

连续将 1 2 3 4 加入队列中(这里只展示 4 入队的操作):

```
1.循环队列的初始化
2.循环队列的判空
3.循环队列的判满
4.循环队列的入队
5.循环队列的出队
6.读取队头元素
7.读取队尾元素内容
4
Input the number that you want to push
4
```

分别读取队头和队尾的元素

```
1.循环队列的初始化
2.循环队列的判空
3.循环队列的判满
4.循环队列的入队
5.循环队列的出队
6.读取队头元素
7.读取队尾元素内容
6
The front of the queue is:1
1.循环队列的初始化
2.循环队列的判空
3.循环队列的判满
4.循环队列的入队
5.循环队列的出队
6.读取队头元素
7.读取队尾元素内容
7
The back of the queue is:4
```

队头出队，再次读取队头队尾元素

```
1.循环队列的初始化
2.循环队列的判空
3.循环队列的判满
4.循环队列的入队
5.循环队列的出队
6.读取队头元素
7.读取队尾元素内容
5
OK
1.循环队列的初始化
2.循环队列的判空
3.循环队列的判满
4.循环队列的入队
5.循环队列的出队
6.读取队头元素
7.读取队尾元素内容
6
The front of the queue is:2
1.循环队列的初始化
2.循环队列的判空
3.循环队列的判满
4.循环队列的入队
5.循环队列的出队
6.读取队头元素
7.读取队尾元素内容
7
The back of the queue is:4
```

对于文件 Parentheses.c

```
{(())}[]
```

括号匹配成功

```
{{[]})
```

括号类型不匹配

```
{(
```

左括号多余

```
()
```

右括号多余

对于文件 Yang\_Hui\_Triangle.c

```
1
1 1
1 2 1
1 3 3 1
1 4 6 4 1 %
```

```
1
1 1
1 2 1
1 3 3 1
1 4 6 4 1
1 5 10 10 5 1
1 6 15 20 15 6 1
1 7 21 35 35 21 7 1
1 8 28 56 70 56 28 8 1
1 9 36 84 126 126 84 36 9 1
1 10 45 120 210 252 210 120 45 10 1
1 11 55 165 330 462 462 330 165 55 11 1
1 12 66 220 495 792 924 792 495 220 66 12 1
1 13 78 286 715 1287 1716 1716 1287 715 286 78 13 1
1 14 91 364 1001 2002 3003 3432 3003 2002 1001 364 91 14 1
1 15 105 455 1365 3003 5005 6435 6435 5005 3003 1365 455 105 15 1
1 16 120 560 1820 4368 8008 11440 12870 11440 8008 4368 1820 560 120 16 1
1 17 136 680 2380 6188 12376 19448 24310 24310 19448 12376 6188 2380 680 136 17 1
1 18 153 816 3060 8568 18564 31824 43758 48620 43758 31824 18564 8568 3060 816 153 18 1
1 19 171 969 3876 11628 27132 50388 75582 92378 92378 75582 50388 27132 11628 3876 969 171 19 1 %
```

## 五. 实验小结:

### 1.文件 1:Stack.h

顺序栈结构: 使用数组 `elem` 和 `top` 指针实现的栈, 支持基本的栈操作 (入栈、出栈、读取栈顶元素等)。

初始化操作: `InitStack` 函数用于初始化栈, 将 `top` 设为 `-1` 表示栈为空。

判空与判满: 通过 `Stack_IsEmpty` 和 `Stack_IsFull` 函数判断栈是否为空或已满, 必要时抛出异常

入栈操作: `Stack_Push` 函数在栈不满的情况下将新元素压入栈顶, 并更新 `top` 指针。

出栈操作: `Stack_Pop` 函数在栈不空的情况下弹出栈顶元素, 更新 `top` 指针。

读取栈顶元素: `Stack_GetTop` 函数用于获取栈顶元素的值, 而不改变栈的状态。

### 2.文件 2:Queue.h

循环队列结构: 使用数组 `elem`、`front`、`rear` 实现的循环队列, 支持基本的队列操作。

初始化操作: `InitQueue` 函数初始化队列, 将 `front` 和 `rear` 都设为 `0`, 表示队列为空。

判空与判满: `Queue_IsEmpty` 和 `Queue_IsFull` 函数分别判断队列是否为空或已满, 必要时抛出异常

入队操作: `Queue_Push` 函数在队列不满时将新元素插入队尾, 并更新 `rear` 指针。

出队操作: `Queue_Pop` 函数在队列不空时从队头移出元素, 并更新 `front` 指针。

读取队头元素: `Queue_Front` 函数用于获取队头元素的值, 而不改变队列状态。

读取队尾元素: `Queue_Back` 函数用于获取队尾元素的值, 注意这里通过计算

$(rear-1+maxsize)\%maxsize$  来定位队尾。

杨辉三角形输出: 使用循环队列实现杨辉三角形的  $N$  行数据输出, 充分利用队列的特性来管理和输出每行的数值。

### 3. 文件 3: DebugStack.c

顺序栈初始化: 通过 `InitStack` 函数将栈顶指针设置为  $-1$ , 以表示栈为空。

栈判空: `Stack_IsEmpty` 函数检查栈是否为空, 即栈顶指针是否为  $-1$ 。

入栈操作: `Stack_Push` 函数将元素添加到栈顶, 并更新栈顶指针。

出栈操作: `Stack_Pop` 函数从栈顶移除元素, 并更新栈顶指针。

读取栈顶元素: `Stack_GetTop` 函数获取栈顶元素而不修改栈状态。

括号匹配检查: 通过顺序栈实现对表达式或文本中括号的匹配检查。

菜单系统: `menu` 函数提供交互式界面以执行各种栈操作, 并显示相应的结果或错误信息。

### 4. 文件 4: DebugQueue.c

循环队列初始化: 通过 `InitQueue` 函数将队列的 `front` 和 `rear` 指针都设置为  $0$ , 以初始化一个空队列。

循环队列判空: `Queue_IsEmpty` 函数检查队列是否为空, 即 `front` 和 `rear` 指针是否相等。

循环队列判满: `Queue_IsFull` 函数检查队列是否已满, 即  $(rear + 1) \% maxsize$  是否等于 `front`。

循环队列入队: `Queue_Push` 函数将元素添加到队尾, 并更新 `rear` 指针。

循环队列出队: `Queue_Pop` 函数从队头移除元素, 并更新 `front` 指针。

读取队头元素: `Queue_Front` 函数获取队头元素而不修改队列状态。

读取队尾元素: `Queue_Back` 函数获取队尾元素而不修改队列状态。

### 5. 文件 5: Parentheses.c

`match` 函数: 检查两个字符是否是匹配的括号对, 返回  $1$  表示匹配,  $0$  表示不匹配。

`BracketMatch` 函数: 使用顺序栈来检查括号匹配。遇到左括号时入栈, 遇到右括号时检查栈顶元素是否匹配, 若不匹配或栈空则输出错误信息。匹配成功则输出“括号匹配成功”。

`main` 函数: 读取一个括号字符串并调用 `BracketMatch` 函数来验证括号的匹配情况。

### 6. 文件 6: Yang\_Hui\_Triangle.c

`Yang_Hui_Triangle` 函数: 使用循环队列输出杨辉三角形的前  $N$  行。每行开始时入队  $1$ , 然后计算并入队每行的中间元素, 最后再入队  $1$ 。

入队和出队: 每行开始时将  $1$  入队。计算每行的中间元素时, 从队列中取出元素进行加法运算, 然后将结果入队。

打印: 在处理每一行后, 打印当前行的元素。处理完所有行后, 输出队列中的剩余元素。

`main` 函数: 调用 `Yang_Hui_Triangle` 函数生成并输出杨辉三角形的前  $100$  行。

## 实验五：串的模式匹配

### 一. 实验内容：

构建串的定长顺序存储结构；实现串的创作，串的访问输出；实现模式串和主串的简单匹配算法和 kmp 模式匹配算法。

### 二. 实验要求：

要有能根据用户的输入来选择不模式匹配算法的菜单界面。

1. 需要设计求模式串的 next 数组的算法
2. 数组的下标从 0 开始，而逻辑序号是从 1 开始

### 三. 源程序：

```
// 构建串的定长顺序存储结构；实现串的创作，串的访问输出；实现模式串和主串的简单匹配算法和 kmp 模式匹配算法。
#include<stdio.h>
#include<string.h>
#define maxlength 100
typedef struct
{
    char ch[maxlength];
    int len;
}SString;
void get_next(SString *str,int nxt[])
{
    int i=1,j=0;
    nxt[0]=-1;nxt[1]=0;
    while(i<str->len)
    {
        if(j== -1 || str->ch[i]==str->ch[j]) {++i;++j;nxt[i]=j;}
        else j=nxt[j];
    }
}
int KMP_Pair(SString *str,SString *substr,int nxt[])
{
    int i=0,j=0;
    while(i<str->len)
    {
        if(str->ch[i]==substr->ch[j]) ++i,++j;
        else j=nxt[j];
        if(j==substr->len) return i-j+1;
    }
}
```

```
}
    return -1;
}

int Easy_Pair(SString *str, SString *substr)
{
    int i=0, j=0;
    while(i<str->len && j<substr->len)
    {
        if(str->ch[i]==substr->ch[j]) ++i, ++j;
        else {i=i-j+1; j=0;}
        if(j==substr->len) return i-j+1;
    }
    return -1;
}

int get_str(SString *str)
{
    scanf("%s", str->ch);
    str->len=strlen(str->ch);
    return 1;
}

void print_str(SString *str)
{
    printf("%s", str->ch);
}

int main()
{
    int nxt[100];
    SString str, substr;
    int ch;
    printf("1.简单模式匹配\n2.KMP 模式匹配\n3.输入一个串并输出\n");
    scanf("%d", &ch);
    if(ch==1)
    {
        printf("Input the main string\n");
        get_str(&str);
        printf("Input the sub str\n");
        get_str(&substr);
        int ans=Easy_Pair(&str, &substr);
        if(ans==-1) printf("Error\n");
        else printf("The start of the string is %d", ans);
    }
    else if(ch==2)
    {
        printf("Input the main string\n");
        get_str(&str);
```

```

    printf("Input the sub str\n");
    get_str(&substr);
    int nxt[100];
    get_next(&substr,nxt);
    int ans=KMP_Pair(&str,&substr,nxt);
    if(ans==-1) printf("Error\n");
    else printf("The start of the string is %d",ans);
}
else if(ch==3)
{
    get_str(&str);
    print_str(&str);
}
return 0;
}

```

#### 四. 运行结果:

##### 1.kmp 匹配

```

1.简单模式匹配
2.KMP模式匹配
3.输入一个串并输出
2
Input the main string
ababc
Input the sub str
abc
The start of the string is 3%

```

##### 2.简单匹配

```

1.简单模式匹配
2.KMP模式匹配
3.输入一个串并输出
1
Input the main string
sdfghgh
Input the sub str
ghg
The start of the string is 4%

```

##### 3.输入一个串再输出

```

1.简单模式匹配
2.KMP模式匹配
3.输入一个串并输出
3
abcdefg
abcdefg%

```



## 五. 实验小结:

- (1) 该代码定义了定长顺序存储结构 `SString`，用于存储和处理字符串。
- (2) 实现了基本的字符串输入输出功能，包括 `get_str` 函数用于读取字符串，`print_str` 函数用于输出字符串。
- (3) 提供了简单模式匹配算法 `Easy_Pair` 和 KMP 模式匹配算法 `KMP_Pair`，可以在主串中查找子串的位置。
- (4) `get_next` 函数生成 KMP 算法所需的 `next` 数组，帮助优化子串匹配过程。
- (5) 主函数根据用户选择执行相应的字符串操作，包括简单匹配、KMP 匹配和字符串的输入输出。

## 实验六：三元组顺序表压缩存储结构的稀疏矩阵的运算

### 一. 实验内容：

构建矩阵的三元组顺序表压缩存储结构；实现三元组顺序表压缩存储结构的矩阵的创建、矩阵的输出、矩阵的简单转置和快速转置算法、以及两个矩阵的相加。

### 二. 实验要求：

(1) 创建矩阵时，只能输入原始的二维矩阵（元素的值有 0 有非 0），存储时以三元组顺序表压缩存储结构来存储。比如，6 行 7 列的二维矩阵 M 的输入如下：

(2) 输出三元组顺序表压缩存储的矩阵时，要将矩阵所有的元素（有 0 有非 0）按行按列输出。比如，M 的转置矩阵 N（7 行 6 列）的输出如下：

A+B=C  
0 0 1 0    0 2 2 0    0 2 3 0  
0 2 0 0    3 1 0 0    3 3 0 0  
3 0 0 0    0 0 5 0    3 0 5 0

### 三. 源程序：

```
#include<stdio.h>
#include<string.h>
#define ElemType int
#define MAXSIZE 100
typedef struct
{
    int row,col;
    ElemType e;
}Triple;
typedef struct
{
    Triple data[MAXSIZE+1];
    int m,n,len;
}TSMarix;
void init(TSMarix *table)
{
    table->len=0;
}
void build(TSMarix *table)
{
    printf("输入矩阵的行数和列数\n");
```

```
scanf("%d%d",&(table->m),&(table->n));
for(int i=1;i<=table->m;++i)
{
    for(int j=1;j<=table->n;++j)
    {
        int e;
        scanf("%d",&e);
        if(e==0) continue;
        table->len++;
        table->data[table->len].row=i;
        table->data[table->len].col=j;
        table->data[table->len].e=e;
    }
}
}

void output(TSMarix *table)
{
    int t=1;
    for(int i=1;i<=table->m;++i)
    {
        for(int j=1;j<=table->n;++j)
        {
            if(table->data[t].row==i&&table->data[t].col==j)
                printf("%d ",table->data[t++].e);
            else
                printf("0 ");
        }
        printf("\n");
    }
}

void add(TSMarix *table1,TSMarix *table2,TSMarix *ans)
{
    if((table1->m!=table2->m)|| (table1->n!=table2->n))
    {
        printf("Error\n");
        return;
    }
    ans->m=table1->m;
    ans->n=table1->n;
    init(ans);
    int t1=1,t2=1,t=1;
    for(int i=1;i<=table1->m;++i)
    {
        for(int j=1;j<=table1->n;++j)
        {
```

```
        int v=0;
        if(table1->data[t1].row==i&&table1->data[t1].col==j)
            v+=table1->data[t1++].e;
        if(table2->data[t2].row==i&&table2->data[t2].col==j)
            v+=table2->data[t2++].e;
        if(v==0) continue;
        ans->len=t;
        ans->data[t].row=i;
        ans->data[t].col=j;
        ans->data[t++].e=v;
    }
}
output(ans);
}
void quick_tranpose(TSMarix *table,TSMarix *ans)
{
    int col,q;
    int num[MAXSIZE],position[MAXSIZE];
    ans->len=table->len;
    ans->n=table->m;
    ans->m=table->n;
    init(ans);
    memset(num,0,sizeof num);
    for(int i=1;i<=table->len;++i) ++num[table->data[i].col]; //统计出现次数
    position[1]=1;
    for(int i=2;i<=table->n;++i) position[i]=position[i-1]+num[i-1];
    for(int i=1;i<=table->len;++i)
    {
        col=table->data[i].col;
        q=position[col];
        ans->data[q].row=table->data[i].col;
        ans->data[q].col=table->data[i].row;
        ans->data[q].e=table->data[i].e;
        ++position[col];
    }
}
void tranpose(TSMarix *table,TSMarix *ans)
{
    int t=0;
    ans->len=table->len;
    ans->m=table->n;
    ans->n=table->m;
    for(int i=1;i<=table->n;++i)
    {
        for(int j=1;j<=table->len;++j)
```

```
{
    if(table->data[j].col==i)
    {
        ans->data[++t].col=table->data[j].row;
        ans->data[t].row=table->data[j].col;
        ans->data[t].e=table->data[j].e;
    }
}
}

void menu()
{
    printf("1.读入矩阵直接输出\n");
    printf("2.读入矩阵，转置输出\n");
    printf("3.读入矩阵，快速转置输出\n");
    printf("4.读入 2 个矩阵，相加后输出\n");
    int op;
    scanf("%d",&op);
    if(op==1)
    {
        TSMarix table;
        init(&table);
        build(&table);
        output(&table);
    }
    else if(op==2)
    {
        TSMarix table,ans;
        init(&table);
        init(&ans);
        build(&table);
        tranpose(&table,&ans);
        output(&ans);
    }
    else if(op==3)
    {
        TSMarix table,ans;
        init(&table);
        init(&ans);
        build(&table);
        quick_tranpose(&table,&ans);
        output(&ans);
    }
    else if(op==4)
    {
```

```

    TSMarix table1,table2,ans;
    init(&table1);
    init(&table2);
    init(&ans);
    build(&table1);
    build(&table2);
    add(&table1,&table2,&ans);
}
}
int main()
{
    while(1) menu();
    return 0;
}

```

#### 四. 运行结果:

##### 1.读入一个矩阵，直接输出

```

1.读入矩阵直接输出
2.读入矩阵，转置输出
3.读入矩阵，快速转置输出
4.读入2个矩阵，相加后输出
1
输入矩阵的行数和列数
3 4
1 2 0 8
2 3 4 1
2 3 4 0
1 2 0 8
2 3 4 1
2 3 4 0

```

##### 2.读入一个矩阵，简单转置输出

```

1.读入矩阵直接输出
2.读入矩阵，转置输出
3.读入矩阵，快速转置输出
4.读入2个矩阵，相加后输出
2
输入矩阵的行数和列数
3 4
1 2 0 8
2 3 4 1
2 3 4 0
1 2 2
2 3 3
0 4 4
8 1 0

```

##### 3.读入一个矩阵，快速转置后输出

- 1.读入矩阵直接输出
- 2.读入矩阵，转置输出
- 3.读入矩阵，快速转置输出
- 4.读入2个矩阵，相加后输出

3

输入矩阵的行数和列数

3 4

1 2 0 8

2 3 4 1

2 3 4 0

1 2 2

2 3 3

0 4 4

8 1 0

- 4.读入两个矩阵，向加后输出

- 1.读入矩阵直接输出
- 2.读入矩阵，转置输出
- 3.读入矩阵，快速转置输出
- 4.读入2个矩阵，相加后输出

4

输入矩阵的行数和列数

3 4

1 2 0 8

2 3 4 1

2 3 4 0

输入矩阵的行数和列数

3 4

2 3 1 0

0 8 6 2

3 4 1 2

3 5 1 8

2 11 10 3

5 7 5 2

4.读入矩阵直接输出

- 5.读入两个矩阵，合并出错

- 1.读入矩阵直接输出
- 2.读入矩阵，转置输出
- 3.读入矩阵，快速转置输出
- 4.读入2个矩阵，相加后输出

4

输入矩阵的行数和列数

2 3

1 2 3 4 5 6

输入矩阵的行数和列数

3 2

1 2 3 4 5 6

Error

## 五. 实验小结:

- (1) 矩阵存储: 代码使用三元组顺序表压缩存储结构来表示稀疏矩阵, 支持高效存储和操作非零元素。
- (2) 基本操作: 实现了矩阵的创建、输出、转置(简单和快速两种)以及矩阵相加操作。
- (3) 转置方法: 包括简单转置(按列遍历)和快速转置(利用列频率优化), 适用于稀疏矩阵的高效操作。

(4) 用户交互：通过菜单选择执行不同的矩阵操作，支持多种功能的灵活调用。



## 实验六：二叉树的基本运算

### 一. 实验内容：

构建二叉树的二叉链表存储结构，实现二叉树的创建、二叉树的先序/中序/后序递归遍历、统计二叉树的高度、统计各类结点的个数、先序/中序非递归遍历、层序遍历等运算。

### 二. 实验要求：

- (1) 二叉树中数据元素的类型统一抽象表示为 TElemType 类型, 在程序中将 TElemType 类型具体化为 char 类型
- (2) 基于栈实现二叉树的先序/中序非递归遍历 #include "stack.h"
- (3) 基于队列实现二叉树的层序遍历 #include "queue.h"

### 三. 源程序：

文件 1:queue.h

```
typedef struct
{
    QueueElementType elem[maxsize];
    int rear;
    int front;
}SeqQueue;

void InitQueue(SeqQueue *q) // 循环链表的初始化操作
{
    q->front=0;
    q->rear=0;
}

int Queue_IsEmpty(SeqQueue *q) // 循环队列的判空
{
    if(q->front==q->rear) return 1;
    else return 0;
}

int Queue_IsFull(SeqQueue *q)
{
    return (q->rear+1)%maxsize==q->rear;
}

int Queue_Push(SeqQueue *q, QueueElementType e)
{
    if(Queue_IsFull(q)) return 0;
    q->elem[q->rear]=e;
    q->rear=(q->rear+1)%maxsize;
    return 1;
}
```

```
}
int Queue_Pop(SeqQueue *q, QueueElementType *e)
{
    if(Queue_IsEmpty(q)) return 0;
    *e=q->elem[q->front];
    q->front=(q->front+1)%maxsize;
    return 1;
}
int Queue_Front(SeqQueue *q, QueueElementType *e)
{
    if(Queue_IsEmpty(q)) return 0;
    *e=q->elem[q->front];
    return 1;
}
int Queue_Back(SeqQueue *q, QueueElementType *e)
{
    if(Queue_IsEmpty(q)) return 0;
    *e=q->elem[(q->rear-1+maxsize)%maxsize];
    return 1;
}
```

文件 2:stack.h

```
typedef struct
{
    StackElementType elem[maxsize];
    int top;
}SeqStack;
void InitStack(SeqStack *s) // 顺序栈的初始化操作
{
    s->top=-1;
}
int Stack_IsEmpty(SeqStack *s) // 顺序栈的判空
{
    if(s->top==-1) return 1;
    else return 0;
}
int Stack_IsFull(SeqStack *s) // 顺序栈的判满
{
    if(s->top==maxsize-1) return 1;
    else return 0;
}
int Stack_Push(SeqStack *s, StackElementType e) // 顺序栈的入栈操作
{
    if(Stack_IsFull(s)) return 0;
    ++s->top;
    s->elem[s->top]=e;
}
```

```
    return 1;
}
int Stack_Pop(SeqStack *s, StackElementType *e) // 顺序栈的出栈操作
{
    if(Stack_IsEmpty(s)) return 0;
    *e=s->elem[s->top--];
    return 1;
}
int Stack_GetTop(SeqStack *s, StackElementType *e) // 读取栈顶元素
{
    if(Stack_IsEmpty(s)) return 0;
    *e=s->elem[s->top];
    return 1;
}
```

文件 3:BiTree.c

```
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
#define maxsize 100
#define TElemType char
typedef struct BiTree
{
    TElemType data;
    struct BiTree *lchild,*rchild;
}BiTree;
#define StackElementType BiTree*
#define QueueElementType BiTree*
#include"queue.h"
#include"stack.h"
BiTree *tree;
int max(int a,int b)
{
    return a>b?a:b;
}
BiTree *build() // 使用先序遍历创建一棵树
{
    char str;
    str=getchar();
    if(str=='#') return NULL;
    BiTree *new_node=(BiTree*)malloc(sizeof(BiTree));
    if(new_node==NULL)
    {
        printf("申请空间失败");
        return NULL;
    }
}
```

```
    new_node->data=str;
    new_node->lchild=build();
    new_node->rchild=build();
    return new_node;
}

void pre_recursion(BiTree *tree)// 先序遍历tree
{
    if(tree==NULL) return;
    printf("%c",tree->data);
    pre_recursion(tree->lchild);
    pre_recursion(tree->rchild);
}

void mid_recursion(BiTree *tree)// 中序遍历tree
{
    if(tree==NULL) return;
    mid_recursion(tree->lchild);
    printf("%c",tree->data);
    mid_recursion(tree->rchild);
}

void post_recursion(BiTree *tree)// 后序遍历tree
{
    if(tree==NULL) return;
    post_recursion(tree->lchild);
    post_recursion(tree->rchild);
    printf("%c",tree->data);
}

int height(BiTree *tree)// 统计二叉树的高度
{
    if(tree==NULL) return 0;
    return max(height(tree->lchild),height(tree->rchild))+1;
}

void cnt(BiTree *tree,int t[])
{
    if(tree->lchild==tree->rchild&&tree->lchild==NULL) ++t[0];
    else if(tree->rchild!=NULL&&tree->lchild!=NULL) ++t[2];
    else ++t[1];
    if(tree->lchild!=NULL) cnt(tree->lchild,t);
    if(tree->rchild!=NULL) cnt(tree->rchild,t);
}

void Inorder(BiTree *root)// 中序遍历
{
    SeqStack stack;
    InitStack(&stack);
    BiTree *p=root;
    while(p!=NULL || !Stack_IsEmpty(&stack))
```

```
{
    if(p!=NULL)
    {
        Stack_Push(&stack,p);
        p=p->lchild;
    }
    else
    {
        Stack_Pop(&stack,&p);
        printf("%c",p->data);
        p=p->rchild;
    }
}
}

void Preorder(BiTree *root)//先序遍历
{
    SeqStack stack;
    InitStack(&stack);
    BiTree *p=root;
    while(p!=NULL || !Stack_IsEmpty(&stack))
    {
        if(p!=NULL)
        {
            printf("%c",p->data);
            Stack_Push(&stack,p);
            p=p->lchild;
        }
        else
        {
            Stack_Pop(&stack,&p);
            p=p->rchild;
        }
    }
}

void Levelorder(BiTree *root)//层序遍历
{
    SeqQueue queue;
    InitQueue(&queue);
    BiTree *p=root;
    Queue_Push(&queue,root);
    while(!Queue_IsEmpty(&queue))
    {
        Queue_Pop(&queue,&p);
        if(p!=NULL)
        {
```

```
        printf("%c",p->data);
        Queue_Push(&queue,p->lchild);
        Queue_Push(&queue,p->rchild);
    }
}
}
int menu()
{
    printf("1.退出\n2.二叉树的先序/中序/后序递归遍历\n3.统计二叉树的高度\n4.统计各类结点的个数\n5.先序/中序非递归遍历\n6.层序遍历等运算\n");
    int op;
    scanf("%d",&op);
    if(op==1) return 0;
    else if(op==2)
    {
        printf("基于递归的先序遍历:");
        pre_recursion(tree);
        printf("\n 基于递归的中序遍历:");
        mid_recursion(tree);
        printf("\n 基于递归的后序遍历:");
        post_recursion(tree);
        printf("\n");
    }
    else if(op==3) printf("树的高度为:%d\n",height(tree));
    else if(op==4)
    {
        int c[3];
        memset(c,0,sizeof c);
        cnt(tree,c);
        printf("入度为 0:%d\n 入度为 1:%d\n 入度为 2:%d\n",c[0],c[1],c[2]);
    }
    else if(op==5)
    {
        printf("基于非递归的先序遍历:");
        Preorder(tree);
        printf("\n 基于非递归的中序遍历:");
        Inorder(tree);
        printf("\n");
    }
    else if(op==6)
    {
        printf("层序遍历:");
        Levelorder(tree);
        printf("\n");
    }
}
```

```

    }
    return 1;
}
int main()
{
    tree=build();
    int v=1;
    while(v) v=menu();
    return 0;
}

```

#### 四. 运行结果:

##### 1.基于递归的二叉树遍历

```
ABD##EH###CF#I##G##
```

- 1.退出
- 2.二叉树的先序/中序/后序递归遍历
- 3.统计二叉树的高度
- 4.统计各类结点的个数
- 5.先序/中序非递归遍历
- 6.层序遍历等运算

2

基于递归的先序遍历:ABDEHCFIG

基于递归的中序遍历:DBHEAFICG

基于递归的后序遍历:DHEBIFGCA

4.退出

##### 2.统计二叉树的高度

- 1.退出
- 2.二叉树的先序/中序/后序递归遍历
- 3.统计二叉树的高度
- 4.统计各类结点的个数
- 5.先序/中序非递归遍历
- 6.层序遍历等运算

3

树的高度为:4

##### 3.统计各类结点的个数

- 1.退出
- 2.二叉树的先序/中序/后序递归遍历
- 3.统计二叉树的高度
- 4.统计各类结点的个数
- 5.先序/中序非递归遍历
- 6.层序遍历等运算

4

入度为0:4

入度为1:2

入度为2:3

4.退出

## 5. 基于栈的非递归先序中序遍历

```

1.退出
2.二叉树的先序/中序/后序递归遍历
3.统计二叉树的高度
4.统计各类结点的个数
5.先序/中序非递归遍历
6.层序遍历等运算
5
基于非递归的先序遍历:ABDEHCFIG
基于递归的中序遍历:DBHEAFICG

```

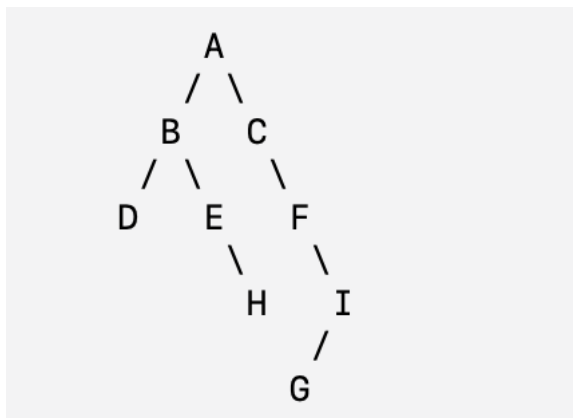
## 6. 基于递归的层序遍历

```

1.退出
2.二叉树的先序/中序/后序递归遍历
3.统计二叉树的高度
4.统计各类结点的个数
5.先序/中序非递归遍历
6.层序遍历等运算
6
层序遍历:ABCDEFGHI

```

\*以上测试的输入先序遍历为: ABD##EH###CF#I##G##  
此树的结构如下图所示:



\*可供测试的样例:

AB#D###C##E##

A#B##C#D#E##

A#B#C#D#E#F###G##

A#B#C#D#E#F#G##

这是 4 种不同形状的二叉树

## 六. 实验小结:

- 在本次实验的过程中, 我遇到了如下的问题: 由于在前面顺序表的实验中, 我封装的 stack.h 头文件中的 pop 函数, 没有将出栈元素给赋值给形参 e, 导致出栈访问一直存在问题。另外树的非递归遍历的写法也较为复杂, 我没有能够很好的记忆。



- 在本次实验的过程中，让我对线性数据结构栈和队列有了更深入的了解，让我了解到栈和队列在整个数据结构体系中的重要性——线性的数据结构是非线性数据结构的基础。
- 二叉树构建：使用先序遍历构建二叉树，# 表示空节点。
- 遍历方法：
  - 递归遍历：
    - 先序遍历 (pre\_recursion)
    - 中序遍历 (mid\_recursion)
    - 后序遍历 (post\_recursion)
  - 非递归遍历：
    - 先序遍历 (Preorder)
    - 中序遍历 (Inorder)
- 统计功能：
  - 计算二叉树的高度 (height)
  - 统计节点的入度（叶子节点、度为 1 的节点、度为 2 的节点）(cnt)
- 层序遍历：使用队列实现层序遍历 (Levelorder)
- 菜单功能：通过 menu() 函数提供操作选项，包括创建树、执行各种遍历、统计树高度和节点个数、以及层序遍历。
- 主函数：在 main() 中创建树并循环显示菜单，直到用户选择退出。

## 实验七：

一．实验内容：

二．实验要求：

三．源程序：

四．运行结果：

五．实验小结：