

线性数据结构

3.1 顺序表基本操作

◇3.1.1 实验目的及要求

- (1) 熟悉线性表的定义和基本操作。
- (2) 掌握线性表的顺序存储结构设计 with 基本操作实现。
- (3) 学会使用顺序表解决实际问题。

◇3.1.2 实验内容

基于顺序表,编写一个学生信息管理程序,实现以下功能。

- (1) 每个学生的信息包含学号、姓名。
- (2) 插入一条学生信息时,可以按学号从小到大顺序插入。
- (3) 可以根据学号删除学生信息。
- (4) 可以输出所有学生信息。

◇3.1.3 实验原理

1. 顺序表的定义

线性表是由有限个同类型的数据元素组成的有序序列,一般记作 (a_1, a_2, \dots, a_n) 。除了 a_1 和 a_n 之外,任意元素 a_i 都有一个直接前驱 a_{i-1} 和一个直接后继 a_{i+1} 。 a_1 无前驱, a_n 无后继。

采用顺序存储结构的线性表称为顺序表,它的数据元素按照逻辑顺序依次存放在一组连续的存储单元中。如图 3-1 所示用数组实现顺序表。

注意: 顺序表中的元素一定是连续存放的,中间没有空隙。

2. 顺序表的基本操作

判断顺序表是否为空只需判断 length 是否为零即可。顺序表的主要算法是在顺序

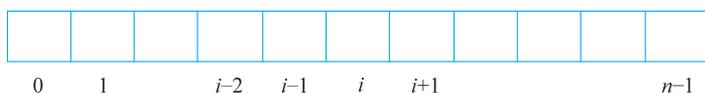


图 3-1 顺序表存储结构示意图

表中插入元素、删除元素等。

1) 在表中第 i 个位置插入新元素 x 算法实现的主要步骤如下。

(1) 判断插入位置的合理性以及表是否已满。

(2) 从最后一个元素开始依次向前,将每个元素向后移动一个位置,直到第 i 个元素为止。

(3) 向空出的第 i 个位置存入新元素 x 。

(4) 最后还要将线性表长度加 1。

2) 在表中删除第 i 个元素

算法实现的主要步骤如下。

(1) 判断删除位置的合理性。

(2) 从第 $i+1$ 个元素开始,依次向后直到最后一个元素为止,将每个元素向前移动一个位置。这时第 i 个元素已经被覆盖删除。

(3) 最后还要将线性表长度减 1。

3. 有序表的插入算法

如果线性表中的数据有序排列,则称为有序表。有序表中元素插入之前,需要先跟有序表中的元素逐个进行对比,以找到合适的插入位置。以下面的例子说明从小到大顺序插入数据的过程。

假设插入数据 5、7、2、3、9

第一步,顺序表空,插入 5

//[5]

第二步,因为 $7 > 5$,向下一个位置移动,因为到达尾部,所以插入 7

//[5 7]

第三步,因为 $2 < 5$,在当前位置插入 2

//[2 5 7]

第四步,因为 $3 > 2$,后移;因为 $3 < 5$,在当前位置插入 3

//[2 3 5 7]

第五步,因为 $9 > 2$,后移; $9 > 3$,后移; $9 > 5$,后移; $9 > 7$,后移;

因为到达尾部,所以插入 9

//[2 3 5 7 9]

◇3.1.4 实验步骤

(1) 定义顺序表中数据元素: 学生信息结构体 `stu`。

(2) 定义顺序表 `SeqList`。

(3) 实现顺序表基本操作函数,包括插入元素(`Insert`)、删除元素(`Delete`)、显示元素(`Display`)。

(4) 主函数对相应函数进行调用,实现根据用户输入,有序插入多名学生信息,删

除其中若干名学生信息,并显示最终学生信息名单。

◇3.1.5 参考代码

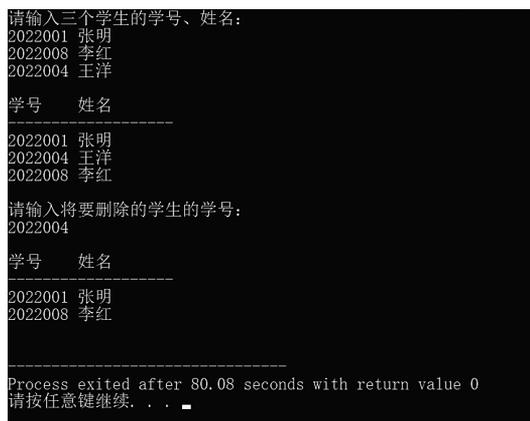
参考代码如下。

```
1. #include<stdio.h>
2.
3. const int maxsize = 100;           //顺序表最大允许长度
4. //定义数据元素
5. struct stu {
6.     int id;                         //学号
7.     char name[20];                 //姓名
8. };
9. //定义顺序表
10. struct SeqList {
11.     stu data[maxsize];             //顺序表存储数组的地址
12.     int length;                    //顺序表当前长度
13. };
14. //插入元素
15. void Insert(SeqList *L, stu x)
16. {
17.     int j = L->length - 1;         //j 为表尾下标
18.     if(L->length > 0 && L->length < maxsize)
19.     {
20.         while (L->data[j].id > x.id && j >= 0)
21.         {
22.             L->data[j + 1] = L->data[j]; //元素依次向后移动
23.             j--;
24.         }
25.     }
26.     L->data[j + 1] = x;             //存入新元素 x
27.     L->length++;                    //表长度加 1
28. }
29. //删除元素
30. void Delete(SeqList *L, int ID)
31. {
32.     int j = 0;                       //j 为元素下标
33.     while (L->data[j].id != ID && j < L->length)
34.         j++;
35.     if(j >= L->length)
36.         printf("无此元素!");
37.     else {
38.         for (int i = j; i < L->length - 1; i++)
39.         {
40.             L->data[i] = L->data[i + 1];
41.         }
42.         L->length--;
43.     }
44. }
```

```
45. void Display(SeqList * L)
46. {
47.     printf("学号\t姓名\n");
48.     for (int i = 0; i < L->length; i++)
49.         printf("%d\t%s\n", L->data[i].id, L->data[i].name);
50. }
51. int main()
52. {
53.     SeqList L;
54.     L.length = 0;
55.     stu s;
56.     printf("请输入三个学生的学号、姓名: \n");
57.     for (int i = 0; i < 3; i++)
58.     {
59.         scanf("%d %s", &s.id, &s.name);
60.         Insert(&L, s);
61.     }
62.     printf("\n");
63.     Display(&L); //显示名单
64.     int ID;
65.     printf("请输入将要删除的学生的学号: \n");
66.     scanf("%d", &ID);
67.     Delete(&L, ID);
68.     printf("\n");
69.     Display(&L); //显示名单
70.     return 0;
71. }
```

◇3.1.6 实验结果

(1) 程序运行结果如图 3-2 所示。



```
请输入三个学生的学号、姓名:
2022001 张明
2022008 李红
2022004 王洋

学号 姓名
-----
2022001 张明
2022004 王洋
2022008 李红

请输入将要删除的学生的学号:
2022004

学号 姓名
-----
2022001 张明
2022008 李红

-----
Process exited after 80.08 seconds with return value 0
请按任意键继续. . .
```

图 3-2 程序运行结果

(2) 算法复杂度分析。

对于存取操作,顺序表可随机存取,按位置访问元素的时间复杂度为 $O(1)$;而在插

入和删除操作中,平均约移动表中一半的元素,故时间复杂度为 $O(n)$ 。

◇3.1.7 实验总结

实验中使用顺序表实现学生信息管理程序,在存取操作时时间复杂度为 $O(1)$,插入和删除操作中,时间复杂度为 $O(n)$,因此线性表比较适合元素个数比较稳定、不轻易插入和删除元素、更多的操作是存取数据的应用。

3.2 单链表基本操作

◇3.2.1 实验目的及要求

- (1) 掌握线性表的链式存储结构设计 with 基本操作实现。
- (2) 理解顺序表和链表的不同和特点。
- (3) 学会使用链表解决实际问题。

◇3.2.2 实验内容

基于单链表,编写一个通讯录管理程序,实现以下功能。

- (1) 每个联系人的信息包含姓名、电话。
- (2) 可以插入新的联系人信息。
- (3) 可以根据姓名删除联系人信息。
- (4) 可以显示通讯录中联系人信息。

◇3.2.3 实验原理

1. 单链表的定义

单链表用一组地址任意的存储单元存放线性表中的数据元素。由于逻辑上相邻的元素其物理位置不一定相邻,为了建立元素间的逻辑关系,需要在线性表的每个元素中附加其后继元素的地址信息。这种地址信息称为指针。附加了其他元素指针的数据元素称为结点(如图 3-3 所示),每个结点都包含数据域和指针域两部分。结点的形式定义如下:

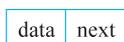


图 3-3 单链表的结点

```
typedef struct NODE
{
    datatype data;           //数据域
```

```
Node * next;           //指针域
}Node;
```

这个定义是自引用类型的。换言之,每个结点都包含另一个同类型结点的地址。单链表就是由这样定义的结点依次连接而成的单向链式结构,如图 3-4 所示。图 3-4 中结点内指向后一结点的箭头代表当前结点指针域存储的正是箭头所指结点的地址。由于最后一个元素无后继,因而其指针域为空(NULL)。另外,为了能顺次访问每个结点,需要保存单链表第一个结点的存储地址。这个地址称为线性表的头指针,我们用 head 表示。为了操作上的方便,可以在单链表的头部增加一个特殊的头结点。头结点的类型与其他结点一样,只是头结点的数据域为空。增加头结点避免了在删除或添加第一个位置的元素时进行的特殊程序处理。图 3-4 为带头结点的单链表。



图 3-4 带头结点的单链表

单链表在存储区的物理状态如图 3-5 所示。head 中存放头结点地址,根据后续结点的指针可以顺次访问所有结点的数据。

2. 单链表的基本操作

单链表的主要算法包括插入结点、删除结点、查询结点等。

1) 在表中第 i 个位置插入新结点 x

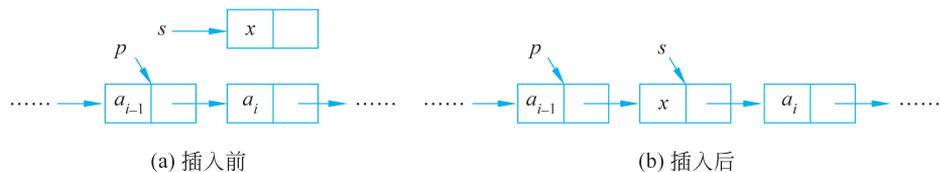
算法实现的主要步骤如下。

- (1) 首先找到第 $i-1$ 个结点的指针 p 。
- (2) 建立新结点 s 并通过语句 $s \rightarrow next = p \rightarrow next$ 将其指针指向第 i 个结点。
- (3) 通过语句 $p \rightarrow next = s$ 将第 $i-1$ 个结点的指针指向新结点。

图 3-6 给出了插入新结点前后链表指针的变化。

存储地址	数据域	数据域
22	a_2	86
...
head 38		94
...
86	a_3	NULL
94	a_1	22

图 3-5 单链表存储结构示意图

图 3-6 在单链表中插入结点 x

2) 从表中删除第 i 个结点

算法实现的主要步骤如下。

- (1) 如果第 i 个结点存在则找到第 i 个和第 $i-1$ 个结点的指针 p 和 q 。
- (2) 通过语句 $q \rightarrow next = p \rightarrow next$ 将第 $i-1$ 个结点的指针赋值为第 i 个结点的指针,从而将第 i 个结点从链表中断开。

(3) 释放第 i 个结点所占空间以便于重用。

图 3-7 显示了删除结点前后链表中指针的变化。

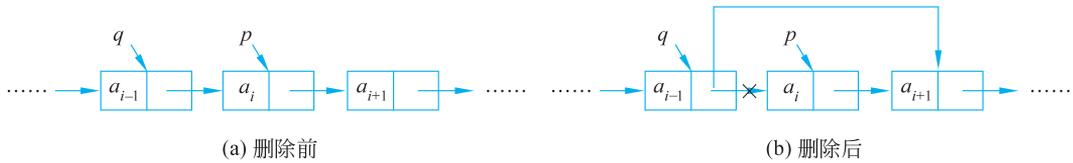


图 3-7 在单链表中删除结点 a_i

◇3.2.4 实验步骤

- (1) 定义联系人结构体。
- (2) 实现单链表的基本操作函数,包括插入函数 Insert 和删除函数 Delete。
- (3) 主函数对相应函数进行测试,实现一个简单的通讯录功能,包括有录入、删除、显示功能。

◇3.2.5 参考代码

参考代码如下。

```

1. #include<stdio.h>
2. #include<string.h>
3.
4. //定义联系人信息结构体
5. struct LNode{
6.     char name[15];           //姓名
7.     char tel[10];           //电话
8.     struct LNode * next;    //指针域
9. };
10.
11. //插入结点
12. void Insert(LNode * head, char * Name, char * Tel)
13. {
14.     LNode * p=head;        //p 指向头结点
15.     LNode * s=new LNode; //建立新结点 s
16.     strcpy(s->name, Name);
17.     strcpy(s->tel, Tel);
18.     s->next=p->next;      //修改结点 s 的指针
19.     p->next=s;           //修改结点 p(头结点)的指针
20. }
21. //删除结点
22. void Delete(LNode * head, char * Name)
23. {
24.     LNode * p, * q;
25.     q=head;
26.     p=head->next;        //p 指向头结点之后的首个结点
27.     while( p!=NULL && strcmp(p->name,Name) !=0 )
28.     {

```

```
29.     q=p;                //q 最终将指向被删除结点的前驱结点
30.     p=p->next;          //p 最终将指向被删除结点
31. }
32. if(p==NULL)
33.     printf("没有这条记录,无法删除!");
34. else
35.     {
36.         q->next=p->next;    //从链表中删除该结点
37.         delete p;          //释放结点 p
38.     }
39. }
40. //主函数
41. int main()
42. {
43.     LNode * head, * p;      //定义头指针、临时指针
44.     head = new LNode;      //定义头结点
45.     head->next = NULL;     //头结点指针域为空
46.     int select=1;          //操作选择: 0-结束,1-录入,2-删除,3-显示
47.     char Name[15];         //姓名
48.     char Tel[10];         //电话
49.     while(select!=0)
50.     {
51.         printf("请输入操作选择: 1-录入 2-删除 3-显示 0-结束\n");
52.         scanf("%d",&select);
53.         switch(select)
54.         {
55.             case 1:
56.                 printf("输入联系人的姓名、电话: ");
57.                 scanf("%s %s",&Name,&Tel);
58.                 Insert(head, Name, Tel);
59.                 break;
60.             case 2:
61.                 printf("请输入待删除联系人的姓名: ");
62.                 scanf("%s",&Name);
63.                 Delete(head, Name);
64.                 break;
65.             case 3:
66.                 p=head->next;    //p 指向头结点之后的首个结点
67.                 printf("联系人\t电话\n");
68.                 printf("=====\n");
69.                 while(p!=NULL)
70.                 {
71.                     printf("%s\t%s\n",p->name,p->tel);
72.                     p=p->next;
73.                 }
74.                 break;
```

```
75.         case 0:  
76.             printf("使用结束,再见!");  
77.             break;  
78.         }  
79.     }  
80.     return 0;  
81. }
```

◇3.2.6 实验结果

(1) 程序运行结果如图 3-8 所示。

```
请输入操作选择: 1-录入 2-删除 3-显示 0-结束  
1  
输入联系人的姓名、电话: 张红 13295043099  
请输入操作选择: 1-录入 2-删除 3-显示 0-结束  
1  
输入联系人的姓名、电话: 王力 13089886575  
请输入操作选择: 1-录入 2-删除 3-显示 0-结束  
3  
联系人 电话  
=====  
王力    13089886575  
张红    13295043099  
请输入操作选择: 1-录入 2-删除 3-显示 0-结束  
2  
请输入待删除联系人的姓名: 张红  
请输入操作选择: 1-录入 2-删除 3-显示 0-结束  
3  
联系人 电话  
=====  
王力    13089886575  
请输入操作选择: 1-录入 2-删除 3-显示 0-结束  
0  
使用结束,再见!
```

图 3-8 程序运行结果

(2) 算法复杂度分析。

链表为顺序存取,按位置访问元素的时间复杂度为 $O(n)$;而在插入和删除操作中,链表不需要移动元素,只需要修改指针,故时间复杂度为 $O(1)$ 。

◇3.2.7 实验总结

实验中使用单链表实现通讯录管理程序,存取操作时间复杂度为 $O(n)$,插入和删除操作时间复杂度为 $O(1)$ 。因此对于长度变化较大、需要频繁进行插入或删除操作的场景适合使用链表结构实现。

3.3 栈的存储与应用

◇3.3.1 实验目的及要求

- (1) 熟悉栈的定义和特点。
- (2) 掌握栈的顺序存储结构设计。

(3) 掌握出栈、入栈等基本操作实现。

◇3.3.2 实验内容

堆栈也有顺序存储方式和链式存储方式,这里只讨论顺序栈。

基于顺序栈,编写程序,实现数值的进制转换功能。

(1) 输入待转换的十进制数值。

(2) 输入要转换的进制。

(3) 输出入栈情况及转换结果。

◇3.3.3 实验原理

1. 栈的定义

栈是限制在表的一端进行插入和删除操作的线性表。允许进行插入和删除操作的一端称为栈顶,另一端称为栈底。入栈示意图如图 3-9 所示。

顺序栈利用一组连续存储的存储单元存放栈中的数据元素,可以用一维数组结构实现。例如,下面的结构创建了顺序栈。

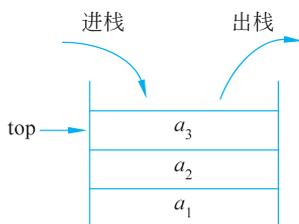


图 3-9 入栈示意图

```
typedef struct Sqstack{
    ElemType * data;           //存储元素的变量
    int top;                   //栈顶指针,存储栈顶元素的下标
    int stacksize;            //堆栈最大可分配空间数量,以元素为单位
}SqStack;
```

变量 data 并没有像顺序表中数组的定义那样,给定一个固定长度,而仅给了一个指针。这样就可以根据空间的需要,在使用顺序栈之前进行数组的初始化。

一般将数组的 0 下标作为栈底,将栈顶元素的下标存储在栈顶指针 top 中,它随着元素进栈出栈而变化。top 为 -1 表示空栈,top 等于 stacksize-1 则表示栈满。如果要将栈置为空栈,只要将 top 设为 -1 即可。

2. 栈的基本操作

堆栈的主要操作有:创建空栈、进栈、出栈、读栈顶元素等。

1) 进栈

即在栈顶插入元素。算法实现的主要步骤如下。

(1) 如果栈不满,则栈顶指针 top 加 1。

(2) 在栈顶处插入元素作为新的栈顶。

(3) 如果栈满,则返回进栈失败。

2) 出栈

即在栈顶删除元素。算法实现的主要步骤如下。

- (1) 如果栈不空,则返回栈顶元素。
- (2) 栈顶指针 top 减 1。
- (3) 如果栈为空,则返回出栈失败。

3. 进制转换的算法

日常生活中常用十进制,计算机中常用二进制、八进制、十六进制等,因此涉及进制的转换。对于十进制数值转换为非十进制 N 的一种方法为:十进制数除以 N ,取余数,然后再对商除以 N ,取余数……如此循环往复,直到商为零,将以上步骤得到的余数逆序输出,就得到了该数值在 N 进制下的表示。例如:十进制 9 转换为二进制的步骤如下。

$$\begin{array}{r}
 9 \div 2 = 4 \cdots 1 \\
 4 \div 2 = 2 \cdots 0 \\
 2 \div 2 = 1 \cdots 0 \\
 1 \div 2 = 0 \cdots 1
 \end{array}
 \begin{array}{l}
 \uparrow \\
 \text{逆序} \\
 \text{输出}
 \end{array}$$

因为余数要求逆序输出,即先得到的余数后输出,符合栈先进后出的特性,因此可以用栈结构实现进制转换算法。

◇3.3.4 实验步骤

- (1) 定义顺序栈 SqStack 并进行初始化。
- (2) 实现栈的基本操作函数,包括进栈(push)、出栈(pop)、遍历栈元素(StackTravel)。
- (3) 实现进制转换函数 conversion(),余数依次进栈。
- (4) 主函数中用户输入进制转换参数,并调用相关函数。

◇3.3.5 参考代码

参考代码如下。

```

1. #include<stdio.h>
2. #include<malloc.h>
3. #include<stdlib.h>
4. #define MAX_STACK_SIZE 10           //静态栈向量大小
5. #define ERROR 0
6. #define OK 1
7. typedef int DataType;
8. typedef int Status;
9. //定义栈

```

```
10. typedef struct sqstack{
11.     DataType stack_array[MAX_STACK_SIZE];
12.     int top;
13.     int bottom;
14. }SqStack;
15.
16. //初始化栈
17. void InitStack(SqStack * S) {
18.     S->bottom=S->top=0;
19.     printf("\n 初始化栈成功! \n");
20. }
21. //push(元素进栈)
22. Status push(SqStack * S , DataType data) {
23.     if(S->top >= MAX_STACK_SIZE - 1) {
24.         printf("栈满! \n");
25.         return ERROR; //栈满
26.     }
27.     printf("-----");
28.     printf("当前入栈元素: %d\n", data);
29.     S->top++; //位置自加
30.     printf("入栈后 S->top==%d\n", S->top);
31.     S->stack_array[S->top] = data; //元素入栈
32.     return OK;
33. }
34.
35. //pop 弹栈(元素出栈)
36. Status pop(SqStack * S , DataType * data) {
37.     if(S->top == 0) {
38.         return ERROR; //栈空
39.     }
40.     * data = S->stack_array[S->top]; //先取
41.     S->top--; //自减
42.     return OK;
43. }
44. //遍历栈(自顶向底)
45. Status StackTravel(SqStack * S) {
46.     int e;
47.     int ptr;
48.     ptr = S->top;
49.     while(ptr > S->bottom) {
50.         e = S->stack_array[ptr];
51.         ptr--;
52.         printf("%d", e);
53.     }
```

```

54.     return OK;
55. }
56. //进制转换(输入十进制数 n, 转换的进制 d)
57. void conversion(int n, int d) {
58.     SqStack S;                //创建栈
59.     DataType k;                //欲进栈的元素
60.     int temp = n;              //保存 n
61.     InitStack(&S);             //初始化栈
62.     while(n>0) {
63.         k = n %d;              //取余
64.         push(&S, k);           //余数进栈
65.         n = n / d;             //结果取整
66.     }
67.     printf("将%d转化成%d进制后为:", temp, d);
68.     StackTravel(&S);          //遍历栈
69. }
70. int main() {
71.     int n, d;
72.     printf("请输入要转换的十进制数: ");
73.     scanf("%d", &n);
74.     printf("请输入转换的进制: ");
75.     scanf("%d", &d);
76.     conversion(n, d);
77. }

```

◇3.3.6 实验结果

(1) 程序运行结果如图 3-10 所示。

```

请输入要转换的十进制数: 234
请输入转换的进制: 2

初始化栈成功!
-----当前入栈元素: 0
入栈后S->top==1
-----当前入栈元素: 1
入栈后S->top==2
-----当前入栈元素: 0
入栈后S->top==3
-----当前入栈元素: 1
入栈后S->top==4
-----当前入栈元素: 0
入栈后S->top==5
-----当前入栈元素: 1
入栈后S->top==6
-----当前入栈元素: 1
入栈后S->top==7
-----当前入栈元素: 1
入栈后S->top==8
将234转化成2进制后为:11101010
-----

```

图 3-10 程序运行结果

示例中为十进制转二进制,待转换的数为 234,按照进制转换方法,对 234 除以 2,商为 117,余数为 0,则 0 进栈;再对 117 除以 2,以此类推,最终,从栈顶到栈底的元素依

次为 11101010, 即转换得到的二进制数。

(2) 算法复杂度分析。

栈是一类特殊的线性表, 因此其复杂度与具体的实现方法有关, 分别与顺序表和链表的算法复杂度对应。本实验中为顺序栈, 即栈的顺序存储结构, 所以复杂度与顺序表相同。

◇3.3.7 实验总结

栈结构的特点是先进后出, 在进制转换方法中, 余数要求逆序输出, 即先得到的余数后输出, 符合栈结构的特性, 因此实验中使用了栈结构实现进制转换算法, 同时复习了进栈、出栈等基本操作。

3.4 队列的存储与应用

◇3.4.1 实验目的及要求

- (1) 熟悉队列的定义和特点。
- (2) 掌握队列的顺序存储结构设计。
- (3) 掌握入队、出队等基本操作实现。
- (4) 综合运用多种线性结构解决实际问题。

◇3.4.2 实验内容

利用栈和队列, 编写程序, 实现一个停车场调度程序。停车场的结构如下。

有一狭长停车场可停放 n 辆车, 只有一个门可供进出。车辆按照到达的早晚从最里面依次向外排列, 若停车场中的车辆 a 要出来, 则在它之后进入停车场的车都要让路, 进入暂停区域, 等要出场的车辆离开后, 这些车辆再依次进场。

例如, 停车场由内到外停放了 1, 2, 3, 4 共 4 辆车, 如图 3-11 所示, 若车辆 2 要出场, 则车辆 4、车辆 3 需要依次进入暂停区域, 车辆 2 离开后, 车辆 4 和车辆 3 通过暂停区域(环形车道)再次进入停车场, 此时停车场由内到外停放情况为: 车辆 1, 车辆 4, 车辆 3。

基于上述背景, 编写车辆调度程序, 实现以下功能。

- (1) 由用户指定停车场的大小 n 及第 a 辆车出停车场。
- (2) 输出以下两个时刻车辆的调度情况。

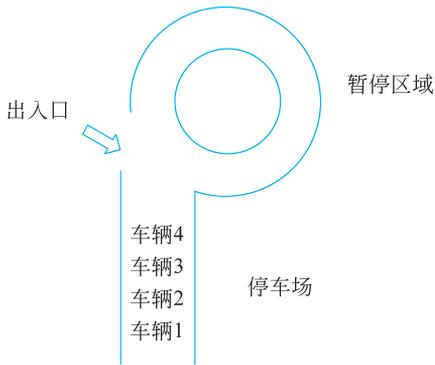


图 3-11 停车场示意图

- ① 依次输出需要驶入暂停区域的车辆。
- ② 第 a 辆车开出后, 暂停区域的车辆重新回到停车场, 此时停车场中的车辆。

◇3.4.3 实验原理

1. 队列的定义

队列是只能在表的一端进行插入、在另一端进行删除操作的线性表。允许删除元素的一端称为队头, 允许插入元素的一端称为队尾。

顺序存储的队列也可用一维数组来实现, $front$ 和 $rear$ 指针分别是队头和队尾元素的下标值, 如图 3-12 所示。顺序结构中, 采用数学方法将 $rear$ 或 $front$ 指针从数组空间的极大下标位置移到最小下标位置, 形成循环队列, 如图 3-13 所示。 $front$ 所在位置留空, 方便编程。



图 3-12 一般队列操作示意图

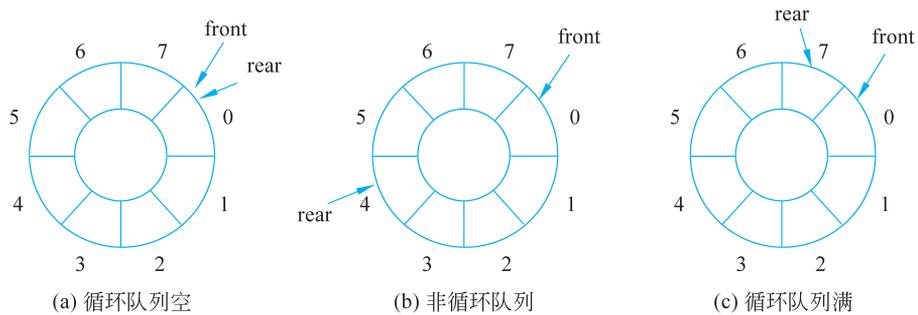


图 3-13 循环队列示意图

循环队列可采用结构体类型描述如下。

```
const int MAX=100;           //队列最大容量
struct SqQueue{
    int data[MAX];           //存放元素的数组(这里是 int 型元素)
    int front;               //队头指针
    int rear;                //队尾指针
};
```

2. 队列的基本操作

队列的主要操作有: 创建队列、入队、出队等。

1) 入队

即在队尾插入元素。算法实现的主要步骤如下。

(1) 如果队列不满,则令 $\text{rear}=(\text{rear}+1)\%M$ 。

(2) 将新元素在 rear 位置插入。

(3) 如果队列已满,则返回入队失败。

2) 出队

即在队头删除元素。算法实现的主要步骤如下。

(1) 如果队列不空,则令 $\text{front}=(\text{front}+1)\%M$ 。

(2) 将下标为 front 的元素取出。

(3) 如果队列为空,则返回出队失败。

3. 问题分析

对于停车场,符合先进后出的特点,因此可以用栈结构模拟。车辆进入停车场即入栈,车辆离开停车场即出栈。

对于暂停区域,车辆进入暂停区域后绕一周再重新进入停车场,符合先进先出的特点,因此可以用队列结构模拟。车辆进入暂停区域即入队,车辆离开暂停区域即出队。

◇3.4.4 实验步骤

(1) 定义顺序栈 SqStack 和循环队列 SqQueue。

(2) 实现栈和队列的基本操作函数,包括进栈(Push)、出栈(Pop)、入队(EnQueue)、出队(DeQueue)。

(3) 主函数中用户输入车辆调度所需参数,调用相关函数实现调度,主要步骤如下。

① 车辆进入停车场(入栈操作)。

② 需要移动的车辆进入暂停区域(出栈和入队操作)。

③ 暂停区域的车重新进入停车场(出队和入栈操作)。

◇3.4.5 参考代码

参考代码如下。

```
1. #include<stdio.h>
2. #include<malloc.h>
3. #include<stdlib.h>
4. #define MAXQSIZE 100
5. #define STACKINCREMENT 10
6. #define MAX_STACK_SIZE 10 //静态栈向量大小
7. #define ERROR 0
8. #define OK 1
9. typedef int DataType;
10. typedef int Status;
11.
```

```
12. //定义栈
13. typedef struct sqstack{
14.     DataType stack_array[MAX_STACK_SIZE];
15.     int top;
16.     int bottom;
17. }SqStack;
18.
19. //初始化栈
20. void InitStack(SqStack * S) {
21.     S->bottom=S->top=0;
22. }
23. //Push(元素进栈)
24. Status Push(SqStack * S , DataType data) {
25.     if(S->top >= MAX_STACK_SIZE - 1) {
26.         printf("栈满!\n");
27.         return ERROR;           //栈满
28.     }
29.     S->top++;                   //位置自加
30.     S->stack_array[S->top] = data; //元素入栈
31.     return OK;
32. }
33.
34. //Pop 弹栈(元素出栈)
35. Status Pop(SqStack * S, DataType * data) {
36.     if(S->top == 0) {
37.         return ERROR;           //栈空
38.     }
39.     * data = S->stack_array[S->top]; //先取
40.     S->top--;                   //自减
41.     return OK;
42. }
43. //遍历栈(自底向顶)
44. Status StackTravel(SqStack * S) {
45.     int e;
46.     int ptr;
47.     ptr = S->bottom+1;
48.     while(ptr <= S->top) {
49.         e = S->stack_array[ptr];
50.         ptr++;
51.         printf("%d  ",e);
52.     }
53.     printf("\n");
54.     return OK;
55. }
56.
```

```
57. //顺序表示的循环队列
58. typedef struct
59. {
60.     int * base;
61.     int front;
62.     int rear;
63. }SqQueue;
64.
65. //队的初始化
66. int InitQueue(SqQueue * Q)
67. {
68.     Q->base=(int *) malloc(MAXQSIZE * sizeof(int));
69.     if(!Q->base)
70.         exit(0);
71.     Q->front=Q->rear=0;
72.     return 1;
73. }
74. //入队
75. int EnQueue(SqQueue * Q,int e)
76. {
77.     if((Q->rear+1) %MAXQSIZE==Q->front)
78.         exit(0);
79.     Q->base[Q->rear]=e;
80.     Q->rear=(Q->rear+1) %MAXQSIZE;
81.     return 1;
82. }
83. //出队
84. int DeQueue(SqQueue * Q)
85. {
86.     int e;
87.     if(Q->front==Q->rear)
88.         exit(0);
89.     e=Q->base[Q->front];
90.     Q->front=(Q->front+1) %MAXQSIZE;
91.     return e;
92. }
93. //主函数部分
94. int main()
95. {
96.     SqQueue tempArea;
97.     SqStack parking;
98.     InitStack(&parking);
99.     InitQueue(&tempArea);
100. int n;
```

```
101.     printf("请输入总车辆数 n:");
102.     scanf("%d", &n);
103.     //车辆进入停车场
104.     for(int i=1;i<=n;i++)
105.         Push(&parking, i);
106.     StackTravel(&parking);
107.     //用户指定要驶离的车辆
108.     int targetCar;
109.     printf("请输入想让第几辆车出停车场:");
110.     scanf("%d", &targetCar);
111.     printf("*****调度情况*****\n");
112.     int car;
113.     Pop(&parking, &car);
114.     if(car==targetCar)
115.         printf("该辆车可直接开出\n");
116.     else{
117.         Push(&parking, car);
118.         printf("以下车辆依次从停车场驶入暂停区域\n");
119.         int k=0;
120.         while(car!=targetCar)
121.             {
122.                 Pop(&parking, &car);
123.                 if(car==targetCar)
124.                     break;
125.                 EnQueue(&tempArea, car);
126.                 printf("%d    ", car);
127.                 k++;
128.             }
129.         //暂停区域的车辆再驶入停车场
130.         for(int j=0;j<k;j++)
131.             {
132.                 car=DeQueue(&tempArea);
133.                 Push(&parking, car);
134.             }
135.         }
136.         printf("\n 停车场里的情况 (由里到外) 为:\n");
137.         StackTravel(&parking);
138.         printf("\n");
139.         printf("\n*****调度完毕*****\n");
140.     }
```

◇3.4.6 实验结果

程序运行结果如图 3-14 所示。

```
请输入总车辆数n:5
请输入您想让第几辆车出停车场a:3
*****调度情况*****
车辆出停车场的情况:
以下车辆依次从停车场驶入暂停区域
5   4
停车场里的情况(由里到外)为:
1   2   5   4
*****调度完毕*****
```

图 3-14 程序运行结果

示例中,停车场由内到外停放了 1,2,3,4,5 共 5 辆车,当用户指定车辆 3 要出场时,程序执行以下步骤。

- (1) 车辆 5,车辆 4 依次进入暂停区域,对应的操作为 5 出栈后入队,4 出栈后入队。
- (2) 车辆 3 离开停车场,对应的操作为 3 出栈。
- (3) 车辆 5 和车辆 4 从暂停区域再次进入停车场,对应的操作为 5 出队后入栈,4 出队后入栈。
- (4) 最终停车场由内到外停放情况,即栈底到栈顶的元素依次为:车辆 1,车辆 2,车辆 5,车辆 4。

◇3.4.7 实验总结

本实验具有一定的综合性,栈结构的特点是先进后出,队列的特点是先进先出。在停车场问题中,用栈模拟停车场,用队列模拟暂停区域,车辆的进出则用出栈入栈、出队入队操作实现。通过本实验可进一步理解栈和队列这两种线性表的结构特点和基本操作方法。