南京信息工程大学 数据结构 I 实验(实习)报告

实验(实习)名称 栈和队列 日期 2024.10. 得分 指导教师

学院 计院 专业 计科

一、实验目的

1、掌握栈的定义及特点；
2、掌握栈的基本操作，并对其进行简单应用；
3、理解什么是循环队列；
4、掌握循环队列的基本操作，并对其进行简单的应用。

二、 实验内容与步骤

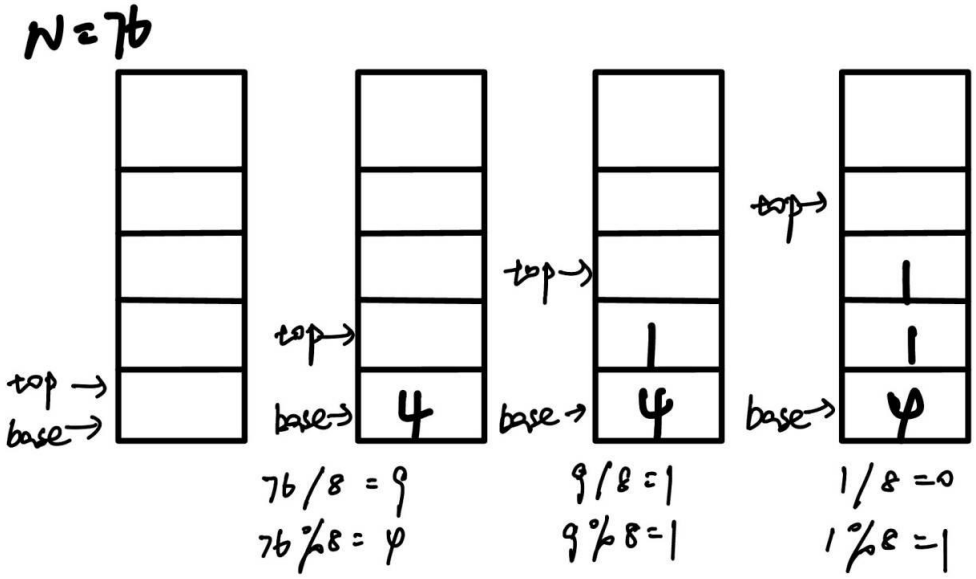
1、编写程序，把十进制正整数转换为 n（n 可以为 2、8、16 等）进制数并输出。 注意：转换用教材算法 3.1 来实现，其它方法不给分；其中顺序栈的定义和基本操作算法见教材 P46-P47 页，主要包括：定义栈结构、初始化一个空栈、获取栈顶元素、栈顶元素出栈、将数据元素压入栈（进栈）等。

要求：

（1）写出解决问题的算法思想；

首先创建一个空栈，用于存储转换过程中得到的余数；接着用循环法求余，将待转换的十进制数依次除以目标进制 n（如 2、8、16 等），每次将余数压入栈，将商作为下一个循环数的被除数，直到商为 0；最后出栈形成结果，依次将栈中元素出栈，即可按从高位到低位的顺序得到目标进制数的表示形式。（对于十六进制转换，需要将余数值大于等于 10 的情况映射到字符 A-F）

（2）仿照教材图 3.2，画出栈操作过程的变化图；



（3）完整的程序代码（文本形式）；

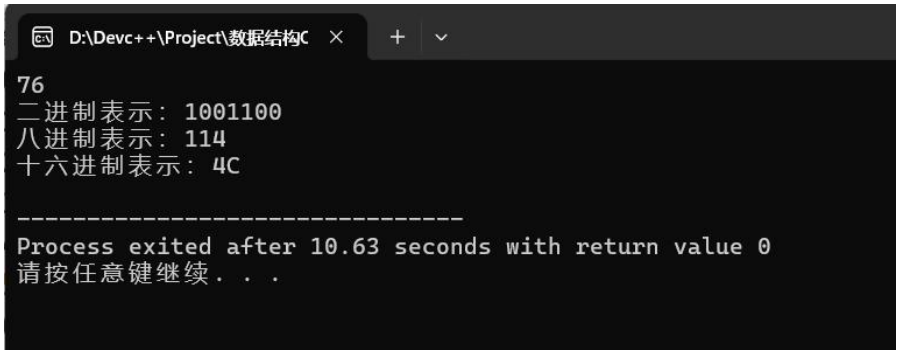
```
#include<iostream>
#include<stdlib.h>
using namespace std;
```

```
#define OK 1
#define OVERFLOW -2
#define ERROR 0
#define STACK_INIT_SIZE 100//初始分配量
#define STACKINCREMENT 10//空间增量

typedef int Status;
typedef int SElemType;
//顺序栈类型
typedef struct{
   SElemType *base;//栈底位置
   SElemType *top;//栈顶位置
   int stacksize;//栈当前分配空间
}SqStack;
//构造空栈
Status InitStack(SqStack &S)
{
   S.base=(SElemType*)malloc(STACK_INIT_SIZE*sizeof(SElemType));
   if(!S.base) exit(OVERFLOW);
   S.top=S.base;
   S.stacksize=STACK_INIT_SIZE;
   return OK;
}
//进栈
Status Push(SqStack &S,SElemType e)
{
   if(S.top-S.base>=S.stacksize)
   {

   S.base=(SElemType*)realloc(S.base,(S.stacksize+STACKINCREMENT)*sizeof(SElemType))
;
        if(!S.base) exit(OVERFLOW);
        S.top=S.base+S.stacksize;
        S.stacksize+=STACKINCREMENT;
   }
   *S.top++=e;
   return OK;
}
//出栈
Status Pop(SqStack &S,SElemType &e)
{
   if(S.top==S.base) return ERROR;
   e=*--S.top;
```

```
    return OK;
}
// 判断栈是否为空
Status StackEmpty(SqStack S) {
    if (S.top == S.base) {
        return true;
    }
    return false;
}
//转换为 2 进制
void conversion_2(int x)
{
    SqStack S;
    InitStack(S);
    while(x!=0)
    {
        Push(S,x%2);
        x/=2;
    }
    printf("二进制表示：");
    while(!StackEmpty(S))
    {
        int e;
        Pop(S,e);
        printf("%d",e);
    }
    printf("\n");
}
//转换为 8 进制
void conversion_8(int x)
{
    SqStack S;
    InitStack(S);
    while(x!=0)
    {
        Push(S,x%8);
        x/=8;
    }
    printf("八进制表示：");
    while(!StackEmpty(S))
    {
        int e;
        Pop(S,e);
        printf("%d",e);
```

```
    }
    printf("\n");
}
//转换为 16 进制
void conversion_16(int x)
{
    SqStack S;
    InitStack(S);
    while(x!=0)
    {
        Push(S,x%16);
        x/=16;
    }
    printf("十六进制表示：");
    while(!StackEmpty(S))
    {
        int e;
          Pop(S, e);
          if(e<10){
              printf("%d",e);   // 0-9  直接输出
          }
        else{
              printf("%c",'A'+(e - 10));   // 10-15  转换为  A-F
          }
    }
    printf("\n");
}
int main() {
    int n;
    cin>>n;
    conversion_2(n);
    conversion_8(n);
    conversion_16(n);
    return 0;
}
```

（4）运行结果（截图）；



2、假设表达式中允许包含两种括号：圆括号和方括号，其嵌套的顺序随意，参考教材 3.2.2 小节或课件上的内容，编写程序判别表达式中的括号是否匹配，以“#”作为算术表达式的结束符。测试数据例如：5*[(8-2)/(7+6)) #、5*[(8-2)/(7+6)]#、5*[(8-2)/(7+6)#。

要求：

（1）写出解决问题的算法思想；

使用栈匹配括号：利用栈的先进后出（LIFO）特性来匹配括号。每遇到一个左括号 (或 [，就将其压入栈中；当遇到右括号) 或] 时，检查栈顶元素是否是对应的左括号。如果匹配，则将栈顶元素出栈；如果不匹配，则括号不平衡，直接返回“不匹配”。

逐字符扫描：从表达式的第一个字符开始，逐个扫描直到结束符 #。在扫描过程中，按照以下规则处理字符：遇到左括号 (或 [时，将其压入栈中。遇到右括号) 或] 时，检查栈是否为空：如果栈为空，说明没有对应的左括号，返回“不匹配”。如果栈非空，检查栈顶元素是否是对应的左括号：若匹配，则出栈继续扫描。若不匹配，则直接返回“不匹配”。

最终栈状态判断：扫描完表达式后，检查栈是否为空。如果栈为空，表示所有的左括号都有对应的右括号，括号匹配成功。如果栈非空，说明存在未匹配的左括号，返回“不匹配”。

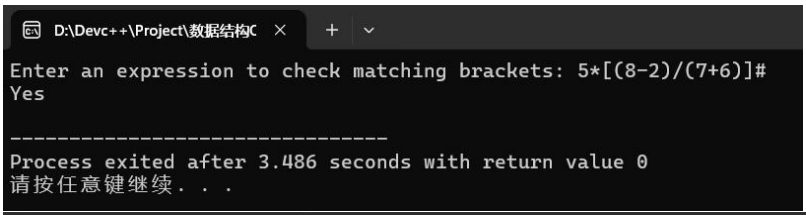
（2）完整的程序代码（文本形式）；

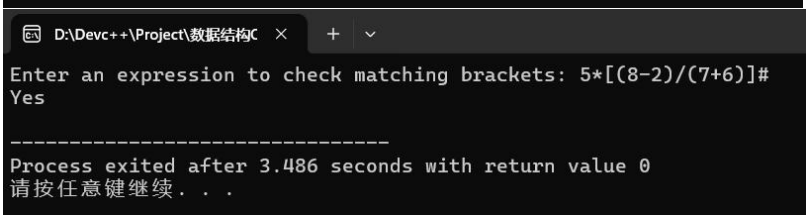
```
#include <iostream>
#include <stdlib.h>
using namespace std;

#define OK 1
#define OVERFLOW -2
#define ERROR 0
#define STACK_INIT_SIZE 100 // 初始分配量
#define STACKINCREMENT 10   // 空间增量

typedef int Status;
typedef char SElemType;
// 顺序栈类型
typedef struct {
    SElemType *base; // 栈底位置
    SElemType *top;  // 栈顶位置
    int stacksize;   // 栈当前分配空间
} SqStack;

// 构造空栈
Status InitStack(SqStack &S) {
    S.base = (SElemType *)malloc(STACK_INIT_SIZE * sizeof(SElemType));
    if (!S.base) exit(OVERFLOW);
    S.top = S.base;
    S.stacksize = STACK_INIT_SIZE;
    return OK;
}
// 进栈
Status Push(SqStack &S, SElemType e) {
```

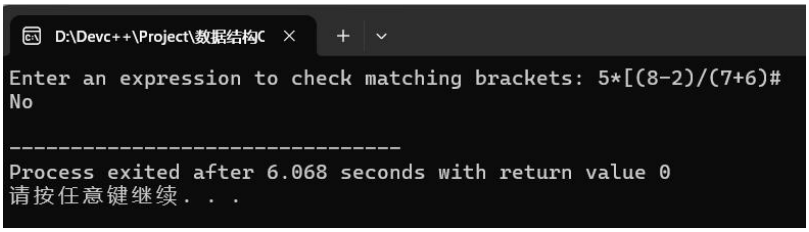
```
        if (S.top - S.base >= S.stacksize) {
            S.base = (SElemType *)realloc(S.base, (S.stacksize + STACKINCREMENT) *
sizeof(SElemType));
            if (!S.base) exit(OVERFLOW);
            S.top = S.base + S.stacksize;
            S.stacksize += STACKINCREMENT;
        }
        *S.top++ = e;
        return OK;
    }
    // 出栈
    Status Pop(SqStack &S, SElemType &e) {
        if (S.top == S.base) return ERROR;
        e = *--S.top;
        return OK;
    }
    // 取栈顶元素
    Status GetTop(SqStack S, SElemType &e) {
        if (S.top == S.base) return ERROR;
        e = *(S.top - 1);
        return OK;
    }
    // 判断栈是否为空
    bool StackEmpty(SqStack S) {
        return S.top == S.base;
    }
    // 判断字符串中的括号匹配
    Status matching(char *exp) {
        SqStack S;
        InitStack(S);
        char e;
        char *p = exp;
        while (*p != '#') {
            switch (*p) {
                case '[':
                case '(':
                    Push(S, *p);
                    break;
                case ')':
                    if (StackEmpty(S)) {
                        free(S.base);
                        return 0;
                    }
                    GetTop(S, e);
```

```
                if (e == '(') {
                    Pop(S, e);
                } else {
                    free(S.base);
                    return 0;
                }
                break;
            case ']':
                if (StackEmpty(S)) {
                    free(S.base);
                    return 0;
                }
                GetTop(S, e);
                if (e == '[') {
                    Pop(S, e);
                } else {
                    free(S.base);
                    return 0;
                }
                break;
            default:
                break;
        }
        p++;
    }
    bool result = StackEmpty(S);
    free(S.base);
    return result ? 1 : 0;
}

int main() {
    char exp[100];
    cout << "Enter an expression to check matching brackets: ";
    cin >> exp;
    if (matching(exp)) {
        cout << "Yes" << endl;
    } else {
        cout << "No" << endl;
    }
    return 0;
}
```

（3）运行结果（截图）。







3、参考课件上的内容编写程序，实现对给定的后缀表示式（逆波兰式）如 2 9 6 3 / +5 - * 4 + 进行求值。假定算术表达式中只包含 +、-、*、/ 四种运算。

要求：

（1）写出解决问题的算法思想；

遇到操作数时，将其压入栈中；遇到运算符时，从栈中弹出两个操作数进行计算，并将结果重新压入栈中。

（2）完整的程序代码（文本形式）；

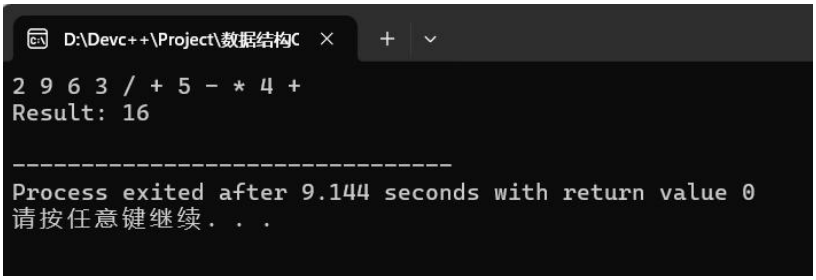
```
#include<iostream>
#include<stdlib.h>
using namespace std;

#define OK 1
#define OVERFLOW -2
#define ERROR 0
#define STACK_INIT_SIZE 100//初始分配量
#define STACKINCREMENT 10//空间增量

typedef char Status;
typedef double SElemType;
//顺序栈类型
typedef struct{
   SElemType *base;//栈底位置
   SElemType *top;//栈顶位置
   int stacksize;//栈当前分配空间
}SqStack;

//构造空栈
Status InitStack(SqStack &S)
```

```
{
    S.base=(SElemType*)malloc(STACK_INIT_SIZE*sizeof(SElemType));
    if(!S.base) exit(OVERFLOW);
    S.top=S.base;
    S.stacksize=STACK_INIT_SIZE;
    return OK;
}
//进栈
Status Push(SqStack &S,SElemType e)
{
    if(S.top-S.base>=S.stacksize)
    {

    S.base=(SElemType*)realloc(S.base,(S.stacksize+STACKINCREMENT)*sizeof(SElemType))
;
        if(!S.base) exit(OVERFLOW);
        S.top=S.base+S.stacksize;
        S.stacksize+=STACKINCREMENT;
    }
    *S.top++=e;
    return OK;
}
//出栈
Status Pop(SqStack &S,SElemType &e)
{
    if(S.top==S.base) return ERROR;
    e=*--S.top;
    return OK;
}
//取栈顶元素
Status GetTop(SqStack S,SElemType &e)
{
    if(S.top==S.base) return ERROR;
    e=*(S.top-1);
    return OK;
}
// 判断栈是否为空
Status StackEmpty(SqStack S)
{
    if (S.top == S.base) {
        return true; // 栈为空
    }
    return false; // 栈不为空
}
```

```
// 打印栈中元素
void PrintStack(SqStack S)
{
    SElemType *p = S.base;
    while (p != S.top)
   {
        cout << *p << " ";
        p++;
    }
    cout << endl;
}

int main() {
    SqStack stack;
    InitStack(stack);
    string str;
    //    2 9 6 3 / + 5 - * 4 +
    getline(cin, str);
    double result = 0;
    for (int i = 0; i < str.size(); i++) {
        char s = str[i];
        // 忽略空格
        if (s == ' ') continue;
        // 如果是数字字符，转换为数值并压入栈
        if (isdigit(s)) {
            Push(stack, s - '0');   // 将字符转换为整数并压入栈
        }
        // 如果是操作符，弹出两个操作数并执行运算
        else if (s == '+' || s == '-' || s == '*' || s == '/') {
            double b, a;
            Pop(stack, b);
            Pop(stack, a);
            switch (s) {
                case '+': result = a + b; break;
                case '-': result = a - b; break;
                case '*': result = a * b; break;
                case '/':
                    if (b == 0) {
                        cout << "Error" << endl;
                        return ERROR;
                    }
                    result = a / b;
                    break;
            }
```

```
            Push(stack, result);   // 将运算结果压入栈
        } else {
            return ERROR;
        }
    }
    // 检查栈中最终结果
    if (Pop(stack, result) == OK && stack.top == stack.base) {
        cout << "Result: " << result << endl;
    } else {
        cout << "ERROR" << endl;
    }

    return 0;
}
```

（3）运行结果（截图）。



4、编写程序：假设循环队列的最大长度为 7，现在依次将以下数据入队列：{7，5，3，9，2，4}； 接着进行 3 次出队列的操作，再将 15、18 这两个数据入队列，最后从队头到队尾依次输出队列中的元素。

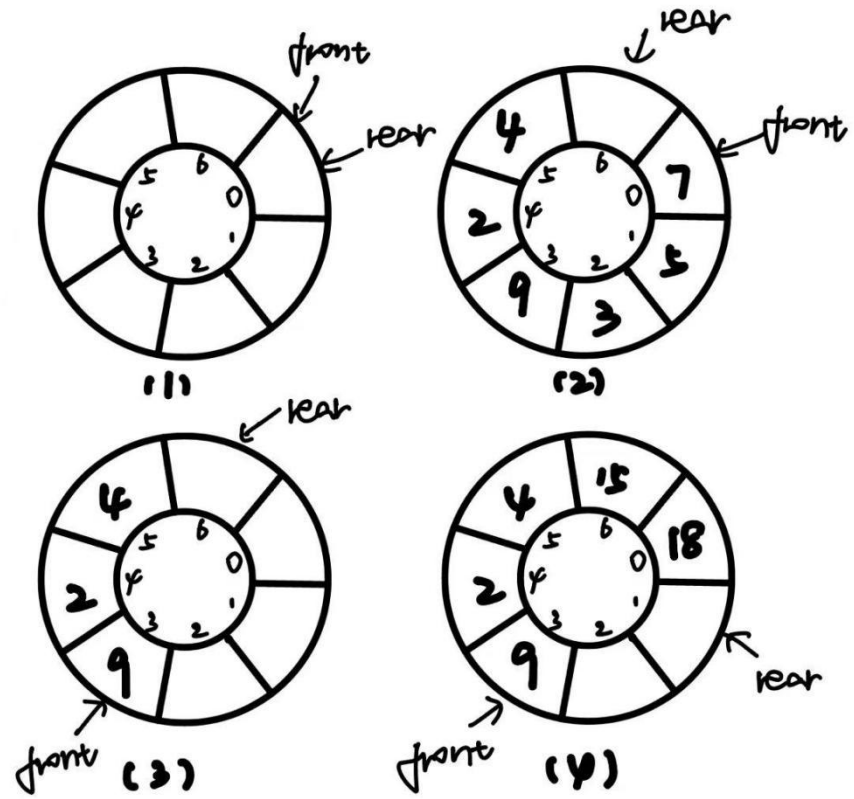
（1）回答为什么要构造循环队列；

构造循环队列的主要原因是为了在固定长度的数组中有效利用内存空间。当队列容量有限（例如本题中的循环队列最大长度为 7）时，普通的线性队列在多次入队和出队操作后，即使数组前面存在空闲空间，队尾也会到达数组末尾，导致无法再入队。循环队列通过将数组末尾与开头相连，可以实现空间的循环利用，避免内存浪费。

（2）复习教材 3.4.3 小节中有关循环队列的内容（主要包括：如何构造循环队列、如何删除对头元素、如何在队列中增加元素、如何判断循环队列的空和满的状态等），分析并写出解决问题的基本思路；

首先初始化队列，将 front 和 rear 指向同一个位置（初始为 0）。依次执行 6 次入队操作，将数据{7, 5, 3, 9, 2, 4}入队。执行 3 次出队操作，从队头删除 3 个元素。再将数据 15 和 18 入队。从队头到队尾依次输出队列中的所有元素。

（3）仿照教材 P64 页图 3.14，画出本题循环队列的操作过程变化图；



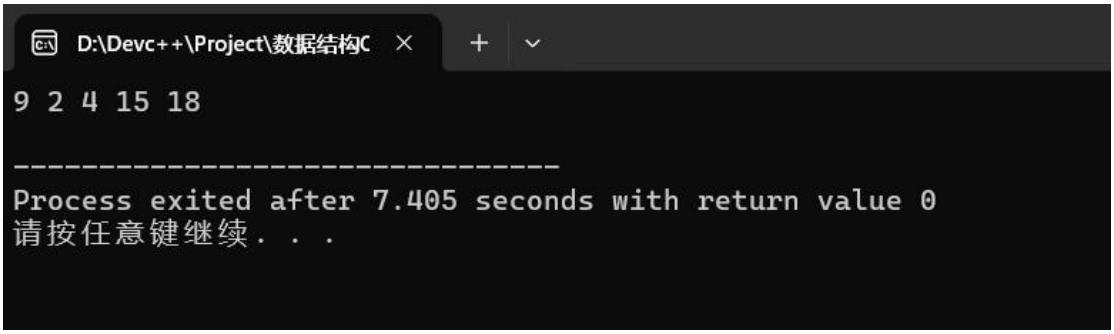
（4）完整的程序代码（文本形式）；

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

#define MAXSIZE 7 //  队列的最大容量
#define ERROR 0
#define OK 1

typedef int Status;
typedef int QElemType;

//  循环队列结构体
typedef struct {
    QElemType data[MAXSIZE];
    int front; //  队头指针
    int rear;   //  队尾指针
} CircularQueue;

//  初始化队列
Status InitQueue(CircularQueue &Q) {
    Q.front = 0;
    Q.rear = 0;
    return OK;
```

```
}

// 判断队列是否为空
bool QueueEmpty(CircularQueue Q) {
    return Q.front == Q.rear;
}

// 判断队列是否已满
bool QueueFull(CircularQueue Q) {
    return (Q.rear + 1) % MAXSIZE == Q.front;
}

// 入队
Status EnQueue(CircularQueue &Q, QElemType e) {
    if (QueueFull(Q)) {
        cout << "Queue is full!" << endl;
        return ERROR;
    }
    Q.data[Q.rear] = e;
    Q.rear = (Q.rear + 1) % MAXSIZE;
    return OK;
}

// 出队
Status DeQueue(CircularQueue &Q, QElemType &e) {
    if (QueueEmpty(Q)) {
        cout << "Queue is empty!" << endl;
        return ERROR;
    }
    e = Q.data[Q.front];
    Q.front = (Q.front + 1) % MAXSIZE;
    return OK;
}

// 获取队头元素
Status GetFront(CircularQueue Q, QElemType &e) {
    if (QueueEmpty(Q)) {
        cout << "Queue is empty!" << endl;
        return ERROR;
    }
    e = Q.data[Q.front];
    return OK;
}
```

```
// 打印队列中的元素
void PrintQueue(CircularQueue Q) {
    if (QueueEmpty(Q)) {
        cout << "Queue is empty!" << endl;
        return;
    }
    int i = Q.front;
    while (i != Q.rear) {
        cout << Q.data[i] << " ";
        i = (i + 1) % MAXSIZE;
    }
    cout << endl;
}

int main() {
    CircularQueue Q;
    InitQueue(Q);
    //入队操作
    int data[6]={7,5,3,9,2,4};
    for (int i=0;i<6;++i){
        EnQueue(Q,data[i]);
    }
    //出队操作
    QElemType e;
    for(int i=0;i<3;++i){
        DeQueue(Q,e);
    }
    //入队操作
    EnQueue(Q,15);
    EnQueue(Q,18);
    //打印队列元素
    PrintQueue(Q);
    return 0;
}
```

（5）运行结果（截图）。



三、 实验心得（必写）

在本次数据结构实验中，我进一步掌握了栈和队列的定义与操作。通过使用栈实现了十进制数的多进制转换、表达式括号匹配和后缀表达式求值等应用，加深了对栈的后进先出特性的理解。在循环队列的实现过程中，理解了其对内存空间的高效利用方式，避免了普通线性队列的空间浪费问题。实验过程中，解决了指针管理和空间动态分配等问题，提高了代码调试和问题排查的能力。总体而言，这次实验让我更深入地理解了栈和队列的应用，提升了编程水平。