

一、实验目的

本次实验要求在 Linux 系统中实现，并通过 git 命令行将实验文件传入到 GitHub 仓库中

二、实验原理

1、队列：

队列（queue）在计算机科学中，是一种先进先出的线性表。

它只允许在表的前端（front）进行删除操作，而在表的后端（rear）进行插入操作。进行插入操作的端称为队尾，进行删除操作的端称为队头。队列中没有元素时，称为空队列。

2、栈：

在计算机领域，堆栈是一个不容忽视的概念，堆栈是两种数据结构。堆栈都是一种数据项按序排列的数据结构，只能在一端(称为栈顶(top))对数据项进行插入和删除。在单片机应用中，堆栈是个特殊的存储区，主要功能是暂时存放数据和地址，通常用来保护断点和现场。要点:堆，队列优先,先进先出。栈，先进后出(First-In/Last-Out)。

3、B+树：

B+ 树是一种树数据结构，是一个 n 叉排序树，每个节点通常有多个孩子，一棵 B+树包含根节点、内部节点和叶子节点。根节点可能是一个叶子节点，也可能是一个包含两个或两个以上孩子节点的节点。

B+ 树通常用于数据库和操作系统的文件系统中。NTFS, ReiserFS, NSS, XFS, JFS, ReFS 和 BFS 等文件系统都在使用 B+树作为元数据索引。B+ 树的特点是能够保持数据稳定有序，其插入与修改拥有较稳定的对数时间复杂度。B+ 树元素自底向上插入。

三、实验过程

1、队列：

定义函数：

```
F:\linux\duilieaaaaa\1.cpp
#include<stdlib.h>
typedef int ElementType;           // 定义数据类型
// 定义节点结构
typedef struct Node {
    ElementType Element;           // 数据域
    struct Node * Next;
}NODE, *PNODE;

// 定义队列结构体
typedef struct QNode {
    PNODE Front, Rear;             // 队列头，尾指针
} Queue, *PQueue;

// 声明函数体
void InitQueue(PQueue);           // 创建队列函数
bool IsEmptyQueue(PQueue);        // 判断队列是否为空函数
void InsertQueue(PQueue, int val); // 入队函数
void DeleteQueue(PQueue, int * val); // 出队函数
void DestroyQueue(PQueue);         // 摧毁队列函数
void TraverseQueue(PQueue);        // 遍历队列函数
void ClearQueue(PQueue);           // 清空队列函数
int LengthQueue(PQueue);           // 求队列长度函数

// 主函数
```

定义插入函数：

```
F:\linux\duilieaaaaa\1.cpp

    return false;
}

// 定义入队函数
// 从队列尾部插入数据val
void InsertQueue(PQueue queue,int val) {
    PNODE P = (PNODE)malloc(sizeof(NODE)); // 创建一个新节点用于存放插入的元素
    if (P == NULL) {
        printf("内存分配失败，无法插入数据%d...", val);
        exit(-1);
    }
    P->Element = val; // 把要插入的数据放到节点数据域
    P->Next = NULL; // 新节点指针指向为空
    queue->Rear->Next = P; // 使上一个队列尾部的节点指针指向新建的节点
    queue->Rear = P; // 更新队尾指针，使其指向队列最后的节点
    printf("插入数据 %d 成功...\n", val);
}

// 定义出队函数
// 从队列的首节点开始出队
// 若出队成功，用val返回其值
void DeleteQueue(PQueue queue,int* val) {
    if (IsEmptyQueue(queue)) {
```

定义删除函数：

```
F:\linux\duilieaaaaa\1.cpp

    queue->Rear = P; // 更新队尾指针，使其指向队列最后的节点
    printf("插入数据 %d 成功...\n", val);
}

// 定义出队函数
// 从队列的首节点开始出队
// 若出队成功，用val返回其值
void DeleteQueue(PQueue queue,int* val) {
    if (IsEmptyQueue(queue)) {
        printf("队列已经空，无法出队...\n");
        exit(-1);
    }
    PNODE P= queue->Front->Next; // 临时指针
    *val = P->Element; // 保存其值
    queue->Front->Next = P->Next; // 更新头节点
    if (queue->Rear==P)
        queue->Rear = queue->Front;
    free(P); // 释放头队列
    P = NULL; // 防止产生野指针
    printf("出栈成功，出栈值为 %d\n", *val);
}

// 定义队列遍历函数
void TraverseQueue(PQueue queue) {
```

遍历函数：

```
F:\linux\duilieaaaaa\1.cpp
if (queue->Rear==P)
    queue->Rear = queue->Front;
free(P); // 释放头队列
P = NULL; // 防止产生野指针
printf("出栈成功, 出栈值为 %d\n", *val);
}
// 定义队列遍历函数
void TraverseQueue(PQueue queue) {
    if (IsEmptyQueue(queue)) {
        exit(-1);
    }
    PNODE P = queue->Front->Next; //从队列首节点开始遍历（非头节点，注意区分）
    printf("遍历队列结果为：");
    while (P != NULL) {
        printf("%d ", P->Element);
        P = P->Next;
    }
    printf("\n");
}
// 定义队列的摧毁函数
// 删除整个队列，包括头节点
void DestroyQueue(PQueue queue) {
    //从首节点开始删除
}
```

2、栈：

判断运算符的优先级：

```
int PRI(char oper1, char oper2) //判断两个运算符的优先级
//如果oper1>oper2返回1 如果oper1<oper2返回-1 如果oper1, oper2是左右符号返回0
{
    int pri;
    switch(oper2){ //判断优先级
        case '+':
        case '-':
            if(oper1=='(' || oper1=='=') //为左括号
                pri=-1; //oper1<oper2
            else
                pri=1; //oper1>oper2
            break;
        case '*':
        case '/':
            if(oper1=='*' || oper1=='/' || oper1=='=')
                pri=1; //oper1>oper2
            else
                pri=-1; //oper1<oper2
            break;
        case '(':
            if(oper1=='') //右括号右侧不能马上出现左括号
            {
                printf("语法错误\n");
                exit(0);
            } else
                pri=-1; //oper1<oper2
            break;
    }
```

```

case ')':
    if(oper1=='(')
        pri=0;
    else if(oper1=='=')
    {
        printf("括号不匹配\n");
        exit(0);
    }
    else
        pri=1;
    break;
case '=':
    if(oper1=='(')
    {
        printf("括号不匹配\n");
        exit(0);
    }
    else if(oper1=='=')
        pri=0; //等号匹配, 返回0
    else
        pri=1; //oper1>oper2
    break;
}
return pri;
}

```

int CalcExp(char exp[]) //表达式计算函数

```

{
    seqStack *StackOper,*StackData; //指向两个栈的指针变量, 一个操作符, 一个运算数
    int i=0,flag=0; //flag 作为标志, 用来处理多位数
    DATA a,b,c,q,x,t,oper;
    StackOper=seqStackInit();
    StackData=seqStackInit(); //初始化两个栈
    q=0; //变量 q 保存多位数的操作
    x='=';
    seqStackPush(StackOper,x); //首先把等号=进入操作栈
    x=seqStackPeek(StackOper); //获取操作栈的首元素
    c=exp[i++];
    while(c!=''|x!='=') //循环处理表达式中的每一个字符
    {
        if(isOperator(c)) //如果是运算符
        {
            if(flag){
                seqStackPush(StackData,q); //表达式入栈
                q=0; //操作数清零
                flag=0; //标志清零, 表示操作数已经入栈
            }
            switch(PRI(x,c)) //判断运算符优先级
            {

```

```

        case -1:
            seqStackPush(StackOper,c); //运算符进栈
            c=exp[i++];
            break;
        case 0:
            c=seqStackPop(StackOper); //运算符括号，等号出栈，被抛弃
            c=exp[i++]; //取下一个 字符
            break;
        case 1:
            oper=seqStackPop(StackOper); //运算符出栈
            b=seqStackPop(StackData);
            a=seqStackPop(StackData); //两个操作数出栈
            t=Calc(a,oper,b); //计算结果
            seqStackPush(StackData,t); //将运算结果入栈
            break;
    }
}
else if(c>='0'&&c<='9') //如果输出的字符在 0 到 9 之间
{
    c-='0'; //把字符转换成数字
    q=q*10+c; //多位数的进位处理
    c=exp[i++]; //取出下一位字符
    flag=1; //设置标志，表示操作数未入栈
}
else
{
    printf("输入错误\n");
    getch();
    exit(0);
}
x=seqStackPeek(StackOper); //获取栈顶操作符
}
q=seqStackPop(StackData);
seqStackfree(StackOper);
seqStackfree(StackData); //释放内存占用空间
return q; //出栈，返回结果
}

```

3、b 树:

主界面:

```
1.cpp
void Test2(){
    int i,k;
    //system("color 70");
    BTree t=NULL;
    Result s;
    while(1){
        printf("b tree \n");
        PrintBTree(t);
        printf("\n");
        printf("====Operation Table====\n");
        printf("  1.Init    2.Insert    3.Delete    \n");
        printf("  4.Destroy  5.Exit      \n");
        printf("====\n");
        printf("Enter number to choose operation:____\b\b\b");
        scanf("%d",&i);
        switch(i){
            case 1:{
                InitBTree(t);
                printf("InitBTree successfully.\n");
                break;
            }
        }
    }
}
```

Result SearchBTree(BTree t,KeyType k){

/*在树 t 上查找关键字 k,返回结果(pt,i,tag)。若查找成功,则特征值

tag=1,关键字 k 是指针 pt 所指结点中第 i 个关键字; 否则特征值 tag=0,

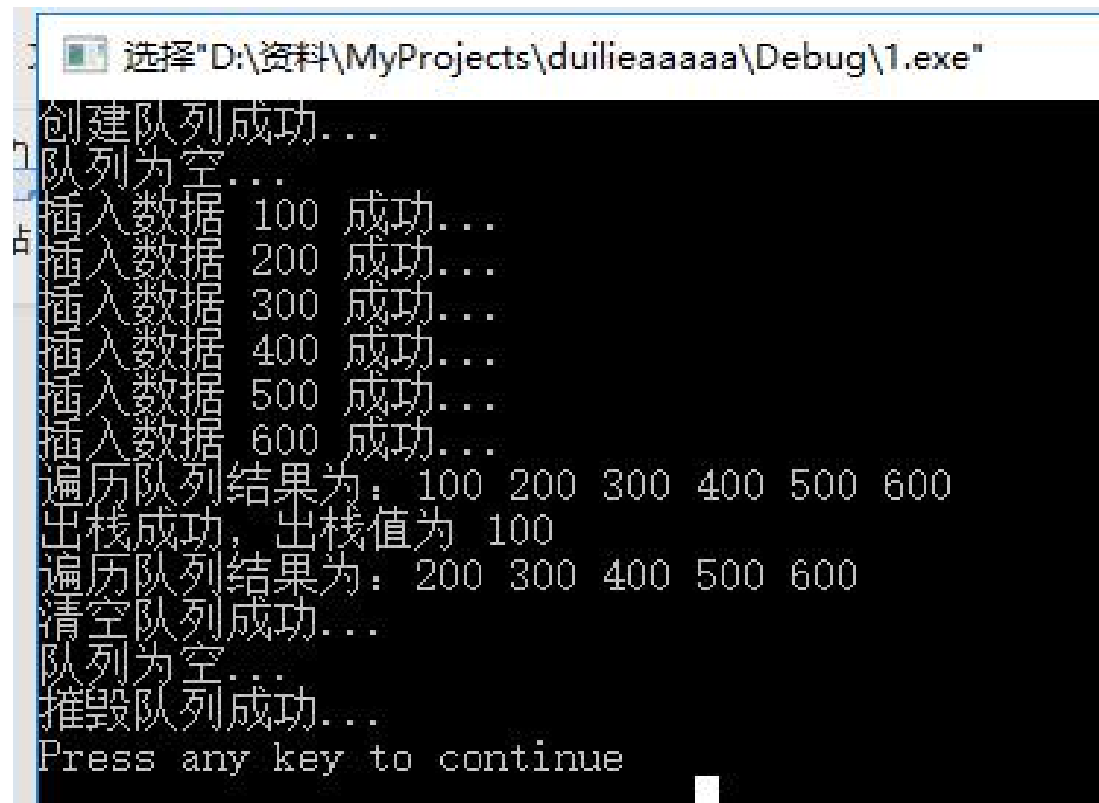
关键字 k 的插入位置为 pt 结点的第 i 个*/

```
    BTreeNode *p=t,*q=NULL;
    查结点,q 指向 p 的双亲
    int found_tag=0;
    int i=0;
    Result r;

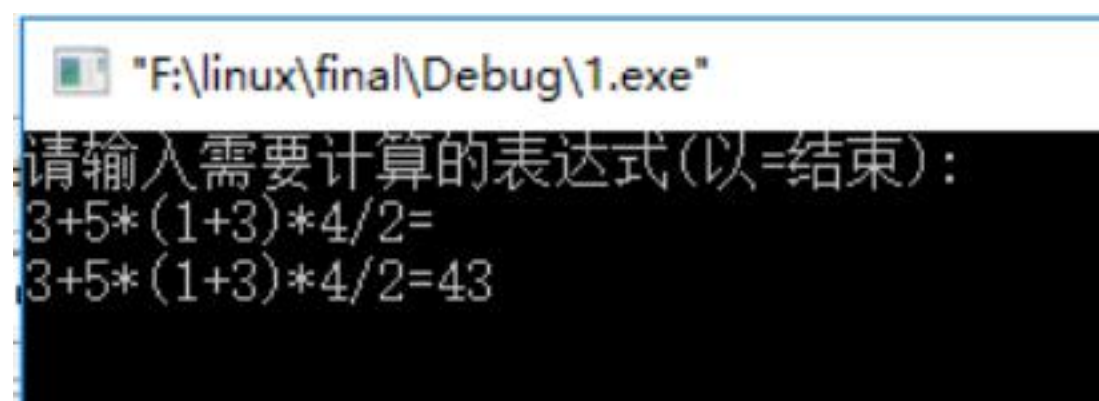
    while(p!=NULL&&found_tag==0){
        i=SearchBTreeNode(p,k);
        p->key[i]<=k<p->key[i+1]
        if(i>0&&p->key[i]==k)
            found_tag=1;
        else{
            q=p;
            p=p->ptr[i];
        }
    }
}
```

//初始化结点 p 和结点 q,p 指向待
//设定查找成功与否标志
//设定返回的查找结果
//在结点 p 中查找关键字 k,使得
//找到待查关键字
//查找成功
//查找失败

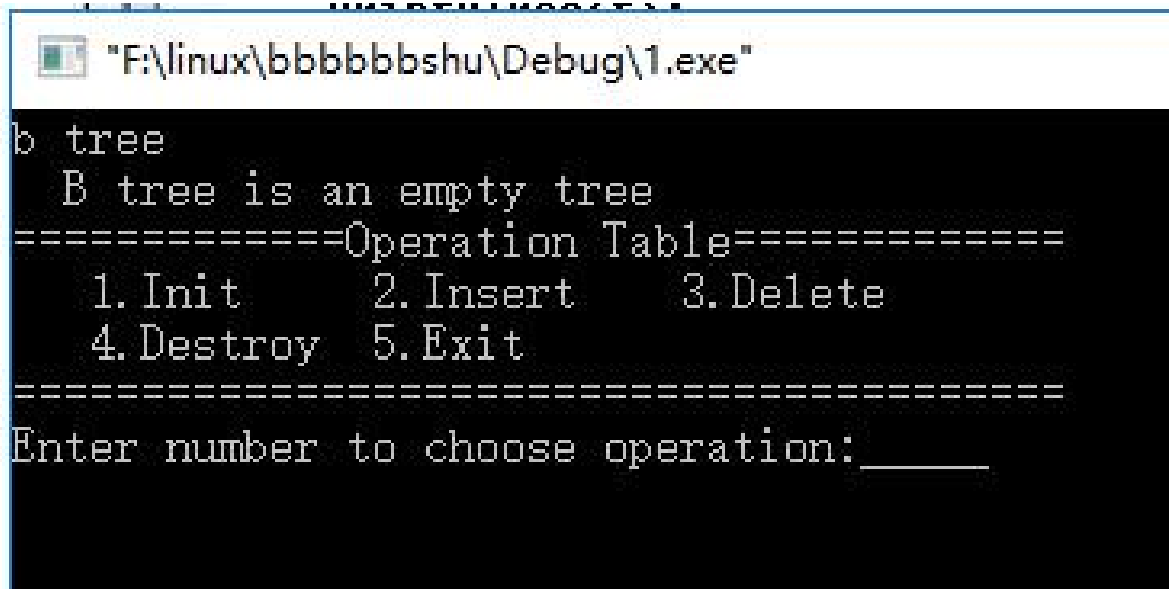
四、实验结果



```
选择"D:\资料\MyProjects\duilieaaaaa\Debug\1.exe"
创建队列成功...
队列为空...
插入数据 100 成功...
插入数据 200 成功...
插入数据 300 成功...
插入数据 400 成功...
插入数据 500 成功...
插入数据 600 成功...
遍历队列结果为: 100 200 300 400 500 600
出栈成功, 出栈值为 100
遍历队列结果为: 200 300 400 500 600
清空队列成功...
队列为空...
摧毁队列成功...
Press any key to continue
```



```
F:\linux\final\Debug\1.exe
请输入需要计算的表达式(以=结束):
3+5*(1+3)*4/2=
3+5*(1+3)*4/2=43
```

```
"F:\linux\bbbbbbshu\Debug\1.exe"
b tree
  B tree is an empty tree
=====Operation Table=====
    1.Init      2.Insert    3.Delete
    4.Destroy   5.Exit
=====
Enter number to choose operation:_____
```

五、实验总结

本次实验中通过 C 语言实现了队列、栈及 b+树的简单应用，从创建、插入、删除等一系列操作中达到实验目的，更加明确了各种数据结构的应用以及实现方法，也在 Linux 系统中实现了 C 语言代码的编译与执行，对 Linux 系统也有了一定的了解，能够熟练掌握基本操作，达到了实验最初的目的，也收获很多。