**《数据结构实验》报告**

|  |  |  |
| --- | --- | --- |
| **第 五 次实验** | **日期：2021.12.3** | **得分：** |
| **学号：** | **姓名：*Steven*** | **专业：智能科学与技术** |

**一、实验目的**

1. 练习C语言函数的使用方法。

2. 练习使用malloc、realloc、free等函数动态分配内存的方法。

3. 练习指针变量的使用，了解其用法。

4. 理解并熟练结构体的使用方法。

5. 理解并学会链表、串的构建和操作

**二、实验内容**

1、假设栈中数据元素类型是字符型， 请采用顺序栈实现栈的以下基本操作：

(1) Status InitStack(&S)//构造空栈S；

(2) Status Push(&S,e)//元素e入栈S；

(3) Status Pop(&S,&e)//栈s出栈，元素为e。

2、假设队列中数据元素类型是字符型， 请采用链队列实现队列的以下基本操作：

(1) Status InitQueue(&Q)//构造空队列Q；

(2) Status EnQueue(&Q, e)//元素e入队列Q；

(3) Status DeQueue(&Q, &e)//队列Q出队列，元素为e。

3、请实现： 对于一个可能包括括号{}、[]、()的表达式， 判定其中括号是否匹配。

4、设循环队列只设头指针Front，不设尾指针Rear，而是另设Count记录队列中元素个数。请实现：队列的入队和出队操作。

5、设以S和X分别表示入栈和出栈操作。根据仅由S和X构成的序列，对空栈进行操作，如果相应操作均可行（例如没有出现出栈时栈空）且最后栈是空栈，则称该序列是合法的栈操作序列。请实现：输入由S和X构成的序列，判断该序列是否是合法的栈操作序列。

**三、数据结构及算法设计**

1、

(1)初始化：

对于顺序栈，初始化即按照设定的默认栈的大小申请一个栈空间，同时设置栈顶指针和栈尾指针的地址。

(2)入栈：

先判断栈是否满，不满的话栈顶指针后移，并将入栈的值赋给栈顶指针的位置；

如果栈满则需要通过realloc扩容，并且由于其实质是寻找一个新的内存地址并将原内容依次拷贝过去，所以需要重新分配栈顶指针和栈底指针的地址。之后再重新执行入栈即可。

(3)出栈：

先判断栈是否空，若空返回ERROR即可，不空则将当前top指针的值赋给e，然后将top的值恢复为未赋值状态，并top后移。

2、

(1)初始化：

对于链队列，初始化即是创建头节点，并让头指针front和尾节点rear都指向该节点。

(2)入队：

由于链表对于内存是用多少申请多少，所以省去判断是否满的步骤，直接声明一个新的节点，将入队的值赋给它，并将该节点挂到队尾指针后面，然后队尾指针再后移即可。

(3)出队：

先判断队列是否为空，空的话return ERROR；不空的话则取出队首指针的值赋给e，然后front指针后移，并将原本的这个节点free掉。

3、

本题很明显用栈实现。首先调用之前定义的栈，然后遍历带括号的字符串。当出现左括号的时候，将括号入栈，当出现右括号的时候，将该右括号和出栈结果比较，如果匹配则继续进行，如果不匹配的直接return ERROR，（当然由于是调用的栈，如果对空栈进行出栈操作也会返回ERROR）

直到遍历过整个字符串后，如果栈内不空则仍然return ERROR；如果栈内也空了才能return OK；

4、

本题实际上操作的对象是数组的下标索引。

(1)初始化：

按照设定的默认队列的大小申请一个空间，同时设置队首下标和计数器Count。

(2)入队：

首先判断Count是否已经达到数组声明长度，达到则需要通过realloc对数组扩容，重新定义front指针位置，然后重新调用EnQueue函数。 如果没达到则执行入队操作。

入队：先通过队首位置和Count计算 到达数组尾前是否已经达到队尾，如果是则按照计算结果插入新的值；如果否的话说明队尾已经循环到数组首了，需要从头计算插入位置。

然后对计算出的数组位置进行赋值，更新Count。

(3)出队：

先判断是否队空，空则直接return ERROR；不空才执行出队。

由于存在头指针，则直接对e赋值头指针的对应值，然后头指针后移，Count自减，将数组种出栈值的部分修改为未定义的值。

5、

其实思路和第3题很像，大体思路也是遍历字符串，遇到S执行入栈、遇到X执行出栈。当出现：空栈要求出栈、栈满仍然入栈、循环完后栈内还有元素的其中任一种时，return ERROR；如果三种都不满足的话，return OK；

**四、核心程序代码**

1.

|  |
| --- |
| stack.h |
| #ifndef \_\_STACK\_H\_\_  #define \_\_STACK\_H\_\_  #include <stdio.h>  #include <stdlib.h>  #define STACK\_INIT\_SIZE 100 //初始容量  #define STACK\_INCREMENT 10 //增量  #define SElemType char //初始化数据元素类型  #define OK 0  #define ERROR -1  #define Status int//用以表示函数的操作状态  // 顺序栈  struct SqStack {  int stackSize = 0;//已用于存储的容量  int lenLimitation = 0;//数组的声明长度，栈的实际容量超出其值则需要扩容  SElemType\* S;//栈，本体  SElemType\* base = NULL;//存储基址  SElemType\* top = NULL;//栈顶指针  Status InitStack() {  if (S = (SElemType\*)malloc(STACK\_INIT\_SIZE \* sizeof(SElemType))) {  lenLimitation = STACK\_INIT\_SIZE;  for (int i = 0; i < STACK\_INIT\_SIZE; i++) {//为整个栈进行初始赋值  S[i] = 45;//45是“-”的ASCii  }  base = &S[0];  top = &S[0];  return OK;  }  else  return ERROR;  }  Status Push(SElemType val) {  if (stackSize < lenLimitation) {//判断栈没满  if (stackSize == 0) {//长度为0时直接往其中存入，top指针不需要移动  \*top = val;  stackSize++;  }  else {//栈长度不为0，先将top指针后移，然后再存入  top++;  \*top = val;  stackSize++;  }  }  else {//栈满  S = (SElemType\*)realloc(S, (STACK\_INIT\_SIZE + STACK\_INCREMENT) \* sizeof(SElemType));  base = &S[0];//重新定位base和top指针  top = &S[stackSize - 1];  lenLimitation += STACK\_INCREMENT;//修改stackSize的判断上限  Push(val);  }  return OK;  }  SElemType Pop(SElemType& e) {  if (stackSize != 0) {  stackSize--;  e = \*top--;  return OK;  }  else {  printf("栈空，无法出栈！");  e = ERROR;  return ERROR;  }  }  Status ShowStack() {  if (stackSize == 0) printf("当前栈为空\n");  else {  printf("当前栈内元素为：");  for (SElemType\* ptr = top; ptr >= base; ptr--) {  printf("%c", \*ptr);  if (ptr != base) {  printf("->");  }  }  printf("(由栈顶至栈底)\n");  }  return OK;  }  };  #endif |

2.

|  |
| --- |
| queue.h |
| #ifndef \_\_QUEUE\_H\_\_  #define \_\_QUEUE\_H\_\_  #include <stdio.h>  #include <stdlib.h>  #define QUEUE\_INIT\_SIZE 100 //初始容量  #define QUEUE\_INCREMENT 10 //增量  #define QElemType char  #define OK 0  #define ERROR -1  #define Status int//用以表示函数的操作状态  //链队列  typedef struct QNode {  QElemType data = 0;//数据域  struct QNode\* next = NULL;//指针域  }QNode, \* QueuePtr;  struct LinkQueue {  int queueSize = 0;  QueuePtr front = NULL;//队头指针  QueuePtr rear = NULL;//队尾指针  Status InitQueue() {  QNode\* node = (QNode\*)malloc(sizeof(QNode));  node->data = 0;  node->next = NULL;  front = node; //暂定队首和队尾指针都是这个头指针  rear = node;  return OK;  }  Status EnQueue(QElemType e) {  if (queueSize == 0) {  rear->data = e;  }  else {  QNode\* node = (QNode\*)malloc(sizeof(QNode)); //申请空间并赋值  node->data = e;  node->next = NULL;  rear->next = node; //将其挂入链表  rear = rear->next;  }  queueSize++;  return OK;  }  Status DeQueue(QElemType& e) {  if (queueSize != 0) {  e = front->data;  QNode\* removeNode = front;  front = front->next;  free(removeNode); //释放申请过的内存空间  queueSize--;  return OK;  }  else {  printf("队空，无法出队！");  return ERROR;  }  }  Status ShowQueue() {  if (queueSize == 0) printf("当前队为空\n");  else {  printf("当前栈内元素为：");  for (QNode\* ptr = front; ptr != rear; ptr = ptr->next) {  printf("%c->", ptr->data);  }  printf("%c", rear->data);  printf("(由队首至队尾)\n");  }  return OK;  }  };  #endif |

3.

|  |
| --- |
| matchBrackets.h |
| #ifndef \_\_MATCHBRACKETS\_H\_\_  #define \_\_MATCHBRACKETS\_H\_\_  #include "stack.h"  Status matchStr(char\* inputStr) {  SqStack S; //声明一个栈并初始化  S.InitStack();  char\* p = inputStr;  while (\*p != '\0') {  if (\*p == '(' || \*p == '[' || \*p == '{') {//左括号则存进去  S.Push(\*p);  }  else if (\*p == ')' || \*p == ']' || \*p == '}') {//出现疑似匹配的括号，则从栈中出一个进行匹配  char baseElem; S.Pop(baseElem); //取出栈顶元素  if (!((baseElem == '(' && \*p == ')') || (baseElem == '[' && \*p == ']') || (baseElem == '{' && \*p == '}')))//如果不是能匹配的任一种情况，则返回ERROR（当然如果是的话，if不执行，循环也就继续）  return ERROR;  }  p++;  }  if (S.stackSize != 0) return ERROR;//左右括号数量不匹配  return OK;  }  #endif |

4.

|  |
| --- |
| queue.h |
| #ifndef \_\_QUEUE\_H\_\_  #define \_\_QUEUE\_H\_\_  #include <stdio.h>  #include <stdlib.h>  #define QUEUE\_INIT\_SIZE 100 //初始容量  #define QUEUE\_INCREMENT 10 //增量  #define QElemType char  #define OK 0  #define ERROR -1  #define Status int//用以表示函数的操作状态  //循环队列  struct circleQueue {  int frontIndex;//记录front指针的index，以便于结合Count找到队尾  int lenLimitation = 0;//数组的声明长度，队列的容量超出它的话需要扩容  QElemType\* Q;//队列本体  QElemType\* front = NULL;//头指针  int Count = 0;//计数器  Status InitQueue() {  if (Q = (QElemType\*)malloc(QUEUE\_INIT\_SIZE\*sizeof(QElemType))) {  lenLimitation = QUEUE\_INIT\_SIZE;  for (int i = 0; i < QUEUE\_INIT\_SIZE; i++) {//初始化整个队列  Q[i] = 45;//45是“-”的ASCii，用它来表示未赋值  }  frontIndex = 0;  front = &Q[frontIndex];//设置头指针  return OK;  }  else  return ERROR;  }  int EndIndex() {  if (frontIndex + Count <= lenLimitation) {//队列没有循环  return frontIndex + Count - 1;  }  else {//队列已经循环过了  return frontIndex + Count - lenLimitation - 1;  }  }  Status EnQueue(QElemType val) {  if (Count == 0) {  Q[frontIndex] = val;  }  else if (Count < lenLimitation) {  Q[EndIndex() == lenLimitation - 1 ? 0 : EndIndex() + 1] = val;  }  else {//长度不够了  Q = (QElemType\*)realloc(Q, (QUEUE\_INIT\_SIZE + QUEUE\_INCREMENT) \* sizeof(QElemType));  front = &Q[frontIndex];//重新定位指针  lenLimitation += QUEUE\_INCREMENT;//更新数组的存入量上限  EnQueue(val);  }  Count++;  return OK;  }  QElemType DeQue(QElemType& e) {  if (Count != 0) {  e = Q[frontIndex];  Q[frontIndex] = 45;//回归未初始化状态  frontIndex++;  Count--;  return OK;  }  else {  printf("队空，无法出队！");  return ERROR;  }  }  Status ShowQueue() {  if (Count == 0) {  printf("当前队列为空\n");  return ERROR;  }  else {  printf("当前队内元素为：");  for (int num = 0; num < Count; num++) {  printf("%c", Q[frontIndex + num < lenLimitation ? frontIndex + num : frontIndex + num - lenLimitation]);  if (num != Count-1) {  printf("->");  }  }  printf("(由队首至队尾）\n");  }  return OK;  }  };  #endif |

5.

|  |
| --- |
| simulatedStack.h |
| #ifndef \_\_SIMULATEDSTACK\_H\_\_  #define \_\_SIMULATEDSTACK\_H\_\_  #include "stack.h"  Status simulatedStack(char\* inputStr) {  SqStack S;  S.InitStack();  char\* p = inputStr;  while (\*p != '\0') {  if (\*p == 'S') {//是'S'则存进去  S.Push(\*p);  }  else if (\*p == 'X') {//是'X'则出栈  char val; S.Pop(val);  if (val == ERROR) return ERROR;  }  p++;  }  if (S.stackSize != 0) return ERROR;//数量不匹配的情况  return OK;  }  #endif |

**五、测试及结果**

1. 测试用例：

定义顺序栈S，初始化栈，展示栈

依次入栈’a’ ’b’ ’c’，展示栈

依次出栈，按顺序展示出栈的元素

展示栈

文本

描述已自动生成

2. 测试用例：

定义链队列Q，初始化队列，展示队列

依次入队列’a’ ’b’ ’c’，展示队列

依次出队列，按顺序展示出队列的元素

展示队列

文本

描述已自动生成

3.

测试用例1："(ab)[{}]"

图形用户界面, 文本

描述已自动生成

测试用例2："({[)]}"

图形用户界面, 文本

描述已自动生成

测试用例3："(){[({}[])]}"

图形用户界面, 文本

描述已自动生成

4. 测试用例：

定义循环队列Q，初始化队列，展示队列

依次入队列’a’ ’b’ ’c’，展示队列

依次出队列，按顺序展示出队列的元素

展示队列

文本

描述已自动生成

5.

测试用例1："SXSXSSXX"

图形用户界面, 文本

描述已自动生成

测试用例2："SSSXXXXS"

图形用户界面, 文本

描述已自动生成