**《数据结构》上机报告**

**2018 年 11 月 2 日**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **姓名：** | **赵得泽** | **学号：** | **1753642** | **班级：** | **电子2班** | **得分：** |  |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **实验题目** | | 栈 | | |
| **问题描述** | | 栈是限制仅在表的一端插入和删除的线性表。栈的操作简单，重点掌握栈具有后进先出（LIFO）的特性。顺序栈是栈的顺序存储结构的实现。链栈是栈的链式存储结构的实现。 | | |
| **基本要求** | | 1. 练习顺序栈的基本操作，包括入栈、出栈、判栈空、判栈满、取栈顶元素、栈的遍历。 2. 栈的应用，实现任意数制之间的转换。 3. 检查其中的的括号是否都是配对的。假定需要判断的括号只有这三组：{}，[]，()。 4. 中缀表达式求值。 | | |
| **已完成基本内容（序号）：** | 1，2，3，4 | |
| **选做要求** | | 1. 列车进站问题。 | | |
| **已完成选做内容（序号）** | | 5 |
| **数据结构设计** | | **class SqStack**  **{**  **protected:**  **SElemType \*base;**  **SElemType \*top;**  **int stacksize;**  **public:**  **SqStack();**  **~SqStack();**  **Status ClearStack();**  **Status GetTop(SElemType&e);**  **Status Push(SElemType e);**  **Status Pop(SElemType&e);**  **};**  本实验用到的的数据结构类型是顺序栈，即线性表的顺序存储结构；它是用一组地址连续的存储单元依次存放栈顶到栈底的元素。其中包括指针域top和base, base用来指向栈底，也就是用来判断栈是否为空，top指针则是进行元素的入栈和出栈操作，还有一个int型数据stacksize是初始化设定的栈的大小。还有其他的成员函数则是栈的基本操作，包括初始化，销毁，清空，取栈顶元素，入栈出栈。通过栈的基本操作来实现来实现相应的功能。 | | |
| **功能(函数)说明** | | /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  函数功能：判断栈是否为空  说明：当栈顶=栈底时，栈为空。  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  bool IsEmpty(SeqStack &s)  {  if (s.top == s.base)  return true;  else  return false;  }  /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  函数功能：判断栈满  说明：当栈顶-栈底>=栈的初始分配的空间时，栈满。  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  bool IsFull(SeqStack&s)  {  if (s.top - s.base >= s.stacksize)  return true;  else  return false;  }  /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  函数功能：出栈  说明：如果栈空，则不能出栈；若栈未空，则栈顶指针减1，输出栈顶元素。  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  void PopStack(SeqStack &s)  {  if (IsEmpty(s))  cout << "Stack is Empty" << endl;  else  cout << \*--s.top << endl;  }  /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  函数功能：进制转换  输入参数：初始转换进制n,目的转换进制m  说明：将读入的数字作为字符(串)压栈，然后通过循环从最低位读取，即一个一个  出栈，用初始进制累加，同时随着数位的增加，初始进制也要累乘，保证累加的正  确性，直到栈空，累加结束，从而将n进制数转换为十进制数，然后再用除m取余的  方法，将得到的m进制的每位的数转换成字符（r <= 9 ? '0' + r : 'A' + r - 10）压栈，结束之后，再出栈，这样就将n进制数转换成了m进制数。  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  void convert\_mTOn(SeqStack &s, int m, int n, char str[])  {  int x, temp = 0, y = 1;  for (int i = 0; i < strlen(str); i++)  PushStack(s, str[i]);  while (!IsEmpty(s))  {  s.top--;  if (\*s.top >= '0'&&\*s.top <= '9')  x = \*s.top - '0';  if (\*s.top >= 'A'&&\*s.top <= 'Z')  x = \*s.top - 'A' + 10;  temp = temp + y \* x;  y \*= m;  }  while (temp)  {  int r = temp % n;  PushStack(s, (r <= 9 ? '0' + r : 'A' + r - 10));  temp /= n;  }  }  /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  函数功能：括号匹配  输入参数：待检测的括号序列s0[]  说明：括号配对的前提是输入的括号种类有三种，即(),[],{};在检测是否匹配时，  若第一个括号为右括号),],}，则直接结束判断（缺少左括号）；其他情况下，若是  左括号则全部压栈，当出现第一个右括号时，将栈顶的左括号和其比较，若是不匹配，则结束判断（栈顶的括号期待相应的右括号）；否则，将栈顶的括号出栈。继续进行输入，按照前面的步骤循环，左压，右比较，直到括号序列结束。若是整个括号序列是配对的，则最终栈为空。  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  void Match(SeqStack &s, char s0[])  {  int i = 0;  if (s0[0] == '}' || s0[0] == ']' || s0[0] == ')')  {  cout << "no" << endl;  cout << s0[0] << "期待左括号" << endl;  return;  }  else  {  PushStack(s, s0[0]);  i++;  while (s0[i] != '\0')  {  int flag = 0;  if ((s0[i] == '}'&&\*(s.top - 1) == '{') || (s0[i] == ']'&&\*(s.top - 1) == '[') || (s0[i] == ')'&&\*(s.top - 1) == '('))  {  PopStack(s);  flag = 1;  }  else if (s0[i] == '{' || s0[i] == '[' || s0[i] == '(')  PushStack(s, s0[i]);  else if (((s0[i] == '}'&&\*(s.top - 1) != '{') || (s0[i] == ']'&&\*(s.top - 1) != '[') || (s0[i] == ')'&&\*(s.top - 1) != '(')) && !IsEmpty(s))  {  cout << "no" << endl;  cout << \*--s.top << "期待右括号" << endl;  return;  }  else if (IsEmpty(s) && (s0[i + 1] != '\0') && (s0[i] == '}' || s0[i] == ']' || s0[i] == ')'))  {  cout << "no" << endl;  cout << s0[i] << "期待左括号" << endl;  return;  }  i++;  if (flag == 1 && s0[i] != '\0')  {  if (s0[i] == '{' || s0[i] == '[' || s0[i] == '(')  {  PushStack(s, s0[i]);  i++;  }  if (IsEmpty(s) && s0[i + 1] == '\0' && (s0[i] == ')' || s0[i] == '}' || s0[i] == ']'))  {  PushStack(s, s0[i]);  i++;  }  }  }  if (s.top == s.base)  cout << "yes" << endl;  else  {  cout << "no" << endl;  switch (\*--s.top)  {  case '{':cout << "{期待右括号" << endl;  break;  case '[':cout << "[期待右括号" << endl;  break;  case '(':cout << "(期待右括号" << endl;  break;  case '}':cout << "}期待左括号" << endl;  break;  case ']':cout << "]期待左括号" << endl;  break;  case ')':cout << ")期待左括号" << endl;  break;  }  }  }  }  /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  函数功能：判断列车出站的序列是否ok  输入参数：列车入站序列InOdr,待判断列车出站序列Odr  说明：首先进行判断，若两者长度不相等，则直接结束判断；否则，再进行比较判  断。  比较时，首先从入站序列的首元素开始和出站序列比较，  若相等，两序列分别前移一位，  若不相等，再和栈顶元素比较；  若相等，将该元素出栈，出站序列前移一（j++），  若此时j等于入站序列的长度，说明出站序列正确，结束循环，  若不相等，  如果此时i已经等于入站序列的长度，则该出站序列不ok;  如果不等，则将该入站元素压栈。i++；  继续执行此循环，直到结束循环。  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  Status Judge\_OK(SqStack &s, char Odr[], char InOdr[], int iLen)  {  int i = 0, j = 0;  SElemType e;  while (1)  {  if (InOdr[i] == Odr[j])  {  i++;  j++;  if (j == iLen)  return OK;  }  else if (s.GetTop(e) && e == Odr[j])  {  s.Pop(e);  j++;  if (j == iLen)  return OK;  }  else  {  if (i == iLen)  return ERROR;  s.Push(InOdr[i]);  i++;  }  }  }  /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  函数功能：将中缀表达式转换为后缀表达式  说明：如果当前读入的时操作数就将其存在Back\_exp数组中，如果当前字符为  操作符，记为x1,将x1与栈顶的运算符x2比较，若栈顶运算符优先级小于当前  运算符优先级，就将当前运算符入栈，否则，将栈顶运算符出栈保存在  Back\_exp数组中。之后继续比较新的栈顶运算符和当前运算符的优先级  若两者优先级相等，且x1=')’，x2=‘(',则将x2出栈，继续读入下一个字符；  若读到“=”,则结束，后缀表达式就存在Back\_exp中。  注：在存数字时，每个数之间隔个空格，以便于计算一位数以上的数的值；  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  void ConvertExp(SeqStack &s, char m[], char b[], int & flag)  {  int i = 0, k = 0;  char c, c1;  c = m[i];  while (c != '=')  {  if (c == '+' || c == '-')  {  while (!IsEmpty(s) && GetTop(s, c1) && c1 != '(')  {  PopStack(s);  b[k++] = c1;  }  PushStack(s, c);  }  else if (c == '\*' || c == '/')  {  while (!IsEmpty(s) && GetTop(s, c1) && (c1 == '\*' || c1 == '/'))  {  PopStack(s);  b[k++] = c1;  }  PushStack(s, c);  }  else if (c == '(')  PushStack(s, c);  else if (c == ')')  {  while (GetTop(s, c1) && c1 != '(')  {  PopStack(s);  b[k++] = c1;  }  PopStack(s);  }  else if (c >= '0'&&c <= '9')  {  while (c >= '0'&&c <= '9')  {  b[k++] = c;  c = m[++i];  }  i--;  b[k++] = ' ';  }  else  {  flag = 1;  return;  }  c = m[++i];  }  while (!IsEmpty(s))  {  GetTop(s, c1);  PopStack(s);  b[k++] = c1;  }  b[k] = '\0';  }  /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  函数功能：计算后缀表达式的值  说明：对BACK\_exp数组进行遍历，如果当前读入的字符是操作数则将它放到opd  栈（操作数栈），如果当前读入的字符是操作符，则opd栈出栈两次，得到操作  数x,y,进行运算，然后将结果继续存到opd栈。重复执行此操作直到opd栈为空，  Back\_exp数组到末尾，将结果opd.val[--opd.top]返回即可；  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  int Calculate(opdStack& opd, char b[])  {  int i = 0, value = 0, tmp = 0;  int v1 = 0, v2 = 0;  char c = b[i];  while (c != '\0')  {  value = 0;  switch (c)  {  case '+':  v2 = --opd.top;  v1 = --opd.top;  tmp = opd.val[v1] + opd.val[v2];  opd.val[opd.top] = tmp;  opd.top++;  break;  case '-':  v2 = --opd.top;  v1 = --opd.top;  tmp = opd.val[v1] - opd.val[v2];  opd.val[opd.top] = tmp;  opd.top++;  break;  case '\*':  v2 = --opd.top;  v1 = --opd.top;  tmp = opd.val[v1] \* opd.val[v2];  opd.val[opd.top] = tmp;  opd.top++;  break;  case '/':  v2 = --opd.top;  v1 = --opd.top;  if (opd.val[v2] == 0)  return 0;  tmp = opd.val[v1] / opd.val[v2];  opd.val[opd.top] = tmp;  opd.top++;  break;  default:  while (b[i] != ' ')  {  value = value \* 10 + (b[i] - '0');  i++;  }  opd.val[opd.top++] = value;  }  c = b[++i];  }  return opd.val[--opd.top];  } | | |
| **开发环境** | | Win10,Vs2017,c++ 高级程序语言设计 | | |
| **调试分析** | | 5.    4.    3.      2.    1. | | |
| **心得体会** | 本次实验主要是练习栈的使用。栈的主要特点是**“后进先出”（LIFO）**,利用这个特点主要进行了关于栈的基本操作，包括出入栈，判栈空栈满，栈的遍历等基本操作，  还有和它特点恰好相对应的应用（括号配对，中缀表达式求值，列车入站问题等）。  通过这次栈的应用，我对栈的特点有了除了理论之外的认识。  在**括号配对**的算法中，就很好的体现了栈的方便之处，将左括号压栈，直到碰到第一个右括号再进行配对检测，即和栈顶的括号匹配，若不匹配，则结束判断，否则将栈顶括号弹栈，对新栈顶的括号进行配对检测。这样一直循环，若栈最终是空的，则证明该括号序列是匹配的。  在**中缀表达式求值**的算法中，我的思路是将中缀表达式转化为可以利用栈进行运算的后缀表达式，然后对后缀表达式进行求值。首先对中缀表达式序列进行遍历，若遇到操作数将其存到后缀表达式数组中（注：为了好区分一位数还是高于一位数，要在每两个操作符之间的数中间加上**区分标志‘#’**），若遇到操作符则进行压栈，在根据优先级顺序进行出栈入栈操作（优先级要首先明确），从而利用栈完成转化操作。接下来的求值操作中，还是利用栈的特点，首先对后缀表达式数组进行遍历，遇到操作数将其转化为int型数字，存到另一个操作数栈，遇到操作符则将出栈两次进行运算，再将结果压栈，这样，将后缀表达式遍历完之后操作数栈就仅剩栈底的结果，那这个结果就是表达式的结果。  在**列车入站**问题中，栈的应用则是更加上了一个层次，具体方法见上**函数功能说明，**  在本题中，栈的LIFO特性的体现，让我深刻认识到栈对某些看似复杂问题的求解时如此的简便。  通过以上训练，我意识到对问题的分析要透彻，要根据问题的特点抽象出我们已知的数据结构，然后利用此数据结构进行算法求解，这也要求我们对其特点的独到性进行彻底地理解和分析，才能运用自如。 | | | |