**实验3 堆栈**

目标

在这个实验中

* 创建两个堆栈的实现一个基于数组表示，另一个基于单链表表示。
* 编写程序计算后缀形式的算术表达式。
* 编写一个可以正确地处理成对的小括号和大括弧的计算表达式的程序。
* 分析通过使用一个堆栈可以提供的排列种类。

概述

堆栈是一种限定的线性数据结构的例子。在一个堆栈中，数据项是从最后人栈（栈顶top）到最先入栈（栈底bottom）排列，所有的插人和删除运算都限定在栈顶。我们可以使用Push运算在栈顶插入一个数据项，或使用pop运算删除一个栈顶数据项，一个push和pop运算的序列如下所示。

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Push a | Push b | Push c | Pop | Pop |
|  |  | c |  |  |
|  | b | b | B |  |
| a | a | a | a | a |
| \_ | \_ | \_ | \_ | \_ |

这些插入和删除运算的限制造成了**“后进先出”**( LIFO）的行为，这是堆栈的特性。虽然堆栈数据结构很简单，但是，它是系统软件中以及计算机体系结构中使用十分频繁的要素之一。堆栈是一种被频繁使用的数据结构。

堆栈结构

* 数据项

一个堆栈中数据项的数据类型是一般类型ElemType。

* 结构

堆栈中数据项是从最后入栈（栈顶）到最先入栈（栈底）线性排列的。数据项在栈顶插入（pushed）或删除（popped）。

* 运算

**InitStack( int maxNumber=defMaxStacksize )**

**要求：**无

**结果：**创建一个空的堆栈。为一个包含maxNumber个数据项的堆栈分配足够的内存（如有必要）。

**DestroyStack ( )**

**要求：**无

**结果：**释放(free)存储一个堆栈的内存。

**void Push ( ElemType &newDataItem )**

**要求：**堆栈非满。

**结果：**把newDataItem 插人到栈顶。

**DT pop ( )**

**要求：**堆栈非空。

**结果：**删除最后加入到栈顶 (top)的数据项并且返回。

**void clear ( )**

**要求：**无

**结果：**删除堆栈中所有数据项。

**bool isEmpty ( )**

**要求：**无

**结果：**如果堆栈为空，则返回true，否则，返回false。

**bool isFull ( )**

**要求：**无

**结果：**如果栈满，则返回true，否则，返回false。

**void showStructure ( )**

**要求：**无

**结果：**输出堆栈中的数据项。如果堆栈为空，输出“Empty stack”。

**实验3 作业单**

姓名： 日期：

请在教师布置的练习对应的已布置列上打一个钩（√）。在提交这个实验的一组材料前面附上这个作业单。

|  |  |  |
| --- | --- | --- |
| **练习** | **已布置：打钩或**  **列出练习编号** | **已完成** |
| 实验前练习 | √ | √ |
| 过渡练习 | √ | √ |
| 实验中练习1 | √ | √ |
| 实验中练习2 | √ | √ |
| 实验中练习3 | √ | √ |
| 实验后练习1 | √ | √ |
| 实验后练习2 | √ | √ |
| 总计 |  |  |

**实验2 实验前练习**

姓名： 日期：

在这个实验中，我们开发两种堆栈ADT的实现。一种采用数组存储堆栈，另一种采用链表构成一个堆栈。

**第一步：**使用一个数组存储堆栈数据项实现堆栈中的运算。堆栈大小可变，因此，需要存储堆栈中可以存储数据项的最大数目（maxSize) ， 最顶端数据项的数组索引(top) ，以及堆栈数据项本身（dataItems）。在文件stacksql.h保存堆栈地类型定义。

**int defMaxStackSize = 100; //Default maximum stack size**

**struct Stack {**

**int maxSize; //Maximum number of data items in the stack**

**int top; //Index of the top data items**

**ElemType \*dataItems;**

**};**

**void InitStack( struct Stack S, int maxNumber );**

**void DestroyStack( struct Stack S );**

**void Push( struct Stack S, ElemType &newDataItem );**

**ElemType Pop( struct Stack S );**

**void clear( struct Stack S );**

**bool isEmpty( struct Stack S );**

**bool isFull( struct Stack S ); l**

**void showStructure( struct Stack S );**

**第二步：**在文件stacksql.cpp中保存堆栈的数组实现，并确认形成代码文档。

**第三步：**通过使用一个单链表存储堆栈数据项，实现堆栈中的运算。链表的每一个结点应该包括一个堆栈数据项（dataItem）和一个指向堆栈中下一个数据项结点的指针。我们的实现还应该包含一个指向堆栈的最顶部数据项节点的指针（top）。在文件stacklnk.h中保存堆栈的链表实现类型定义。

**struct StackNode {**

**ElemType dataltem; // Stack data item**

**struct StackNode \*next; // Pointer to the next data item**

**};**

**struct Stack {**

**int Size;**

**struct StackNode \*top;**

**};**

**void InitStack( struct Stack S, int maxNumber );**

**void DestroyStack( struct Stack S );**

**void Push( struct Stack S, ElemType &newDataItem );**

**ElemType Pop( struct Stack S );**

**void clear( struct Stack S );**

**bool isEmpty( struct Stack S );**

**bool isFull( struct Stack S ); l**

**void showStructure( struct Stack S );**

**第四步：**把堆栈的链表实现保存在文件stacklnk.cpp中，确认形成代码文档。

**实验2 过渡练习**

姓名： 日期：

在文件teststack.cpp中完成测试程序。测试程序使用下面的命令交互式地测试堆栈的实现。

|  |  |
| --- | --- |
| 命令 | 操作 |
| +x | 压入数据项x到栈顶 |
| - | 弹出栈顶数据项并输出 |
| E | 报告堆栈是否为空 |
| F | 报告堆栈是否已满 |
| C | 清空堆栈 |
| Q | 退出测试程序 |

**第一步：**编译并链接测试程序。

**第二步：**增加测试项目完成下面的测试计划

* 从一个仅包含一个数据项的堆栈中弹出一个数据项。
* 向一个经过一系列弹出运算已为空的堆栈中压人一个数据项。
* 从一个已满的堆栈中弹出一个数据项（数组实现）。
* 清空堆栈。

**第三步：**执行测试程序，如果在堆栈的数组实现中发现了错误，改正这些错误并且重新执行测试计划。

**第四步：**修改测试程序，采用堆栈的链表实现替代数组实现。

**第五步：**重新编译并链接测试程序。

**第六步：**使用测试程序检查堆栈的链表实现，如果在堆栈的链表实现中发现了错误，改正这些错误并且重新执行测试计划。

**堆栈中运算的测试计划**

|  |  |  |  |
| --- | --- | --- | --- |
| 测试项目 | 命令 | 预期结果 | 检查 |
| 一系列压入运算  一系列弹出运算  多压入运算  多弹出运算  栈空？栈满？  清空堆栈  栈空？栈满？ | +a +b +c +d  - - -  +e +f  - -  E F  -  E F | a b c **d**  **a**  a e **f**  **a**  false false  空堆栈  true false | √  √  √  √  √  √  √ |

注：栈顶数据项以黑体显示

**实验2 实验中练习1**

姓名： 日期：

我们一般使用中缀形式书写算术表达式，即每一个运算符在它的运算数中间，如下面的表达式：(3+4)\*(5/2) 。

虽然我们习惯于表达式中缀形式，但是，中缀形式有一个缺点，必须使用圆括号指定运算顺序，这些圆括号极大地增加了运算过程的复杂性。

如果我们简单地从左到右对运算数进行运算，那么运算会简单的多。遗憾的是，这种从左到右的运算策略不能应用在中缀形式的算术表达式上。不过，如果采用后缀形式就可以达到这一目的。在后缀形式的算术表达式中，每一个运算符紧跟在它的运算数之后，上面的表达式的后缀形式可以写成：34 + 52/\*。

我们不难注意到，在这两种形式中，运算数的次序是相同的（从左到右地读），而运算符的次序不同。

**后缀形式的运算符的次序和他们运算的次序是相同的，后缀形式的表达式的结果难以读取，但是容易运算**。所以我们需建立一个堆栈存放中间结果。

假设我们有一个后缀形式的算术表达式，包括单数字的非负整数和四个基本的算术运算符（加，减，乘和除）。这个表达式可以通过一个**数字堆栈**和相关的**运算法则**进行运算。

* 逐字符读表达式，每一个字符被读取时,做以下工作：
* 如果字符是一个单数字的（字符‘0'到‘9' ), 则把相应的数字存人堆栈中；
* 如果字符是一个算术运算符（字符‘+’,‘-’,‘\*’,‘/’)，那么从堆栈中弹出一个数字，称为operand1；
* 从堆栈中弹出一个数字，称为operand2；
* 使用算术运算符计算这两个运算数，Result = operand2运算符operand1；
* 把result压入堆栈。当表达式结束时，从堆栈中弹出剩下的数字，这个数字就是表达式的值。

例如，对34+52/\*应用这个运算法则，产生下面的计算：

‘3 ’: 压入3.0

‘4 ’: 压入4.0

‘+ ’: 弹出，operand1=4.0

弹出，operand2=3.0

计算，result=3.0+4.0=7.0

压入7.0

‘5 ’: 压入5.0

‘2 ’: 压入2.0

‘/ ’: 弹出，operand1=2.0

弹出，operand2=5.0

计算，result=5.0/2.0=2.5

压入2.5

‘\* ’: 弹出，operand1=2.5

弹出，operand2=7.0

计算，result=7.0\*2.5=17.5

压入17.5

‘\n ’: 弹出，表达式的值=17.5

**第一步：**创建一个应用程序，该程序可以读取一个后缀形式的算术表达式，进行运算，并且显示结果。假设这个后缀形式的算术表达式，包括单数字的、非负的整数（‘0 ’到‘9 ’,）和四个基本的算术运算符（‘+ ’,‘- ’, ‘/ ’,和‘/ ’）。还可以假设算术表达式从键盘输人并且所有的字符都在一行，把程序保存在postfix.cpp文件中。

**第二步：**填充每一个算术表达式的预期结果，完成下面的测试计划。

**第三步：**执行测试计划。如果发现程序中有错误，改正这些错误并且重新执行测试计划。

后**缀形式的算术表达式的运算程序测试计划**

|  |  |  |  |
| --- | --- | --- | --- |
| 测试项目 | 算术表达式 | 预期结果 | 检查 |
| 一个运算符  嵌套运算  不对称运算  所有运算符在最后  零除  单个数字 | 34+  34+52/\*  34+52/\*  4675-+\*  02/  7 | 7  17.5  17.5  32  0  7 | √  √  √  √  √  √ |

**实验2 实验中练习2**

姓名： 日期：

相对于一个堆栈从数组下标0到下标maxSize-1的向上的数组实现，可以构造一个从数组下标maxSize-1到下标0的向下的数组实现，把这种向下的数组实现和在实验前创建的向上的数组实现结合起来，形成一个双堆栈的实现，使两个堆栈共用同一个数组。假设两个堆栈的数据项数之和不会超过maxSize。这就是一个双堆栈的奇异类型，在操作系统中进程的处理就采用了该方式。

**第一步：**文件stacksql.h中的定义相同，创建一个堆栈的向下增长的数组实现。

**第二步：**在文件stackdwn.cpp中，实现堆栈的向下的数组操作。

**第三步：**修改测试程序stacktest.cpp，使用文件stackdwn.cpp中堆栈的向下的数组实现代替堆栈的向上的数组实现。

**第四步：**使用在实验中练习1中创建的测试计划检查堆栈的向下数组实现。如果在实现中发现错误，修改这些错误并且重新执行测试计划。

**后缀形式的算术表达式的运算程序测试计划**

|  |  |  |  |
| --- | --- | --- | --- |
| 测试项目 | 算术表达式 | 预期结果 | 检查 |
| 一个运算符  嵌套运算  不对称运算  所有运算符在最后  零除  单个数字 | 34+  34+52/\*  34+52/\*  4675-+\*  02/  7 | 7  17.5  17.5  32  0  7 | √  √  √  √  √  √ |

**实验2 实验中练习3**

姓名： 日期：

编译器和解释器的任务之一是频繁地执行判断操作，判别表达式分界符是否正确配对，即使表达式中使用了多重嵌套配对，也能正确判断。例如对于表达式：

a = ( f[b]-(c+d) ) / 2;

编译器必须能够判断成对的开和闭的圆括号、方括号等，继而判断整个表达式是否正确地分界。

对于不配对的分界符，或分界符使用的位置不对的表达式都会引发一系列可能的错误。例如，下面的表达式缺少一个闭的圆括号。

a = ( f[b]-(c+d) / 2;

下面的表达式也是无效的，虽然圆括号和方括号的数目是正确的，但是匹配不正确。第一个闭圆括号不能与最近的开分界符（方括号）相匹配。

a = ( f[b)-(c+d] ) / 2;

由于堆栈具有LIFO(后进先出)的特性，所以堆栈对实现解决这种类型的问题特别有帮助。一个闭的分界符必须正确地匹配最近遇到的开分界符。处理方法是**把遇到的开分界符压入一个堆栈中，但遇到一个闭分界符时，从对堆栈中弹出的应该是一个匹配的开分界符。如果每一个闭分界符都有一个匹配的开分界符，那么整个表达式是有效的。**

**bool delimitersOk( string &expression )**

**要求：**无

**结果：**如果表达式中所有的圆括号和方括号都是合理的配对，则返回true；否则，返回false。

**第一步：**在程序文件delim.cpp 中实现delimitersOk运算。

**第二步：**增加测试项目完成下面的测试计划。检查delimtersOk运算是否能够正确地检查输人表达式中分界符是否正确配对。请注意，输人不需要是合法的表达式，只是检查分界符使用是否正确。

**第三步：**执行测试计划。如果发现delimitersOk运算的实现中有错误，改正这些错误并且重新执行测试计划。

**delimitersOk运算的测试计划测试项目**

|  |  |  |  |
| --- | --- | --- | --- |
| 测试项目 | 命令 | 预期结果 | 检查 |
| 带圆括号的有效表达式  带混合分界符的有效表达式  带混合分界符的无效表达式  空的表达式  不配对的方括号 | 3 \* ( a + b )  f [3 \* (a + b)]  (f[b) - (c + d]) / 2  空串<新行>  a = f [ b + 3 | true  true  false  true  false | √  √  √  √  √ |

注：没有正确匹配的分界符以黑体显示

**实验2 实验后练习1**

姓名： 日期：

给出一个输人字符串“abc”，可以通过不断的使用Push操作和Pop操作的任意组合形成相关的输出序列，从而可以判断这个字符串的输出结果。

cin>>ch; Push(permuteStack,ch);

和

ch= Pop(permuteStack); cout<<ch;

其中，ch是一个字符，permuteStack是一个字符类堆栈。

例如，代码段

cin>>ch; Push(permuteStack,ch);

cin>>ch; Push(permuteStack,ch);

cin>>ch; Push(permuteStack,ch);

ch = pop(permuteStack); cout<<ch;

ch = pop(permuteStack); cout<<ch;

ch = pop(permuteStack); cout<<ch;

输出字符串“cba”。

A 部分

对下面表中的每一种排列次序，给出一段Push和Pop组合，输出这些排列或者简要说明为什么不能产生这种排列次序。

“abc” PushPopPushPopPushPop “acb” PushPopPushPushPopPop

“bac”PushPushPopPopPushPop “bca”PushPushPopPushPopPop

“cab”先输出c，则ab已进站，输出ba “cba”PushPushPushPopPopPop

B部分

给出一个输入字符串“abcd”，哪些以‘d ’开头的四字符排列可以由Push和Pop组合产生输出？为什么只能产生这几种排列次序？

dcba 以d开头则所有已按顺序进栈，出站只能按dcba

**实验3 实验后练习2**

姓名： 日期：

在实验中练习1 ，我们使用过一个堆栈计算算术表达式。试描述堆栈的其他应用情况。在这个应用中，每一个堆栈数据项存储何种类型的信息？

背包问题中记录上一次 的状态。