COMP102

Part 1

Data Types and Structures

Lecture 7

Stacks

Linked Lists provide a general mechanism implementing a dynamically adaptable structure.

There are many applications where a strict regime governing the order in which data can be accessed is needed.

Examples are:

1. The **L**ast item placed **I**nto the structure is the **F**irst item taken **O**ut. (**LIFO**, or Last-In-First-Out)
2. The **F**irst item placed **I**nto the structure is the **F**irst item taken **O**ut. (**FIFO**, or First-In-First-Out)
3. The first and last items placed can be obtained ‘immediately’, but not any items between these, e.g. if three items were input in the order

   Item A :: Item B :: Item C

   then this provides access to A and C, but not (immediate) access to B.

The form prescribed by the LIFO regime, is that realised by the LinkedList class.

The ADTs used to hold data so that access to these is controlled by the different protocols described have specific names in a Computer Science context.

1. A structure in which data are held/obtained using a LIFO regime is known as a **Stack**.
2. One in which data are held/obtained using a FIFO regime is known as a **Queue**.
3. One in which the most recently added and longest resident item can both be obtained is known as a **Double-Ended Queue** or **Deque**.

Stacks have already been seen in the context of low-level programming on COMP103.

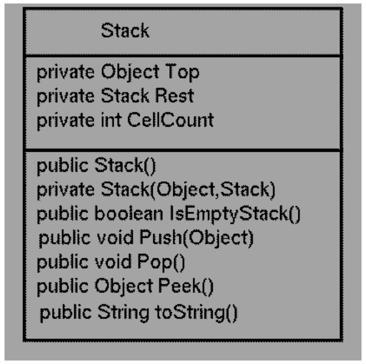
In this lecture the standard set of operations associated with Stacks, and a realisation in Java are examined.

Stacks

The Stack ADT is frequently used in systems software, e.g. Compilers, Operating Systems.

Typical applications of Stacks in the former case:

1. In compilers for *imperative* language supporting **recursive methods**, Stacks are used to aid in resolving references to **local** parameters over successive recursive calls.
2. In order to evaluate *arithmetic expressions*, many compilers first transform these into a notation known as **Reverse Polish Notation** which is parenthesis free. Stacks give a simple algorithm for processing and evaluating arithmetic expressions in RPN.



Notice the similarity to the ListCell class.

The fields of a Stack instance:

An Object **Top**: the item currently at the top of this instance.

A **Stack Rest**: a pointer to the rest of this **Stack**.

An **int CellCount**: the number of elements in this instance.

The first two of are similar to the fields **Datum** and **Link** in the ListCell class.

Of the two constructors, only one

public Stack()

is visible to applications importing the class.

The other,

private Stack(Object, Stack)

is only available in Instance methods of the class.

Its implementation is similar to the corresponding constructor of ListCell.

Using 2 constructors, one of which is hidden, obviates nesting the ListCell class details in the Stack class, while still avoiding the drawbacks that led to this being done for the LinkedList realisation.

Instance methods

The important operations on a Stack are

1. public boolean IsEmptyStack();

   (tests if the Stack is empty)
2. public void Push(Object T);

   (places a new Object, **T**, onto the Stack)
3. public void Pop();

   (removes the Object at the top of the Stack)
4. public Object Peek();

   (returns the Object at the top of the Stack but without altering the Stack)


```
public class Stack
  {
  //***************
  // Stack Fields *
  //***************
  private Object Top;
  private Stack Rest;
  private int CellCount;
  //*******************
  // Stack Constructor*
  //*******************
  public Stack()
    {
    Top  = null;
    Rest = null;
    CellCount=0; // Initiates the empty Stack
    }
  private Stack( Object Item, Stack rest_of )
     {
     Top = Item;
     Rest = rest_of;
     }


//*****************************
//  Test if instance is Empty.*
//*****************************
public boolean IsEmptyStack()
  {
  return (CellCount==0);
  }
```

```
//***********************************
//  Add a new Cell as to this Stack *
//***********************************
public void Push ( Object Datum )
  {
  if (IsEmptyStack())
    {
    Top =Datum;
    Rest = null;
    }
  else
    {
    Rest = new Stack (Top,Rest);
    Top = Datum;
    };
  CellCount++;
 }
```



```
//**********************************
// Remove the Cell at top of Stack *
// Current Stack is [Top]::[Rest], *
// the 'new' one is [Rest].        *
//**********************************
public void Pop()
  {
  if (CellCount<=1)
    {
    Top = null;
    Rest=null;
    CellCount=0;
    }
  else
    {
    Top = Rest.Top;
    CellCount--;
    if (CellCount==1)
      Rest=null;
    else
      Rest = Rest.Rest;
    };
  }


//***********************************
//  Get the Object at top of Stack. *
//***********************************
public Object Peek()
  {
  if (CellCount!=0)
    return Top;
  else
    return null;
  }
```



```
public String toString()
  {
  String res = new String();
  Stack temp = new Stack();
  temp.Top = Top;
  temp.Rest = Rest;
  temp.CellCount=CellCount;
  while (!temp.IsEmptyStack())
    {
    res = res+(temp.Peek()).toString()+"\n";
    temp.Pop();
    };
  return res;
  }
```


Stacks and Reverse Polish Expressions

Definition: An Integer value arithmetic expression, E is in *Reverse Polish Form* if:

1. E is a single integer.

OR

2. E has the form,

$$E_1 \; E_2 \; \textbf{opn}$$

where E_1, E_2 are integer value expressions in Reverse Polish and **opn** is one of **+, −, *, /**.

If E is an integer arithmetic expression in Reverse Polish Form, its *arithmetic value*, $V(E)$ is,

1. The constant in E for case (1),
2. The value of,

$$V(E_1) \; \textbf{opn} \; V(E_2)$$

in case (2).

For example,

Example	Expr.	(1) or (2)	Value
1	25	(1)	25
2	0 5 -	(2)	-5 = 0-5
3	25 0 5 - /	(2)	-5 = V(Ex1)/V(Ex2)
4	25 0 5 - / 10 +	(2)	5 = V(Ex3) + 10

Requirements

Given a sequence, S, of non-negative integer values and arithmetic operations, ending with the character @, output the integer value $V(S)$ if S is a correctly formed expression in RPN; if S is **not** correctly formed output an error message.

It may be assumed that all constant values occurring in S are non-negative integers (i.e. no floating point values).

Analysis

RPN Evaluation Algorithm

Consider the recursive definition of $V(E)$. This indicates that the value of

$$E_1 \; E_2 \; \textbf{opn}$$

could be found be finding the operation at the end, and then applying this to the value of the two preceding expressions.

Question: how do we find out where E_1 ends and E_2 starts in S?

Answer: We don't have to.

All we have to notice is that any well-formed RPN (if it's not a constant value) **must** end with an operator symbol.

Consider the following 'informal' algorithm for processing S:

Let

$$S = w_1 \, w_2 \, w_3 \cdots w_k \cdots w_m$$

where each w_ii is either a constant or an operator:

1. $i = 1$
2. if w_i is a constant
   'save' its value 'somewhere'
3. if w_i is an operator
   'replace' the 'last' value saved with
   the result of applying the operator to the
   'last' **2** values saved.
4. $i = i + 1$;
5. Go to (2.) if $i \leq m$

This performs a calculation **every time an operator** is found in the input data.

The operation is carried out on the last 2 saved constants.

The ordering regime is such that the last constant 'saved' is the first to be used when an operator is found.

All of which indicates realising steps (2) and (3) using a **stack** to 'save' values.

Implementation

Fields used *main()*:

```
Stack RPN_Expr;  // The expression
Integer Operand= new Integer(0); // Used when pushing
                                 // values on Stack.
int ValueLeft,ValueRight; // Used when evaluating expres
                          // Left Right opn, so that
                          // ValueLeft == V(Left);
                          // ValueRight == V(Right);
int FinalValue=0;         // For the expression result.
```

The natural control mechanism is a **while** loop, during the running of which input data is read and processed, i.e.

```
while ( termination symbol not read &&
        no errors found in expression )
  {
  Continue processing input data
  };
```

There are many ways in which a badly-formed Reverse Polish Expression could arise: a **boolean** ‘flag’ is set when an error is found.

boolean error_flag **=false**

A single character (**’@’**) is used for the end of an expression, so one character at a time is read from the input stream.

char next_ch;

```
while ( !( (next_ch=='@')||(error_flag) ))
  {
  Continue processing input data
  };
```

Processing Input Data

Input characters fall into 5 distinct groups:

1. Digits: {’0’,’1’,’2’,’3’,’4’,’5’,’6’,’7’,’8’,’9’}
2. Operators, {’+’,’-’,’*’,’/’}
3. "white space", e.g. spaces, tabs, line breaks,
4. The ‘end of expression’ symbol, ’@’.
5. Anything else.

Characters in (5) indicate an illegal expression.
Characters in (4) result in the loop finishing.

Characters in (3) are ignored.

This leaves the first two groups.

The framework we now have is:

```
next_ch = IgnoreWhiteSpace(keyboardInput);
while (!( (next_ch=='@')||(error_flag) ))
   {
   if Character.IsDigit(next_ch)
     Process digit case
   else if OperatorSymbol(next_ch)
     Process Operator case
   else
     error_flag = true
   }
   ...
```

Process Digit Case

A digit indicates that item being read is a constant value.

1. Read the remaining digits.
2. ‘Wrap’ the value read as an Integer.
3. Push this Integer onto the **RPN_Expr Stack**

Process Operator Symbol Case

Using the algorithm earlier, if **next_ch** is an operator symbol,

1. Get the integer at the top of the stack: $V(\,E_2\,)$.
2. Remove (**Pop()**) this value from the stack.
3. Get the integer now at the top of the stack: $V(\,E_1\,)$.
4. Remove (**Pop()**) this value from the stack.
5. Calculate the result of

$$V(\,E_1\,)\ \mathbf{opn}\ V(\,E_2\,)$$

6. **Push** this value onto the Stack.

If there are fewer than two items on the **RPN_Expr** Stack then the expression was not properly formed and the **error_flag** can be set.


```
//*******************************************************
// If the character is an operator then the top 2 items    *
// currently on the Stack, (V1, V2 say) can be replaced by *
// the integer value V2 <op> V1.                           *
//*******************************************************
else if (OperatorSymbol(next_ch))
  {
  // Must be (>=) 2 values on the stack at this point.
  if (RPN_Expr.IsEmptyStack())
    error_flag = true;
  else
    {
    // Find out the value of the ‘right-hand’ operand.
    ValueRight = (Integer.valueOf(
                   RPN_Expr.Peek().toString()
                                 )
                 ).intValue();
    RPN_Expr.Pop();      // Remove it from the Stack.
    // Find out the value of the ‘left-hand’ operand.
    if (RPN_Expr.IsEmptyStack())
      error_flag=true;
    else
      {
      ValueLeft = (Integer.valueOf(
                    RPN_Expr.Peek().toString()
                                  )
                  ).intValue();
      RPN_Expr.Pop();      // Remove it from the Stack,
      switch (next_ch)     // Compute replacement
        {
        case '+': Operand = new Integer(ValueLeft+ValueRight);
                  break;
        case '-': Operand = new Integer(ValueLeft-ValueRight);
                  break;
        case '*': Operand = new Integer(ValueLeft*ValueRight);
                  break;
        case '/': Operand = new Integer(ValueLeft/ValueRight);
        };
      RPN_Expr.Push(Operand);   // and save on the stack.
      next_ch = IgnoreWhiteSpace(keyboardInput);
      };
```


The final stage is to output the value remaining on the **RPN_Expr** Stack when the loop is exited, checking that exactly one value remains on this.

Summary

1. Many Computer Science applications use models in which information can only be held and retrieved according to a precisely defined protocol. Among the most commonly used such regimes are the ADTs known as **Stacks** and **Queues**.
2. Dynamic data structures are well-suited to realising both.
3. The Stack ADT considered in this lecture, is widely used in a number of systems programming tools, such as high-level language compilers.
4. The application involving using a stack to evaluate an arithmetic expression in Reverse Polish Notation, is one such example. It is a simple matter to develop the method described to cater for ‘code-generation’.