**Data Structures: Stacks and Queues**

**Stacks:**

A stack is a linear data structure that is used whenever LIFO (Last In First Out) type of data processing is required. A stack can be considered as a box with one open end. Everything that is to be put into the box (***insertion***) is put from this end and whatever is to be taken out of the box (***deletion***) is taken out from this open end only. This way whatever is put first into the box is taken out at the end, and whatever is put at the end is taken out first and hence LIFO processing. The open end of a stack is called its TOP, the process of insertion in a stack is called PUSH and the process of deletion from a stack is called POP.

**C**

**D**

**TOP**

**B**

**A**

|  |
| --- |
| Operation |
| **A**  **PUSH**  **TOP**  **A** |

|  |
| --- |
| Operation  **B**  **PUSH**  **A**  **B**  **TOP** |
|  |

**A**

**B**

**TOP**

**C**

**C**

#### PUSH

|  |
| --- |
| Operation |
|  |

|  |
| --- |
| Operation  **POP**  **A**  **B**  **TOP**  **C** |
|  |

|  |
| --- |
| Operation |
| **POP** |

**A**

**B**

**TOP**

**C**

|  |
| --- |
| Operation |
| **A**  **B**  **C**  **TOP**  **POP** |

**Implementing a stack**:

A stack can be implemented using an array or a linked list.

Array Implementation of a stack: For this we take an array to be used as a stack and an integer variable to keep track of the TOP of the stack. Initially the value of TOP is –1 indicating that the stack is empty. For each PUSH TOP is incremented by 1, and for each POP TOP is decremented by 1. Overflow and Underflow conditions have to be checked at each PUSH and POP respectively.

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | |  | 0 | 1 | 2 | 3 |  | | S |  |  |  |  | TOP=-1(Empty Stack) | |
| Now POP() gives the following stack |
| |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | |  | 0 | 1 | 2 | 3 |  | | S |  |  |  |  | TOP=-1(Empty Stack) |   Here we get an Underflow Error because we are trying to remove an element from an empty stack |
| PUSH(5) gives the following stack |
| |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | |  | 0 | 1 | 2 | 3 |  | | S | 5 |  |  |  | TOP=0 | |
| PUSH(3) gives the following stack |
| |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | |  | 0 | 1 | 2 | 3 |  | | S | 5 | 3 |  |  | TOP=1 | |
| PUSH(4) gives the following stack |
| |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | |  | 0 | 1 | 2 | 3 |  | | S | 5 | 3 | 4 |  | TOP=2 | |
| PUSH(9) gives the following stack |
| |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | |  | 0 | 1 | 2 | 3 |  | | S | 5 | 3 | 4 | 9 | TOP=3 | |
| PUSH(6) gives the following stack |
| |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | |  | 0 | 1 | 2 | 3 |  | | S | 5 | 3 | 4 | 9 | TOP=9 |     Here we get an Overflow Error because the stack is already full and no more data elements can be inserted in it. |
| POP() gives the following stack |
| |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | |  | 0 | 1 | 2 | 3 |  | | S | 5 | 3 | 4 |  | TOP=2 |     Element POPped is 9 |

Linked Implementation of a stack: For this each data element is represented by a Node. A pointer variable TOP is taken to store the address of the last node PUSHed (or the first node to be POPped). PUSH is same as inserting a node at the head of the linked list and POP is same as deleting the first node of the list.

|  |
| --- |
| TOP  (Empty Stack. TOP has a NULL value) |
| Now POP() gives the following stack |
| TOP  Here we get an Underflow Error because we are trying to remove an element from an empty stack. |
| PUSH(5) gives the following stack |
| TOP  5 |
| PUSH(3) gives the following stack |
| TOP  3  5 |
| PUSH(4) gives the following stack |
| TOP  5  4  3 |
| POP() gives the following stack |
| TOP  5  3  Element 4 is POPped |

**Applications of a stack**:

A stack is used in whatever application the data is to be processed in LIFO order. A few such applications are:

a) Conversion from Infix to Postfix

b) Evaluation of a postfix expression

c) Execution of Function calls

d) Execution of Recursion

e) Matching parentheses

Conversion from Infix to postfix: The mathematical or Boolean expressions that we use in daily mathematics are written in Infix notation. For example an expression A+B is an Infix expression. This notation is called infix notation because in this notation an operator (+ in this example) is written in-between its operands (A, B in this example). Such an expression is very convenient (really?) for us to evaluate but for the computer evaluation of an infix expression may be a cumbersome and time-consuming process. To save the evaluation time the compiler converts every infix expression that appears in a program into its corresponding postfix expression. A postfix expression can be executed faster than the corresponding infix expression. In a postfix expression an operator appears after its operands. For example the postfix expression for A+B is A B +. Given below are a few more infix and corresponding postfix expressions:

|  |  |  |
| --- | --- | --- |
|  | Infix | Postfix |
|  | A+B\*C | A B C \* + |
|  | (A+B)\*C | A B + C \* |
|  | (A+B)/C\*D+E^F | A B + C / D \* E F ^ + |
|  | A+B\*C/D\*E | A B C \* D / E \* + |

A postfix expression does not have any parentheses and while evaluating it priority of operators does not have to be taken care of because it is already taken care of while converting the expression from infix to postfix. Priority of operators is :

|  |  |
| --- | --- |
| **Operators** | **Priority** |
| unary operators (+, -, NOT) | 4 |
| Exponent (^) | 3 |
| \*, /, AND | 2 |
| +, - (Binary), OR | 1 |

**Examples of Conversion from Infix to Postfix:**

(1) Convert A+(B-C)\*D/E into corresponding postfix form

First we put a right parentheses at the end of the expression and Push the corresponding left parentheses in the Operator Stack. Then we proceed as follows:

|  |  |  |  |
| --- | --- | --- | --- |
| **Symbol**  **Scanned** | **Operator**  **Stack** | **Output** | Remarks / Explanation |
|  | ( |  | Initially we push a **)** in the Operator Stack |
| A | ( | A | The operand **A** is taken to the output |
| + | (+ | A | The operator **+** is to be PUSHed on the stack. As the topmost element of the stack is (, + is pushed w/o POPping out anything from the stack. |
| ( | (+( | A | **(** is PUSHed on the stack. |
| B | (+( | AB | The operand **B** is taken to the output. |
| - | (+(- | AB | Operator **–** is PUSHed on the stack w/o POPping out anything. |
| C | (+(- | ABC | The operand **C** is taken to the output. |
| ) | (+ | ABC- | When a **)** is encountered, all the operators (only – in this example) till the first ( are POPped from the stack and are taken to the output. ( is also POPped but not taken to the output. |
| \* | (+\* | ABC- | The operator **\*** is to be PUSHed in the stack. As the priority of \* is more than that of + (topmost element of the stack), therefore \* can be PUSHed above +. |
| D | (+\* | ABC-D | The operand **D** is taken to the output. |
| / | (+/ | ABC-D\* | The operator **/** is to be PUSHed in the stack. As the priority of / is not more than that of \* (topmost element of the stack), therefore / cannot be PUSHed above \*. So first we POP \* and take it to the output. Now + is the topmost element of the stack. As the priority of / is more than +, therefore / is PUSHed above +. |
| E | (+/ | ABC-D\*E | The operand **E** is taken to the output. |
| ) |  | ABC-D\*E/+ | When a **)** is encountered, all the operators (/ and + in this example) till the first ( are POPped from the stack and are taken to the output. ( is also POPped but not taken to the output. |

Now the stack is empty, therefore the expression has been converted. The corresponding postfix expression is ABC-D\*E/+. To distinguish between different operators and operands , may be inserted between two consecutive symbols and the postfix expression may be written as: A,B,C,-,D,\*,E,/,+

(2) Convert **(A+B)-(C-D\*E)^F** into corresponding postfix form. Here ^ represents exponenent or ‘raised to the power of’ operator.

First we put a right parentheses at the end of the expression and Push the corresponding left parentheses in the Operator Stack. Then we proceed as follows:

|  |  |  |  |
| --- | --- | --- | --- |
| **Symbol**  **Scanned** | **Operator**  **Stack** | **Output** | Remarks / Explanation |
|  | ( |  | Initially we push a **)** in the Operator Stack |
| ( | (( |  | **(** is PUSHed on the stack. |
| A | (( | A | The operand **A** is taken to the output |
| + | ((+ | A | The operator **+** is to be PUSHed on the stack. As the topmost element of the stack is (, + is pushed w/o POPping out anything from the stack. |
| B | ((+ | AB | The operand **B** is taken to the output. |
| ) | ( | AB+ | When a **)** is encountered, all the operators (only + in this example) till the first **(** are POPped from the stack and are taken to the output. **(** is also POPped but not taken to the output. |
| - | (- | AB+ | Operator **–** is PUSHed on the stack w/o POPping out anything. |
| ( | (-( | AB+ | **(** is PUSHed on the stack |
| C | (-( | AB+C | The operand **C** is taken to the output. |
| - | (-(- | AB+C | The operator **-** is to be PUSHed in the stack. As the topmost element of the stack is (, - can be PUSHed in the stack w/o POPping anything. |
| D | (-(- | AB+CD | The operand **D** is taken to the output. |
| \* | (-(-\* | AB+CD | The operator **\*** is to be PUSHed in the stack. As the priority of \* is more than that of - (topmost element of the stack), therefore \* can be PUSHed above -. |
| E | (-(-\* | AB+CDE | The operand **E** is taken to the output. |
| ) | (- | AB+CDE\*- | When a **)** is encountered, all the operators (\* and - in this example) till the first **(** are POPped from the stack and are taken to the output. **(** is also POPped but not taken to the output. |
| ^ | (-^ | AB+CDE\*- | The operator **^** is to be PUSHed in the stack. As the priority of ^ is more than that of - (topmost element of the stack), therefore ^ can be PUSHed above -. |
| F | (-^ | AB+CDE\*-F | The operand **F** is taken to the output. |
| ) |  | AB+CDE\*-F^- | When a **)** is encountered, all the operators (\* and - in this example) till the first **(** are POPped from the stack and are taken to the output. **(** is also POPped but not taken to the output. |

Now the stack is empty, therefore the expression has been converted. The corresponding postfix expression is A,B,+,C,D,E,\*,-,F,^,-.

(3) Convert **A AND B OR NOT C AND D** into corresponding postfix form.

First we put a right parentheses at the end of the expression and Push the corresponding left parentheses in the Operator Stack. Then we proceed as follows:

|  |  |  |  |
| --- | --- | --- | --- |
| **Symbol**  **Scanned** | **Operator**  **Stack** | **Output** | Remarks / Explanation |
|  | ( |  | Initially we push a **)** in the Operator Stack |
| A | ( | A | The operand **A** is taken to the output |
| AND | (AND | A | The operator **AND** is to be PUSHed on the stack. As the topmost element of the stack is a (, **AND** is pushed w/o POPping out anything from the stack. |
| B | (AND | A B | The operand **B** is taken to the output |
| OR | (OR | A B AND | The operator **OR** is to be PUSHed in the stack. As the priority of **OR** is not more than that of AND (topmost element of the stack), therefore OR cannot be PUSHed above AND. So first we POP AND and take it to the output. Now ( is the topmost element of the stack and therefore OR is PUSHed on the stack. |
| NOT | (OR NOT | A B AND | The operator **NOT** is to be PUSHed in the stack. As the priority of NOT is more than that of OR (topmost element of the stack), therefore NOT can be PUSHed above OR. |
| C | (OR NOT | A B AND C | The operand **C** is taken to the output |
| AND | (OR AND | A B AND C NOT | The operator **AND** is to be PUSHed in the stack. As the priority of **AND** is not more than that of NOT (topmost element of the stack), therefore AND cannot be PUSHed above NOT. So first we POP NOT and take it to the output. Now OR is the topmost element of the stack. As the priority of AND is more than that of OR, AND is PUSHed above OR. |
| D | (OR AND | A B AND C NOT D | The operand **D** is taken to the output |
| ) |  | A B AND C NOT D AND OR | When a **)** is encountered, all the operators (AND and OR in this example) till the first **(** are POPped from the stack and are taken to the output. **(** is also POPped but not taken to the output. |

Now the stack is empty, therefore the expression has been converted. The corresponding postfix expression is A,B,AND,C,NOT,D,AND,OR.

**Evaluation of a Postfix Expression**:

If we evaluate a postfix expression, the result will be the corresponding infix expression (if the variables are given in the postfix expression) or a value (if the values are given in the postfix expression).

**Examples of Evaluation of Postfix Expressions:**

(1) Evaluate: A,B,C,-,D,\*,E,/,+

|  |  |  |
| --- | --- | --- |
| **Symbol**  **Scanned** | **Stack** | Remarks / Explanation |
| A | A | Operand A is PUSHed in the Stack |
| B | A,B | Operand B is PUSHed in the Stack |
| C | A,B,C | Operand C is PUSHed in the Stack |
| - | A,**(B-C)** | When an operator is encountered, two elements are POPped from the stack, the relevant operator is performed and the result is PUSHed on the stack. |
| D | A,**(B-C)**,D | Operand D is PUSHed in the Stack |
| \* | A,**((B-C)\*D)** | When an operator is encountered, two elements are POPped from the stack, the relevant operator is performed and the result is PUSHed on the stack. |
| E | A,**((B-C)\*D)**,E | Operand A is PUSHed in the Stack |
| / | A,**(((B-C)\*D)/E)** | When an operator is encountered, two elements are POPped from the stack, the relevant operator is performed and the result is PUSHed on the stack. |
| + | **(A+(((B-C)\*D)/E))** | When an operator is encountered, two elements are POPped from the stack, the relevant operator is performed and the result is PUSHed on the stack. |
| ; |  | When **;** is encountered, it indicates the end of the expression and the result (the only element present in the stack) is POPped from the stack. |

Therefore the corresponding infix expression is: **(A+(((B-C)\*D)/E))**

(2) Evaluate: **True,False,AND,False,NOT,True,AND,OR**.

|  |  |  |
| --- | --- | --- |
| **Symbol**  **Scanned** | **Stack** | Remarks / Explanation |
| True | True | Operand **True** is PUSHed in the Stack |
| False | True, False | Operand **False** is PUSHed in the Stack |
| AND | (True AND False) i.e. False | When an operator is encountered, two elements are POPped from the stack, the relevant operator is performed and the result is PUSHed on the stack. |
| False | False, False | Operand **False** is PUSHed in the Stack |
| NOT | False, NOT False i.e. False, True | When an operator is encountered, two elements are POPped from the stack (Here **NOT** is a unary operator, therefore only one operand is POPped), the relevant operator is performed and the result is PUSHed on the stack. |
| True | False, True, True | Operand **True** is PUSHed in the Stack |
| AND | False, True AND True i.e. False, True | When an operator is encountered, two elements are POPped from the stack, the relevant operator is performed and the result is PUSHed on the stack. |
| OR | False OR True i.e. True | When an operator is encountered, two elements are POPped from the stack, the relevant operator is performed and the result is PUSHed on the stack. |
| ; |  | When **;** is encountered, it indicates the end of the expression and the result (the only element present in the stack) is POPped from the stack. |

Therefore the value of the given postfix expression is **True**.

**Queues:**

A queue is a linear data structure that is used whenever FIFO (First In First Out) type of data processing is required. A queue can be considered as a pipe with two open ends. One end is used for insertions and the other for deletions from the queue. This way whatever is put first into the pipe is taken out first, and whatever is put at the end is taken out at the end only and hence FIFO processing. The open end used for insertions is called the **Rear** of the queue and the end used for deletions is called the **Front** of the queue.

**Implementing a queue**:

A queue can be implemented using an array or a linked list.

Array Implementation of a queue: For this we take an array to be used as a queue and two integer variables to keep track of the Front and the Rear of the queue. Initially the value of Front as well as Rear is -1 indicating that the queue is empty. To understand how Front and Rear change with insertions and deletions let us try to understand the following example.

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  |  |  |  | | --- | --- | --- | --- | --- | |  | 0 | 1 | 2 | 3 | | Q |  |  |  |  |   F = R = -1  (Initially the queue is empty) |
| The operation Delete() results in: |
| |  |  |  |  |  | | --- | --- | --- | --- | --- | |  | 0 | 1 | 2 | 3 | | Q |  |  |  |  |   F = R = -1  (Initially the queue is empty)  (Here we get an underflow error as we are trying to delete an element from an empty queue) |
| The operation Insert(5) results in: |
| |  |  |  |  |  | | --- | --- | --- | --- | --- | |  | 0 | 1 | 2 | 3 | | Q | 5 |  |  |  |   F = R = 0 |
| The operation Insert(6) results in |
| |  |  |  |  |  | | --- | --- | --- | --- | --- | |  | 0 | 1 | 2 | 3 | | Q | 5 | 6 |  |  |   F=0, R=1 |
| The operation Insert(7) results in |
| |  |  |  |  |  | | --- | --- | --- | --- | --- | |  | 0 | 1 | 2 | 3 | | Q | 5 | 6 | 7 |  |   F=0, R=2 |
| The operation Delete() results in |
| |  |  |  |  |  | | --- | --- | --- | --- | --- | |  | 0 | 1 | 2 | 3 | | Q |  | 6 | 7 |  |   F=1, R=2 |
|  |
|  |

|  |
| --- |
| The operation Delete() results in |
| |  |  |  |  |  | | --- | --- | --- | --- | --- | |  | 0 | 1 | 2 | 3 | | Q |  |  | 7 |  |   F=2, R=2 |

|  |
| --- |
| The operation Insert(3) results in |
| |  |  |  |  |  | | --- | --- | --- | --- | --- | |  | 0 | 1 | 2 | 3 | | Q |  |  | 7 | 3 |   F=2, R=3 |
| The operation Insert(5) results in |
| Now where should we insert 5? It cannot be inserted after 3 because there is no location vacant. So is it an “overflow” condition because no more data elements can be inserted in the queue? If it is so then there may be a case where we define a large queue (to store 100’s of data elements) and after many insertions and deletions  there is only one element at the end of the queue but we cannot insert any more data elements although all the locations but one of the queue are vacant. We do not want such a situation. Therefore when we have to opt for array implementation of a queue, we make the queue a circular one. In a circular queue if the last element is at the last location and some initial locations are vacant (F != 0) then the new element is inserted at the first location. This way the above operation results in: |

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  |  |  |  | | --- | --- | --- | --- | --- | |  | 0 | 1 | 2 | 3 | | Q | 5 |  | 7 | 3 |   F=2, R=0 |
| The operation Insert(8) results in |
| |  |  |  |  |  | | --- | --- | --- | --- | --- | |  | 0 | 1 | 2 | 3 | | Q | 5 | 8 | 7 | 3 |   F=2, R=1 |
| The operation Insert(9) results in |
| |  |  |  |  |  | | --- | --- | --- | --- | --- | |  | 0 | 1 | 2 | 3 | | Q | 5 | 8 | 7 | 3 |   F=2, R=1  (Here we get an overflow error because now the queue is full and we are trying to insert an element from the queue.) |
| The operation Delete() results in |
| |  |  |  |  |  | | --- | --- | --- | --- | --- | |  | 0 | 1 | 2 | 3 | | Q | 5 | 8 |  | 3 |   F=3, R=1  (7 is deleted) |
| The operation Delete() results in |
| |  |  |  |  |  | | --- | --- | --- | --- | --- | |  | 0 | 1 | 2 | 3 | | Q | 5 | 8 |  |  |   F=0, R=1  (3 is deleted) |
| The operation Delete() results in |
| |  |  |  |  |  | | --- | --- | --- | --- | --- | |  | 0 | 1 | 2 | 3 | | Q |  | 8 |  |  |   F=1, R=1  (5 is deleted) |
|  |
|  |

|  |
| --- |
| The operation Delete() results in |
| |  |  |  |  |  | | --- | --- | --- | --- | --- | |  | 0 | 1 | 2 | 3 | | Q |  |  |  |  |   F=-1, R=-1  (8 is deleted)  Now the queue is again empty. |

Linked Implementation of a queue: For this each data element is represented by a Node. Two pointer variable Front and Rear are are taken to store the address of the first and the last node respectively. INSERT\_Q() is same as inserting a node at the end of the linked list and Delete\_Q() is same as deleting the first node of the list.

Applications of a queue:

a) Traversal of a graph

b) Simulations

**C++ Implementation of stacks and queues**

**\*\* As done in the class \*\***