

MODULE 3: STACKS and QUEUES

3.1 STACKS

Stack is a linear data structure in which the insertion and deletion operations are performed at only one end. In a stack, adding and removing of elements are performed at single position which is known as "top". That means, new element is added at top of the stack and an element is removed from the top of the stack. In stack, the insertion and deletion operations are performed based on LIFO (Last In First Out) principle.

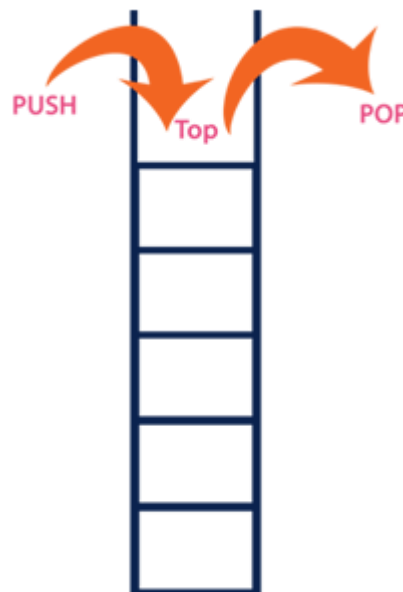


Fig 3.1: Stack

In a stack, the insertion operation is performed using a function called "push" and deletion operation is performed using a function called "pop".

In the figure 3.1, PUSH and POP operations are performed at top position in the stack. That means, both the insertion and deletion operations are performed at one end (i.e., at Top)

A stack data structure can be defined as “a linear data structure in which the operations are performed based on LIFO principle”

(or)

"A Collection of similar data items in which both insertion and deletion operations are performed based on LIFO principle".

If we want to create a stack by inserting 10,45,12,16,35 and 50. Then 10 becomes the bottom most element and 50 is the top most element. Top is at 50 as shown in the fig 3.2.



Fig 3.2: Top of a Stack

Stack data structure can be implement in two ways. They are as follows:

1. Using Array
2. Using Linked List

When stack is implemented using array, that stack can organize only limited number of elements. When stack is implemented using linked list, that stack can organize unlimited number of elements.

3.1.1 Operations on Stacks:

The following operations are performed on the stack:

1. Push (To insert an element on to the stack)
2. Pop (To delete an element from the stack)
3. Display (To display elements of the stack)

3.1.1.1 Operations on Stacks Using Arrays:

A stack data structure can be implemented using one dimensional array. But stack implemented using array, can store only fixed number of data values. This implementation is very simple, just define a one-dimensional array of specific size and insert or delete the values into that array by using LIFO principle with the help of a variable 'top'. Initially top is set to -1. Whenever we want to insert a value into the stack, increment the top value by one and then insert. Whenever we want to delete a value from the stack, then delete the top value and decrement the top value by one.

A stack can be implemented using array by following the below steps to create an empty stack.

- Step 1: Include all the header files which are used in the program and define a constant 'SIZE' with specific value.
- Step 2: Declare all the functions used in stack implementation.
- Step 3: Create a one-dimensional array with fixed size (int stack[SIZE])
- Step 4: Define an integer variable 'top' and initialize with '-1'. (int top = -1)
-

```
1. #include <stdio.h>
2. #include <conio.h>
3. #define SIZE 10
4. int stack[SIZE], top = -1;
```

push(value) - Inserting value into the stack

In a stack, push() is a function used to insert an element into the stack. In a stack, the new element is always inserted at top position. Push function takes one integer value as parameter and inserts that value into the stack. We can use the following steps to push an element on to the stack:

- Step 1: Check whether stack is FULL. (top == SIZE-1)
- Step 2: If it is FULL, then display "Stack is FULL!!! Insertion is not possible!!!" and terminate the function.
- Step 3: If it is NOT FULL, then increment top value by one (top++) and set stack[top] to value (stack[top] = value).

```
1. void push(int value) {
2.     if (top == SIZE - 1) printf("\nStack is Full!!! Insertion is not possible!!!");
3.     else {
4.         top++;
5.         stack[top] = value;
6.         printf("\nInsertion success!!!");
7.     }
8. }
```

pop() - Delete a value from the Stack

In a stack, pop() is a function used to delete an element from the stack. In a stack, the element is always deleted from top position. Pop function does not take any value as parameter. We can use the following steps to pop an element from the stack:

- Step 1: Check whether stack is EMPTY. (top == -1)
- Step 2: If it is EMPTY, then display "Stack is EMPTY!!! Deletion is not possible!!!" and terminate the function.
- Step 3: If it is NOT EMPTY, then delete stack[top] and decrement top value by one (top--).

```
1. void pop() {
2.     if (top == -1) printf("\nStack is Empty!!! Deletion is not possible!!!");
3.     else {
4.         printf("\nDeleted : %d", stack[top]);
5.         top--;
6.     }
7. }
```

display() - Displays the elements of a Stack

We can use the following steps to display the elements of a stack...

- Step 1: Check whether stack is EMPTY. (top == -1)
- Step 2: If it is EMPTY, then display "Stack is EMPTY!!!" and terminate the function.

- Step 3: If it is NOT EMPTY, then define a variable 'i' and initialize with top. Display stack[i] value and decrement i value by one (i--).
- Step 4: Repeat above step until i value becomes '0'.

```
1. void display() {
2.     if (top == -1) printf("\nStack is Empty!!!");
3.     else {
4.         int i;
5.         printf("\nStack elements are:\n");
6.         for (i = top; i >= 0; i--) printf("%d\n", stack[i]);
7.     }
8. }
```

Stack Implementation Using ARRAY & it's operations

```
1. #include <stdio.h>
2. #include <conio.h>
3. #define SIZE 10
4. void push(int);
5. void pop();
6. void display();
7. int stack[SIZE], top = -1;
8. void main() {
9.     int value, choice;
10.    clrscr();
11.    while (1) {
12.        printf("\n***** MENU *****\n");
13.        printf("1. Push\n2. Pop\n3. Display\n4. Exit");
14.        printf("\nEnter your choice: ");
15.        scanf("%d", & choice);
16.        switch (choice) {
17.            case 1:
18.                printf("Enter the value to be insert: ");
19.                scanf("%d", & value);
20.                push(value);
21.                break;
22.            case 2:
23.                pop();
24.                break;
25.            case 3:
26.                display();
27.                break;
28.            case 4:
29.                exit(0);
30.            default:
31.                printf("\nWrong selection!!! Try again!!!");
32.        }
33.    }
34. }
35. void push(int value) {
36.     if (top == SIZE - 1) printf("\nStack is Full!!! Insertion is not possible!!!");
```

```

37. else {
38.     top++;
39.     stack[top] = value;
40.     printf("\nInsertion success!!!");
41. }
42. }
43. void pop() {
44.     if (top == -1) printf("\nStack is Empty!!! Deletion is not possible!!!");
45.     else {
46.         printf("\nDeleted : %d", stack[top]);
47.         top--;
48.     }
49. }
50. void display() {
51.     if (top == -1) printf("\nStack is Empty!!!");
52.     else {
53.         int i;
54.         printf("\nStack elements are:\n");
55.         for (i = top; i >= 0; i--) printf("%d\n", stack[i]);
56.     }
57. }

```

3.1.1.2 Operations on Stacks Using Linked Lists

The major problem with the stack implemented using array is, it works only for fixed number of data values. That means the amount of data must be specified at the beginning of the implementation itself. Stack implemented using array is not suitable when we don't know the size of data which we are going to use. A stack data structure can be implemented by using linked list data structure. The stack implemented using linked list can work for unlimited number of values. That means, stack implemented using linked list works for variable size of data.

So, there is no need to fix the size at the beginning of the implementation. The Stack implemented using linked list can organize as many data values as we want.

In linked list implementation of a stack, every new element is inserted as 'top' element. That means every newly inserted element is pointed by 'top'. Whenever we want to remove an element from the stack, simply remove the node which is pointed by 'top' by moving 'top' to its next node in the list. The next field of the first element must be always NULL.

To implement stack using linked list, we need to follow the below given steps.

- Step 1: Include all the header files which are used in the program. And declare all the user defined functions.
- Step 2: Define a 'Node' structure with two members data and next.
- Step 3: Define a Node pointer 'top' and set it to NULL.

```

1. #include <stdio.h>
2. #include <conio.h>
3. struct Node {
4.     int data;
5.     struct Node * next;
6. } * top = NULL;

```

push(value) - Inserting an element into the Stack

We can use the following steps to insert a new node into the stack...

- Step 1: Create a newNode with given value.
- Step 2: Check whether stack is Empty (top == NULL)
- Step 3: If it is Empty, then set newNode → next = NULL.
- Step 4: If it is Not Empty, then set newNode → next = top.
- Step 5: Finally, set top = newNode.

```

1. void push(int value) {
2.     struct Node * newNode;
3.     newNode = (struct Node * ) malloc(sizeof(struct Node));
4.     newNode -> data = value;
5.     if (top == NULL) newNode -> next = NULL;
6.     else newNode -> next = top;
7.     top = newNode;
8. }

```

pop() - Deleting an Element from a Stack

We can use the following steps to delete a node from the stack...

- Step 1: Check whether stack is Empty (top == NULL).
- Step 2: If it is Empty, then display "Stack is Empty!!! Deletion is not possible!!!" and terminate the function
- Step 3: If it is Not Empty, then define a Node pointer 'temp' and set it to 'top'.
- Step 4: Then set 'top = top → next'.
- Step 7: Finally, delete 'temp' (free(temp)).

```

1. void pop() {
2.     if (top == NULL) printf("\nStack is Empty!!!\n");
3.     else {
4.         struct Node * temp = top;
5.         printf("\nDeleted element: %d", temp -> data);
6.         top = temp -> next;
7.         free(temp);
8.     }
9. }

```

display() - Displaying stack of elements

We can use the following steps to display the elements (nodes) of a stack...

- Step 1: Check whether stack is Empty (top == NULL).
- Step 2: If it is Empty, then display 'Stack is Empty!!!' and terminate the function.
- Step 3: If it is Not Empty, then define a Node pointer 'temp' and initialize with top.
- Step 4: Display 'temp → data --->' and move it to the next node. Repeat the same until temp reaches to the first node in the stack (temp → next != NULL).
- Step 5: Finally! Display 'temp → data ---> NULL'.

```

1. void display() {
2.     if (top == NULL) printf("\nStack is Empty!!!\n");
3.     else {
4.         struct Node * temp = top;

```

```

5.     while (temp -> next != NULL) {
6.         printf("%d--->", temp -> data);
7.         temp = temp -> next;
8.     }
9.     printf("%d--->NULL", temp -> data);
10. }
11. }

```

Stack Implementation Using LINKED LIST & it's operations

```

1. #include <stdio.h>
2. #include <conio.h>
3. struct Node {
4.     int data;
5.     struct Node * next;
6. } * top = NULL;
7. void push(int);
8. void pop();
9. void display();
10. void main() {
11.     int choice, value;
12.     clrscr();
13.     printf("\n:: Stack using Linked List ::\n");
14.     while (1) {
15.         printf("\n***** MENU *****\n");
16.         printf("1. Push\n2. Pop\n3. Display\n4. Exit\n");
17.         printf("Enter your choice: ");
18.         scanf("%d", & choice);
19.         switch (choice) {
20.             case 1:
21.                 printf("Enter the value to be insert: ");
22.                 scanf("%d", & value);
23.                 push(value);
24.                 break;
25.             case 2:
26.                 pop();
27.                 break;
28.             case 3:
29.                 display();
30.                 break;
31.             case 4:
32.                 exit(0);
33.             default:
34.                 printf("\nWrong selection!!! Please try again!!!\n");
35.         }
36.     }
37. }
38. void push(int value) {
39.     struct Node * newNode;
40.     newNode = (struct Node *) malloc(sizeof(struct Node));
41.     newNode -> data = value;

```

```

42.  if (top == NULL) newNode -> next = NULL;
43.  else newNode -> next = top;
44.  top = newNode;
45.  printf("\nInsertion is Success!!!\n");
46. }
47. void pop() {
48.  if (top == NULL) printf("\nStack is Empty!!!\n");
49.  else {
50.      struct Node * temp = top;
51.      printf("\nDeleted element: %d", temp -> data);
52.      top = temp -> next;
53.      free(temp);
54.  }
55. }
56. void display() {
57.  if (top == NULL) printf("\nStack is Empty!!!\n");
58.  else {
59.      struct Node * temp = top;
60.      while (temp -> next != NULL) {
61.          printf("%d--->", temp -> data);
62.          temp = temp -> next;
63.      }
64.      printf("%d--->NULL", temp -> data);
65.  }
66. }

```

3.1.2 Applications of Stack:

1. Parsing
2. Recursive Function
3. Calling Function
4. Expression Evaluation
5. Expression Conversion
 - a. Infix to Postfix
 - b. Infix to Prefix
 - c. Postfix to Infix
 - d. Prefix to Infix
6. Towers of Hanoi

Expression Conversion: Infix, Prefix and Postfix

Infix	Prefix	Postfix
a + b	+ a b	a b +
a + b * c	+ a * b c	a b c * +
(a + b) * (c - d)	* + a b - c d	a b + c d - *
b * b - 4 * a * c	- * b b * * 4 a c	b b * 4 a * c * -
40 - 3 * 5 + 1 = 26	+ - 40 * 3 5 1	40 3 5 * - 1 +

3.1.2.5(a) Infix to Postfix Conversion using Stack**Algorithm for Infix to Postfix Conversion:**

Read the infix expression one character at a time until it encounters the delimiter. "#"

- Step 1 : If the character is an operand, place it on to the output.
- Step 2 : If the character is an operator, push it onto the stack. If the stack operator has a higher
or
equal priority than input operator then pop that operator from the stack and place it
onto the output.
- Step 3 : If the character is a left parenthesis, push it onto the stack.
- Step 4 : If the character is a right parenthesis, pop all the operators from the stack till it
 - encounters left parenthesis, discard both the parenthesis in the output.

Program:

```
#include <ctype.h>
#include <stdio.h>
#define SIZE 50          /* Size of Stack */
char s[SIZE];
int top=-1;              /* Global declarations */
push(char elem)          /* Function for PUSH operation */
{
    s[++top]=elem;
}
char pop()                /* Function for POP operation */
{
    return(s[top--]);
}
int pr(char elem)          /* Function for precedence */
{
    switch(elem)
    {
        case '#': return 0;
        case '(': return 1;
        case '+':
        case '-': return 2;
        case '*':
        case '/': return 3;
    }
}

void main()
{
    char infix[50],postfix[50],ch,elem;
    int i=0,k=0;
    printf("\n\nRead the Infix Expression ? ");
    scanf("%s",infix);
    push('#');
    while( (ch=infix[i++]) != '\0')
    {
        if( ch == '(')          /* For Open Parenthesis */
```

```

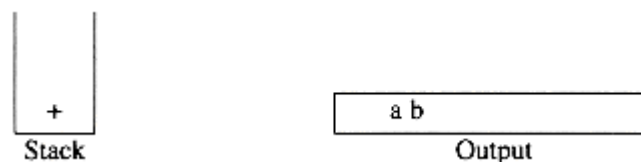
        push(ch);
    else if(isalnum(ch))                /*Alpha numeric Character*/
        postfix[k++]=ch;
    else if( ch == ')')                /* For Close Parenthesis */
    {
        while( s[top] != '(')
            postfix[k++]=pop();
        elem=pop();
    }
    else
    {
        /* Operator */
        while ( pr(s[top]) >= pr(ch) )
            postfix[k++]=pop();
        push(ch);
    }
}
while( s[top] != '#')                /* Pop from stack till empty */
{
    postfix[k++]=pop();
}
postfix[k]='\0';                    /* Make postfix as valid string */
printf("\n\nGiven Infix Expression: %s Postfix Expression: %s\n",infix,postfix);
}

```

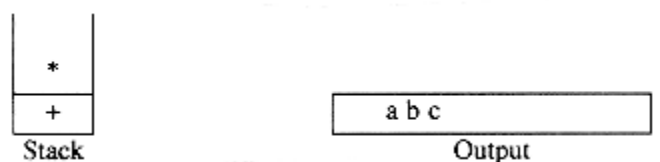
Example:

- Suppose we want to convert the infix expression $a+b*c+(d*e+f)*g$ into postfix expression.
- First, the symbol a is read, so it is passed through to the output. Then $+$ is read and pushed onto the stack.
- Next b is read and passed through to the output.

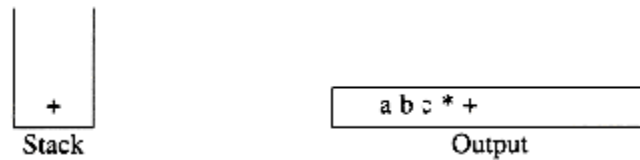
The state is as follows:



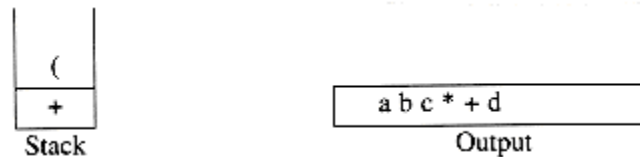
- Next a $*$ is read. The top entry on the operator stack has lower precedence than $*$, so nothing is output and $*$ is put on the stack. Next, c is read and output. Thus far, we have



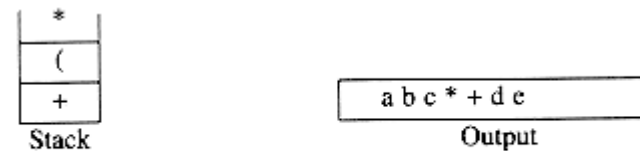
- The next symbol is a $+$. Checking the stack, we find that we will pop a $*$ and place it on the output, pop the other $+$, which is not of *lower* but equal priority, on the stack, and then push the $+$.



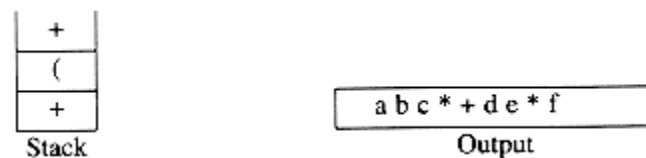
- The next symbol read is an '(', which, being of highest precedence, is placed on the stack. Then *d* is read and output.



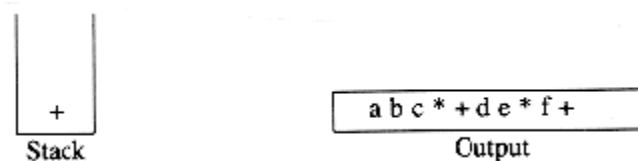
- We continue by reading a '*'. Since open parentheses do not get removed except when a closed parenthesis is being processed, there is no output. Next, *e* is read and output.



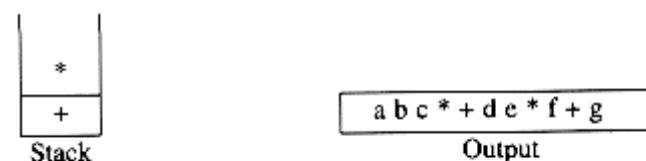
- The next symbol read is a '+'. We pop and output '*' and then we push '+'. Then we read and output



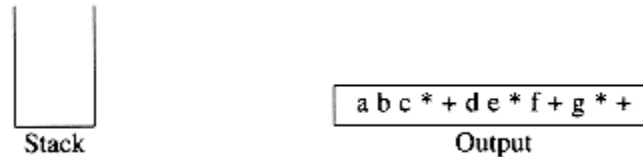
Now we read a ')', so the stack is emptied back to the '('. We output a '+'. We



We read a '*' next; it is pushed onto the stack. Then *g* is read and output.



The input is now empty, so we pop and output symbols from the stack until it is empty.



3.1.2.5(c) Postfix to Infix Conversion using Stack

Algorithm:

- Read all the symbols one by one from left to right in the given Postfix Expression
- If the reading symbol is operand, then push it on to the Stack.
- If the reading symbol is operator (+, -, *, / etc.), then perform TWO pop operations and store the two popped operands in two different variables (operand1 and operand2). Then perform reading symbol operation using operand1 and operand2 and push result back on to the Stack.
- Finally,! Perform a pop operation and display the popped value as final result.

Program:

```
#define SIZE 50                /* Size of Stack */
#include <ctype.h>
#include <stdio.h>
int s[SIZE];
int top=-1;                   /* Global declarations */

push(int elem)
{
    s[++top]=elem;
}

int pop()
{
    return(s[top--]);
}

main()
{
    /* Main Program */
    char posfx[50],ch;
    int i=0,op1,op2;
    printf("\n\nRead the Postfix Expression ? ");
    scanf("%s",posfx);
    while( (ch=posfx[i++]) != '\0')
    {
        if(isdigit(ch))
            push(ch-48);          /* Push the operand */
        else
        {
            /* Operator,pop two operands */
            op2=pop();
            op1=pop();
```

```




switch(ch)
{
    case '+':
        push(op1+op2);
        break;
    case '-':
        push(op1-op2);
        break;
    case '*':
        push(op1*op2);
        break;
    case '/':
        push(op1/op2);
        break;
}
}
}
printf("\n Given Postfix Expn: %s\n",posfx);
printf("\n Result after Evaluation: %d\n",s[top]);
}

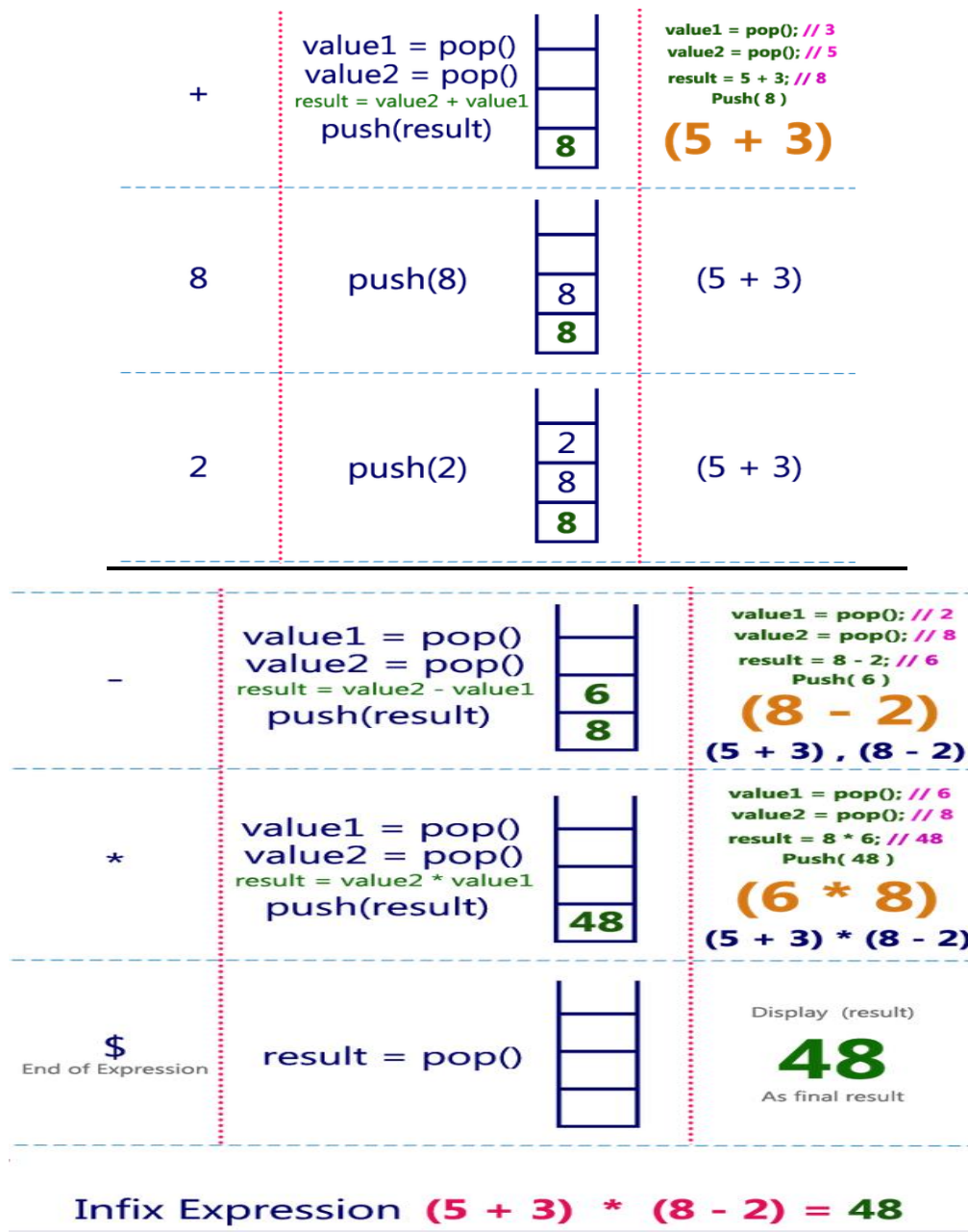
```

Infix Expression **(5 + 3) * (8 - 2)**

Postfix Expression **5 3 + 8 2 - ***

Above Postfix Expression can be evaluated by using Stack Data Structure as follows...

Reading Symbol	Stack Operations	Stack	Evaluated Part of Expression
Initially	Stack is Empty		Nothing
5	push(5)		Nothing
3	push(3)		Nothing



3.2 QUEUES

Queue is a linear data structure in which the insertion and deletion operations are performed at two different ends. In a queue data structure, adding and removing of elements are performed at two different positions. The insertion is performed at one end and deletion is performed at another end. In a queue data structure, the insertion operation is performed at a position which is known as 'rear' and the deletion operation is performed at a position which is known as 'front'. In queue data structure, the insertion and deletion operations are performed based on FIFO (First In First Out) principle.

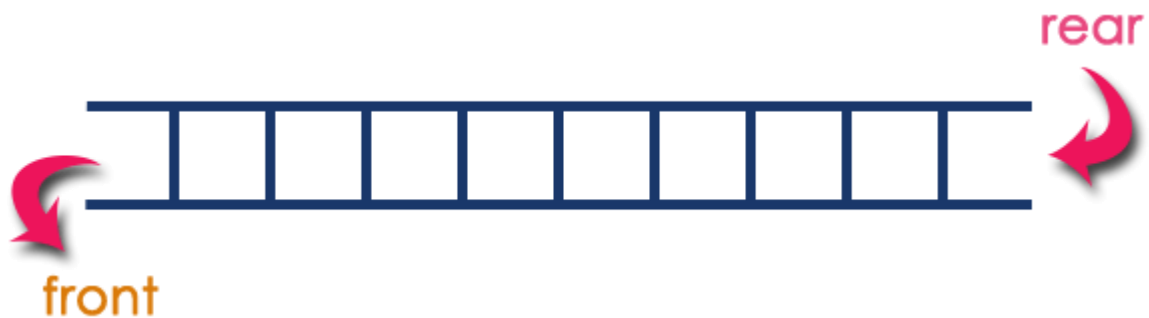


Fig 3.: Queue

In a queue data structure, the insertion operation is performed using a function called "enqueue()" and deletion operation is performed using a function called "dequeue()".

Queue data structure can be defined as “a data structure is a linear data structure in which the operations are performed based on FIFO principle.”

(OR)

“It is a collection of similar data items in which insertion and deletion operations are performed based on FIFO principle.”

Queue data structure can be implemented in two ways. They are as follows:

3. Using Array
4. Using Linked List

When a queue is implemented using array, that queue can organize only a limited number of elements. When a queue is implemented using linked list, that queue can organize an unlimited number of elements.

3.2.1 Operations on Queues:

The following operations are performed on the queues:

1. enqueue(value) - (To insert an element into the queue)
2. dequeue() - (To delete an element from the queue)
3. display() - (To display the elements of the queue)

3.2.1 Operations on Queues Using Arrays:

A queue data structure can be implemented using one dimensional array. But, queue implemented using array can store only fixed number of data values. The implementation of queue data structure using array is very simple, just define a one-dimensional array of specific size and insert or delete the values into that array by using FIFO (First In First Out) principle with the help of variables 'front' and 'rear'. Initially both 'front' and 'rear' are set to -1. Whenever, we want to insert a new value into the queue, increment 'rear' value by one and then insert at that position. Whenever we want to delete a value from the queue, then increment 'front' value by one and then display the value at 'front' position as deleted element.

A queue can be implemented using array by following the below steps to create an empty queue.

- Step 1: Include all the header files which are used in the program and define a constant 'SIZE' with specific value.
- Step 2: Declare all the user defined functions which are used in queue implementation.
- Step 3: Create a one-dimensional array with above defined SIZE (int queue[SIZE])
- Step 4: Define two integer variables 'front' and 'rear' and initialize both with '-1'. (int front = -1, rear = -1)

```
1. #include <stdio.h>
2. #include <conio.h>
3. #define SIZE 10
4. int queue[SIZE], front = -1, rear = -1;
```

enQueue(value) - Inserting value into the queue

In a queue data structure, enQueue() is a function used to insert a new element into the queue. In a queue, the new element is always inserted at rear position. The enQueue() function takes one integer value as parameter and inserts that value into the queue. We can use the following steps to insert an element into the queue...

Step 1: Check whether queue is FULL. (rear == SIZE-1)

Step 2: If it is FULL, then display "Queue is FULL!!! Insertion is not possible!!!" and terminate the function.

Step 3: If it is NOT FULL, then increment rear value by one (rear++) and set queue[rear] = value.

```
1. void enQueue(int value) {
2.     if (rear == SIZE - 1) printf("\nQueue is Full!!! Insertion is not possible!!!");
3.     else {
4.         if (front == -1) front = 0;
5.         rear++;
6.         queue[rear] = value;
7.     }
8. }
```


deQueue() - Deleting a value from the Queue

In a queue data structure, deQueue() is a function used to delete an element from the queue. In a queue, the element is always deleted from front position. The deQueue() function does not take any value as parameter. We can use the following steps to delete an element from the queue...

- Step 1: Check whether queue is EMPTY. (front == rear)
- Step 2: If it is EMPTY, then display "Queue is EMPTY!!! Deletion is not possible!!!" and terminate the function.
- Step 3: If it is NOT EMPTY, then increment the front value by one (front ++). Then display queue[front] as deleted element. Then check whether both front and rear are equal (front == rear), if it TRUE, then set both front and rear to '-1' (front = rear = -1).

```

1. void deQueue() {
2.     if (front == rear) printf("\nQueue is Empty!!! Deletion is not possible!!!");
3.     else {
4.         printf("\nDeleted : %d", queue[front]);
5.         front++;
6.         if (front == rear) front = rear = -1;
7.     }
8. }
```

display() - Displays the elements of a Queue

We can use the following steps to display the elements of a queue...

- Step 1: Check whether queue is EMPTY. (front == rear)
- Step 2: If it is EMPTY, then display "Queue is EMPTY!!!" and terminate the function.
- Step 3: If it is NOT EMPTY, then define an integer variable 'i' and set 'i = front+1'.
- Step 3: Display 'queue[i]' value and increment 'i' value by one (i++). Repeat the same until 'i' value is equal to rear (i <= rear)

```

1. void display() {
2.     if (rear == -1) printf("\nQueue is Empty!!!");
3.     else {
4.         int i;
5.         printf("\nQueue elements are:\n");
6.         for (i = front; i <= rear; i++) printf("%d\t", queue[i]);
7.     }
8. }
```

Queue Implementation Using ARRAY & it's operations

```

1. #include <stdio.h>
2. #include <conio.h>
3. #define SIZE 10
4.
5. void enQueue(int);
6. void deQueue();
7. void display();
8. int queue[SIZE], front = -1, rear = -1;
9. void main() {
```

```

10.  int value, choice;
11.  clrscr();
12.  while (1) {
13.      printf("\n***** MENU *****\n");
14.      printf("1. Insertion\n2. Deletion\n3. Display\n4. Exit");
15.      printf("\nEnter your choice: ");
16.      scanf("%d", & choice);
17.      switch (choice) {
18.          case 1:
19.              printf("Enter the value to be insert: ");
20.              scanf("%d", & value);
21.              enQueue(value);
22.              break;
23.          case 2:
24.              deQueue();
25.              break;
26.          case 3:
27.              display();
28.              break;
29.          case 4:
30.              exit(0);
31.          default:
32.              printf("\nWrong selection!!! Try again!!!");
33.      }
34.  }
35. }
36. void enQueue(int value) {
37.     if (rear == SIZE - 1) printf("\nQueue is Full!!! Insertion is not possible!!!");
38.     else {
39.         if (front == -1) front = 0;
40.         rear++;
41.         queue[rear] = value;
42.         printf("\nInsertion success!!!");
43.     }
44. }
45. void deQueue() {
46.     if (front == rear) printf("\nQueue is Empty!!! Deletion is not possible!!!");
47.     else {
48.         printf("\nDeleted : %d", queue[front]);
49.         front++;
50.         if (front == rear) front = rear = -1;
51.     }
52. }
53. void display() {
54.     if (rear == -1) printf("\nQueue is Empty!!!");
55.     else {
56.         int i;
57.         printf("\nQueue elements are:\n");
58.         for (i = front; i <= rear; i++) printf("%d\t", queue[i]);
59.     }
60. }

```

3.2.1.2 Operations on Queues using Linked Lists:

The major problem with the queue implemented using array is, It will work for only fixed number of data. That means, the amount of data must be specified in the beginning itself. Queue using array is not suitable when we don't know the size of data which we are going to use. A queue data structure can be implemented using linked list data structure. The queue which is implemented using linked list can work for unlimited number of values. That means, queue using linked list can work for variable size of data (No need to fix the size at beginning of the implementation). The Queue implemented using linked list can organize as many data values as we want.

In linked list implementation of a queue, the last inserted node is always pointed by 'rear' and the first node is always pointed by 'front'.



Fig 3.: Queue using Linked List

To implement queue using linked list, we need to follow the below given steps.

- Step 1: Include all the header files which are used in the program. And declare all the user defined functions.
- Step 2: Define a 'Node' structure with two members data and next.
- Step 3: Define two Node pointers 'front' and 'rear' and set both to NULL.

```
1. #include <stdio.h>
2. #include <conio.h>
3. struct Node {
4.     int data;
5.     struct Node * next;
6. } * front = NULL, * rear = NULL;
```

enQueue(value) - Inserting an element into the Queue

We can use the following steps to insert a new node into the queue:

- Step 1: Create a newNode with given value and set 'newNode → next' to NULL.
- Step 2: Check whether queue is Empty (rear == NULL)
- Step 3: If it is Empty then, set front = newNode and rear = newNode.
- Step 4: If it is Not Empty then, set rear → next = newNode and rear = newNode.

```
1. void enQueue(int value) {
2.     struct Node * newNode;
3.     newNode = (struct Node *) malloc(sizeof(struct Node));
4.     newNode -> data = value;
5.     newNode -> next = NULL;
6.     if (front == NULL) front = rear = newNode;
7.     else {
8.         rear -> next = newNode;
```

```

9.     rear = newNode;
10. }
11. }

```

deQueue() - Deleting an Element from Queue

- We can use the following steps to delete a node from the queue:
- Step 1: Check whether queue is Empty (front == NULL).
- Step 2: If it is Empty, then display "Queue is Empty!!! Deletion is not possible!!!" and terminate from the function
- Step 3: If it is Not Empty then, define a Node pointer 'temp' and set it to 'front'.
- Step 4: Then set 'front = front → next' and delete 'temp' (free(temp)).

```

1. void deQueue() {
2.     if (front == NULL) printf("\nQueue is Empty!!!\n");
3.     else {
4.         struct Node * temp = front;
5.         front = front -> next;
6.         printf("\nDeleted element: %d\n", temp -> data);
7.         free(temp);
8.     }
9. }

```

display() - Displaying the elements of Queue

We can use the following steps to display the elements (nodes) of a queue:

- Step 1: Check whether queue is Empty (front == NULL).
- Step 2: If it is Empty then, display 'Queue is Empty!!!' and terminate the function.
- Step 3: If it is Not Empty then, define a Node pointer 'temp' and initialize with front.
- Step 4: Display 'temp → data --->' and move it to the next node. Repeat the same until 'temp' reaches to 'rear' (temp → next != NULL).
- Step 4: Finally! Display 'temp → data ---> NULL'.

```

1. void display() {
2.     if (front == NULL) printf("\nQueue is Empty!!!\n");
3.     else {
4.         struct Node * temp = front;
5.         while (temp -> next != NULL) {
6.             printf("%d--->", temp -> data);
7.             temp = temp -> next;
8.         }
9.         printf("%d--->NULL\n", temp -> data);
10.    }
11. }

```

Queue Implementation Using LINKED LIST & it's operations

```
1. #include < stdio.h >
2. #include < conio.h >
3. struct Node {
4.     int data;
5.     struct Node * next;
6. } * front = NULL, * rear = NULL;
7. void insert(int);
8. void delete();
9. void display();
10. void main() {
11.     int choice, value;
12.     clrscr();
13.     printf("\n:: Queue Implementation using Linked List ::\n");
14.     while (1) {
15.         printf("\n***** MENU *****\n");
16.         printf("1. Insert\n2. Delete\n3. Display\n4. Exit\n");
17.         printf("Enter your choice: ");
18.         scanf("%d", & choice);
19.         switch (choice) {
20.             case 1:
21.                 printf("Enter the value to be insert: ");
22.                 scanf("%d", & value);
23.                 insert(value);
24.                 break;
25.             case 2:
26.                 delete();
27.                 break;
28.             case 3:
29.                 display();
30.                 break;
31.             case 4:
32.                 exit(0);
33.             default:
34.                 printf("\nWrong selection!!! Please try again!!!\n");
35.         }
36.     }
37. }
38. void insert(int value) {
39.     struct Node * newNode;
40.     newNode = (struct Node * ) malloc(sizeof(struct Node));
41.     newNode -> data = value;
42.     newNode -> next = NULL;
43.     if (front == NULL) front = rear = newNode;
44.     else {
45.         rear -> next = newNode;
46.         rear = newNode;
47.     }
48.     printf("\nInsertion is Success!!!\n");
```

```

49. }
50. void delete() {
51.     if (front == NULL) printf("\nQueue is Empty!!!\n");
52.     else {
53.         struct Node * temp = front;
54.         front = front -> next;
55.         printf("\nDeleted element: %d\n", temp -> data);
56.         free(temp);
57.     }
58. }
59. void display() {
60.     if (front == NULL) printf("\nQueue is Empty!!!\n");
61.     else {
62.         struct Node * temp = front;
63.         while (temp -> next != NULL) {
64.             printf("%d--->", temp -> data);
65.             temp = temp -> next;
66.         }
67.         printf("%d--->NULL\n", temp -> data);
68.     }
69. }

```

Differences between Stack and Queue

	Stack	Queue
1	Objects are inserted and removed at the same end .	Objects are inserted and removed from different ends.
2	In stacks only one pointer is used. It points to the top of the stack.	In queues, two different pointers are used for front and rear ends.
3	In stacks, the last inserted object is first to come out.	In queues, the object inserted first is first deleted.
4	Stacks follow Last In First Out (LIFO) order.	Queues following First In First Out (FIFO) order.
5	Stack operations are called push and pop.	Queue operations are called enqueue and dequeue.
6	Stacks are visualized as vertical collections.	Queues are visualized as horizontal collections.
7	Collection of dinner plates at a wedding reception is an example of stack.	People standing in a file to board a bus is an example of queue.