Module II

: **Stack:** Representation and operations on Stack using arrays and linked list, application of Stack - **Polish Notation**- Conversions to Infix, postfix and prefix notations, Infix to postfix conversion using stack, Evaluation of postfix expression using stack **Queue**: Implementation and operations on Queue using arrays and linked list, **Deque**- Types Input and output restricted, **Priority Queues**-Array and Linked list representation . (15hr

STACK DATA STRUCTURE

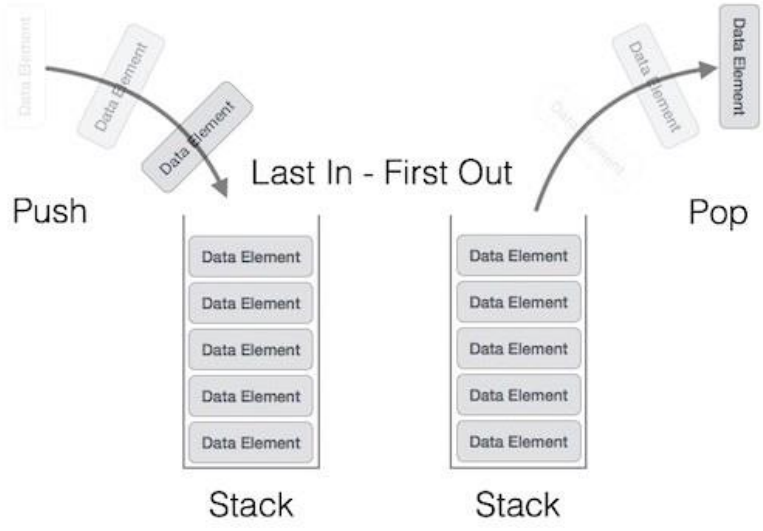
A stack is an Abstract Data Type (ADT), commonly used in most programming languages. It is named stack as it behaves like a real-world stack, for example – a deck of cards or a pile of plates, etc.

A real-world stack allows operations at one end only. For example, we can place or remove a card or plate from the top of the stack only. Likewise, Stack ADT allows all data operations at one end only. At any given time, we can only access the top element of a stack.

This feature makes it **LIFO** data structure. LIFO stands for Last-in-firstout. Here, the element which is placed (inserted or added) last, is accessed first. In stack terminology, insertion operation is called **PUSH** operation and removal operation is called **POP** operation.

Stack Representation

The following diagram depicts a stack and its operations –



A stack can be implemented by means of Array, Structure, Pointer, and Linked List. Stack can either be a fixed size one or it may have a sense of dynamic resizing. Here, we are going to implement stack using arrays, which makes it a fixed size stack implementation.

Basic Operations

Stack operations may involve initializing the stack, using it and then deinitializing it. Apart from these basic stuffs, a stack is used for the following two primary operations −

- **push()** − Pushing (storing) an element on the stack.
- **pop()** − Removing (accessing) an element from the stack.

When data is PUSHed onto stack.

To use a stack efficiently, we need to check the status of stack as well. For the same purpose, the following functionality is added to stacks −

- **peek()** − get the top data element of the stack, without removing it.
- **isFull()** − check if stack is full.
- **IsEmpty()** - check if stack is empty.

At all times, we maintain a pointer to the last PUSHed data on the stack. As this pointer always represents the top of the stack, hence named **top**.

The **top** pointer provides top value of the stack without actually removing it.

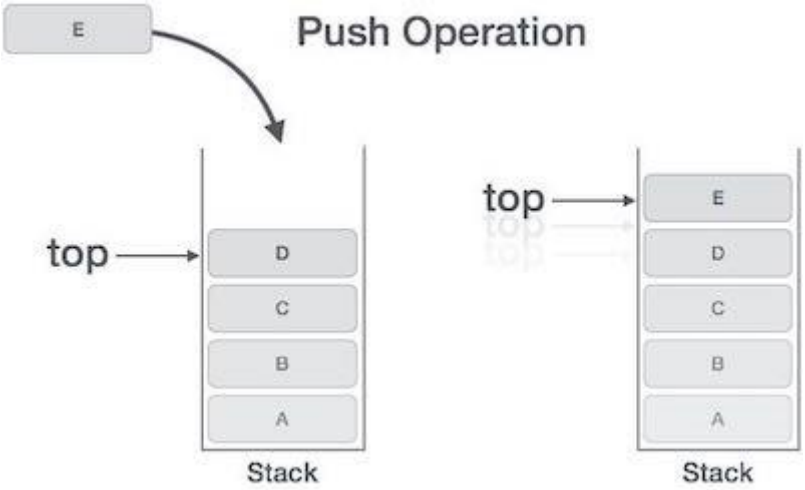
PUSH OPERATION

The process of putting a new data element onto stack is known as a Push

Operation. Push operation involves a series of steps − **Step 1**

− Checks if the stack is full.

- **Step 2** − If the stack is full, produces an error and exit.
- **Step 3** − If the stack is not full, increments **top** to point next empty space.
- **Step 4** − Adds data element to the stack location, where top is pointing.
- **Step 5** − Returns success.



If the linked list is used to implement the stack, then in step 3, we need to allocate space dynamically.

Algorithm for PUSH Operation

A simple algorithm for Push operation can be derived as follows −

```
begin procedure push: stack, data if
   stack is full return null
   endif top ← top + 1
stack[top] ← data end
procedure
```

Implementation of this algorithm in C, is very easy. See the following code −

Example

```
void push(int data)
{
   if(!isFull())
   { top = top + 1;
      stack[top] = data;
   }
   else printf("Could not insert data, Stack is full.\n"); }
```

POP OPERATION

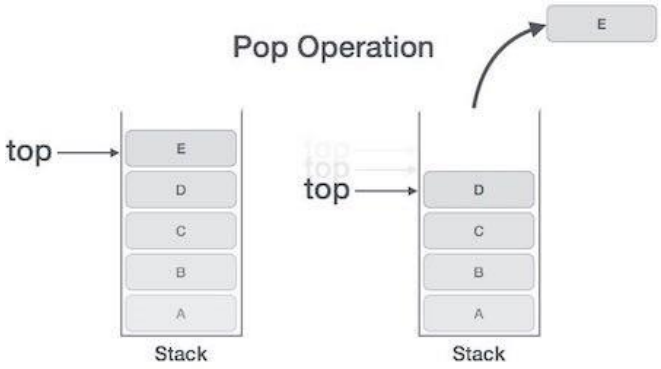
Accessing the content while removing it from the stack, is known as a Pop Operation. In an array implementation of pop() operation, the data element is not actually removed,

instead **top** is decremented to a lower position in the stack to point to the next value. But in linked-list implementation, pop() actually removes data element and deallocates memory space.

A Pop operation may involve the following steps − **Step 1**

− Checks if the stack is empty.

- **Step 2** − If the stack is empty, produces an error and exit.
- **Step 3** − If the stack is not empty, accesses the data element at which **top** is pointing.
- **Step 4** − Decreases the value of top by 1.
- **Step 5** − Returns success.



Algorithm for Pop Operation

A simple algorithm for Pop operation can be derived as follows −

```
begin procedure pop: stack
if stack is empty
   return null
  endif

  data ← stack[top]
  top ← top - 1
return data end
procedure
```

Implementation of this algorithm in C, is as follows −

Example

```
int pop(int data)
{
   if(!isempty())
  { data = stack[top]; top
      = top - 1;
      return data;
   } else printf("Could not retrieve data, Stack is empty.\n"); }
```

peek()

Algorithm of peek() function −

```
begin procedure peek
return stack[top]    end
procedure
```

Implementation of peek() in C language

```
int peek()
{
return stack[top];
}
```

isfull()

Algorithm of isfull() function −

```
BEGIN
if top equals to MAXSIZE
```

```
    return true
  else return
    false
  endif
end procedure
```

Implementation of isfull() in C language

```
bool isfull()
{ if(top == MAXSIZE)
  return true;
  else return
    false;
}
```

isempty()

Algorithm of isempty() function −

```
begin procedure isempty if
  top less than 1 return true
  else return
    false
  endif end
  procedure
```

Implementation of isempty() function in C programming language is slightly different. We initialize top at -1, as the index in array starts from 0. So we check if the top is below zero or -1 to determine if the stack is empty.

Here's the code −

Implementation of isempty() function in C

```
bool
isempty()
{ if(top == -1)
  return true;
  else return
    false;

}
```

Implementation of Stack using Array in C

```
#include <stdio.h>
int MAXSIZE = 8;
int stack[8]; int
top = -1; int
isempty()
{ if(top == -1)
   return 1;
   else return
      0;
}

int isfull()
{ if(top == MAXSIZE)
   return 1;
   else return
      0;
}

int peek()
{
   return stack[top];
}

int pop()
{
   int data;

   if(!isempty())
  { data = stack[top]; top
      = top - 1;
      return data;
   } else printf("Could not retrieve data, Stack is empty.\n");
}

int push(int data)
{
```

```
   if(!isfull())
  { top = top + 1;
      stack[top] = data;
   } else
      printf("Could not insert data, Stack is full.\n");
}

int main()
{
   // push items on to the stack
   push(3); push(5);
   push(9); push(1);
   push(12);
   push(15);

   printf("Element at top of the stack: %d\n" ,peek()); printf("Elements: \n");

   // print stack data while(!isempty())
 {
      int data = pop();
      printf("%d\n",data);
   }
   printf("Stack full: %s\n" , isfull()?"true":"false"); printf("Stack empty:
   %s\n" , isempty()?"true":"false"); return 0;
}
```

If we compile and run the above program, it will produce the following result −

Output

```
Element at top of the stack: 15 Elements:
15
12
1
9
5
3
Stack full: false
```

```
Stack empty: true
```

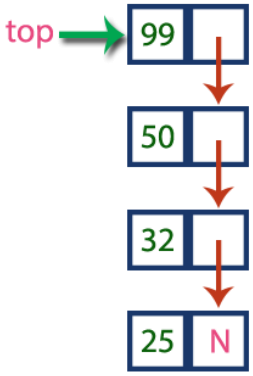
STACK USING LINKED LIST

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**.

Example



In above example, the last inserted node is 99 and the first inserted node is 25. The order of elements inserted is 25, 32,50 and 99.

Operations

To implement stack using linked list, we need to set the following things before implementing actual operations.

Step 1: Include all theheader fileswhich are used in the program.

And declare all theuser defined functions.

Step 2: Define a 'Node' structure with two membersdataandnext.

Step 3: Define aNodepointer 'top' and set it toNULL.

Step 4: Implement themainmethod by displaying Menu with list of operations and make suitable function calls in themainmethod.

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 anewNodewith given value.

Step 2: Check whether stack isEmpty(top == NULL) Step 3: If it isEmpty, then setnewNode → next= NULL.

Step 4: If it isNot Empty, then setnewNode → next= top.

Step 5: Finally, settop = newNode.

pop() - Deleting an Element from a Stack

Step 1: Check whetherstackisEmpty(top == NULL).

Step 2: If it isEmpty, then display"Stack is Empty!!! Deletion is not possible!!!" and terminate the function Step 3: If it isNot Empty, then define aNodepointer 'temp' and set it to 'top'.

Step 4: Then set 'top = top → next'.

Step 7: Finally, delete 'temp' (free(temp)).

display() - Displaying stack of elements

Step 1: Check whether stack isEmpty(top == NULL).

Step 2: If it isEmpty, then display'Stack is Empty!!!' and terminate the function.

Step 3: If it isNot Empty, then define a Node pointer'temp' and initialize withtop.

Step 4: Display 'temp → data--->' and move it to the next node. Repeat the same untiltemp reaches to the first node in the stack (temp → next!= NULL).

Step 4: Finally! Display 'temp → data---> NULL'.

Program for Stack Using Linked List

```
#include<stdio.h>
#include<conio.h> struct
Node
{
   int data;
```

```
   struct Node *next; }*top =
NULL;

void push(int);
void pop(); void
display();

void main()
{
   int choice, value;
   clrscr(); printf("\n:: Stack using Linked List
   ::\n"); while(1){
      printf("\n****** MENU ******\n"); printf("1. Push\n2.
      Pop\n3. Display\n4. Exit\n"); printf("Enter your choice: ");
      scanf("%d",&choice); switch(choice){ case 1:
      printf("Enter the value to be insert: "); scanf("%d",
      &value); push(value); break;
            case 2: pop(); break; case 3:
            display(); break; case 4:
            exit(0);
default: printf("\nWrong selection!!! Please try again!!!\n");
     }
  }
}

void push(int value)
{ struct Node *newNode; newNode = (struct
   Node*)malloc(sizeof(struct Node)); newNode->data = value;
   if(top == NULL)
      newNode->next = NULL;
   else
      newNode->next = top;
   top = newNode;
   printf("\nInsertion is Success!!!\n");
}

void pop()
{ if(top == NULL)
      printf("\nStack is Empty!!!\n");
   else
  {
      struct Node *temp = top; printf("\nDeleted element:
      %d", temp->data); top = temp->next;
      free(temp);
  }
}

void display()
{ if(top == NULL)
      printf("\nStack is Empty!!!\n");
```

```
   else{
      struct Node *temp = top; while(temp->next !=
      NULL){ printf("%d--->",temp->data);
              temp = temp -> next;
     }
      printf("%d--->NULL",temp->data);
   }
}
```

Application of Stack

The stack data structure can be use to implement the following concepts:

1. Parsing
2. Recursive Function
3. Calling Function
4. Expression Evaluation
5. Expression Conversion ○ Infix to Postfix ○ Infix to Prefix ○ Postfix to Infix ○ Prefix to Infix
6. Towers of hanoi

EXPRESSION PARSING

The way to write arithmetic expression is known as a **notation**. An arithmetic expression can be written in three different but equivalent notations, i.e., without changing the essence or output of an expression. These notations are −

- Infix Notation
- Prefix (Polish) Notation
- Postfix (Reverse-Polish) Notation

These notations are named as how they use operator in expression.

Infix Notation

Operators are written in-between their operands. We write expression in **infix** notation, e.g. a - b + c, where operators are used **in**-between operands. It is easy for us humans to read, write, and speak in infix notation but the same does not go well with computing devices. An algorithm to process infix notation could be difficult and costly in terms of time and space consumption.

Prefix Notation (Polish Notation)

Operators are written before their operands. In this notation, operator is **prefix**ed to operands, i.e. operator is written ahead of operands. For example, **+ab**. This is equivalent to its infix notation **a + b**. Prefix notation is also known as **Polish Notation**.

Postfix Notation (Reversed Polish Notation)

Operators are written after their operands. This notation style is known as **Reversed Polish Notation**. In this notation style, the operator is **postfix**ed to the operands i.e., the operator is written after the operands. For example, **ab+**.
This is equivalent to its infix notation **a + b**.

The following table briefly tries to show the difference in all three notations −

Sr.No.	Infix Notation	Prefix Notation	Postfix Notation
1	a + b	+ a b	a b +
2	(a + b) ∗ c	∗ + a b c	a b + c ∗
3	a ∗ (b + c)	∗ a + b c	a b c + ∗
4	a / b + c / d	+ / a b / c d	a b / c d / +
5	(a + b) ∗ (c + d)	∗ + a b + c d	a b + c d + ∗
6	((a + b) ∗ c) - d	- ∗ + a b c d	a b + c ∗ d -

Parsing Expressions

As we have discussed, it is not a very efficient way to design an algorithm or program to parse infix notations. Instead, these infix notations are first converted into either postfix or prefix notations and then computed.

To parse any arithmetic expression, we need to take care of operator precedence and associativity also.

Precedence

When an operand is in between two different operators, which operator will take the operand first, is decided by the precedence of an operator over others. For example −

As multiplication operation has precedence over addition, b * c will be evaluated first. A table of operator precedence is asfollows:

Operators	**Symbols**
Parenthesis	{ }, ( ), [ ]
Exponential notation	^
Multiplication and Division	*, /
Addition and Subtraction	+, -

Associativity

Associativity describes the rule where operators with the same precedence appear in an expression. For example, in expression a + b − c, both + and – have the same precedence, then which part of the expression will be evaluated first, is determined by associativity of those operators. Here, both + and − are left associative, so the expression will be evaluated as **(a + b) − c**.

Precedence and associativity determines the order of evaluation of an expression. Following is an operator precedence and associativity table

(highest to lowest) −

Sr.No.	**Operator**	**Precedence**	**Associativity**
1	Exponentiation ^	Highest	Right to Left
2	Multiplication (∗) & Division (/)	Second Highest	Left to Right
3	Addition (+) & Subtraction (−)	Lowest	Left to Right

The above table shows the default behavior of operators. At any point of time in expression evaluation, the order can be altered by using parenthesis.

For example −

In **a + b*c**, the expression part **b*c** will be evaluated first, with multiplication as precedence over addition. We here use parenthesis for **a +**

b to be evaluated first, like **(a + b)*c**.

POLISH NOTATION

Polish notation (PN), also known as normal Polish notation (NPN), Łukasiewicz notation, Warsaw notation, Polish prefix notation or simply prefix notation, is a form of notation for logic, arithmetic, and algebra. Its distinguishing feature is that it places operators to the left of their operands. If the arity of the operators is fixed, the result is a syntax lacking parentheses or other brackets that can still be parsed without ambiguity.
The Polish logician Jan Łukasiewicz invented this notation in 1924 in order to simplify sentential logic.

The term Polish notation is sometimes taken (as the opposite of infix notation) to also include Polish postfix notation, or reverse Polish notation (RPN), in which the operator is placed after the operands.

When Polish notation is used as a syntax for mathematical expressions by programming language interpreters, it is readily parsed into abstract syntax trees and can, in fact, define a one-to-one representation for the same. Because of this, Lisp (see below) and related programming languages define their entire syntax in terms of prefix notation (and others use postfix notation).

Infix to Postfix Conversion using Stack

Example

Infix Expression : 3+4*5/6

Step 1 : Initially Stack is Empty ans the very first literal of Infix Expression is '3' which is operand hence push it on output stack.

```
Stack  :
Output : 3
```

Step 2 : Next literal of expression is + which is operand, hence needed to be pushed on stack but intially stack is empty hence literal will directly pushed on to stack.

```
Stack  : +
Output : 3
```

Step 3 : Further 4 is an operand should be pushed on stack.

```
Stack  : +
Output : 3 4
```

Step 4 : Next literal is * which is an operator, as stack is not empty, priority should be checked of instack operator(top of stack) and of incoming operator i.e * as priority of instack operator is less than incoming operator, * will be pushed on to stack.

```
Stack  : + *
Output : 3 4
```

Step 5 : Next literal is 5 which is an operand, hence should be pushed on to output stack.

```
Stack  : + *
Output : 3 4 5
```

Step 6 : Next literal is / which is an operator, as stack is not empty, priority should be checked of instack operator(top of stack) i.e * and of incoming operator i.e /, as priority of / and * are equal hence * will be poped out of stack and will be stored on output stack and operator / will be stored on stack.

```
Stack  : + /
Output : 3 4 5 *
```

Step 7 : Next literal is 6 which is an operand, hence should be pushed on output stack.

```
Stack  : + /
Output : 3 4 5 * 6
```

Step 8 : As now all litrals are traversed, despite stack is not empty, hence pop all litrals from stack and pushed it on to output stack.
Postfix Expression : 3 4 5 * 6 / +

Another Example 2

For a better understanding, let's trace out an example: **A * B- (C + D) + E**

Input Character	Operation on Stack	Stack	Postfix Expression
A		Empty	A
*	Push	*	A
B		*	AB
–	Check and Push	–	AB*
(	Push	-(	AB*
C		-(	AB*C
+	Check and Push	-(+	AB*C
D			AB*CD
)	Pop and Append to Postfix till '('	–	AB*CD+

+	Check and Push	+	AB*CD+-
E		+	AB*CD+-E
End	Pop till empty		**AB*CD+-E+**

Evaluation rule of a Postfix Expression states:

Postfix Evaluation Algorithm

Step 1 − scan the expression from left to right

Step 2 − if it is an operand push it to stack

Step 3 − if it is an operator pull operand from stack and perform operation

Step 4 − store the output of step 3, back to stack

Step 5 − scan the expression until all operands are consumed

Step	Input Symbol	Operation	Stack	Calculation
1.	4	Push	4	
2.	5	Push	4,5	
3.	6	Push	4,5,6	
4.	*	Pop(2 elements) & Evaluate	4	5*6=30
5.		Push result(30)	4,30	
6.	+	Pop(2 elements) & Evaluate	Empty	4+30=34
7.		Push result(34)	34	
8.		No-more elements(pop)	Empty	34(Result)

QUEUE DATA STRUCTURE

Queue is an abstract data structure, somewhat similar to Stacks. Unlike stacks, a queue is open at both its ends. One end is always used to insert data (enqueue) and the other is used to

remove data (dequeue). Queue follows FirstIn-First-Out methodology, i.e., the data item stored first will be accessed first.

A real-world example of queue can be a single-lane one-way road, where the vehicle enters first, exits first. More real-world examples can be seen as queues at the ticket windows and bus-stops.

Queue Representation

As we now understand that in queue, we access both ends for different reasons. The following diagram given below tries to explain queue
representation as data structure −

As in stacks, a queue can also be implemented using Arrays, Linked-lists, Pointers and Structures. For the sake of simplicity, we shall implement queues using one-dimensional array.

Basic Operations

Queue operations may involve initializing or defining the queue, utilizing it, and then completely erasing it from the memory. Here we shall try to understand the basic operations associated with queues −

- enqueue() − add (store) an item to the queue.

- dequeue() − remove (access) an item from the queue.

Few more functions are required to make the above-mentioned queue operation efficient. These are −

- peek() − Gets the element at the front of the queue without removing it.
- isfull() − Checks if the queue is full.
- isempty() − Checks if the queue is empty.

In queue, we always dequeue (or access) data, pointed by front pointer and while enqueing (or storing) data in the queue we take help of rear pointer.

Let's first learn about supportive functions of a queue −

peek() this function helps to see the data at the front of the queue. The algorithm of peek() function is as follows:

```
begin procedure peek

return queue[front] end

procedure
```

Implementation of peek() function in C programming language −

```
int peek()
{
      return queue[front];

}
```

isfull()

As we are using single dimension array to implement queue, we just check for the rear pointer to reach at MAXSIZE to determine that the queue is full. In case we maintain the queue in a circular linked-list, the algorithm will differ. Algorithm of isfull() function −

```
begin procedure isfull
   if rear equals to MAXSIZE return true
   else return
      false
   endif end
   procedure
```

Implementation of isfull() function in C programming language −

Example

```
bool isfull()
{ if(rear == MAXSIZE - 1)
return true;
else
return false;
}
```

isempty()

Algorithm of isempty() function −

```
begin procedure isempty

   if front is less than MIN  OR front is greater than rear return true
   else return
      false
   endif end
 procedure
```

If the value of **front**is less than MIN or 0, it tells that the queue is not yet initialized, hence empty.

Here's the C programming code −

```
bool isempty()
{
   if(front < 0 || front > rear) return
      true;
   else return
      false;
}
```

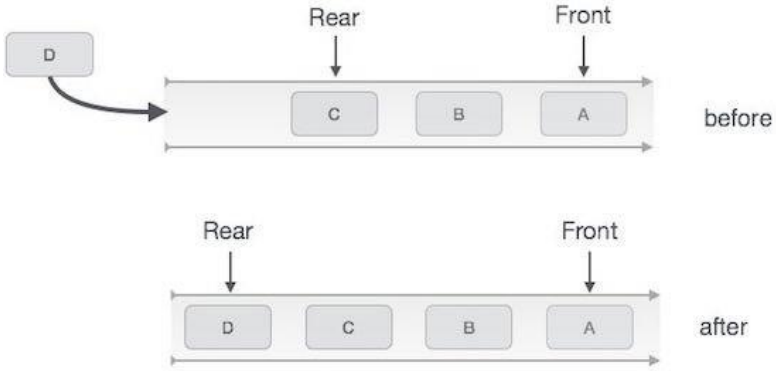
Enqueue Operation

Queues maintain two data pointers, **front**and **rear**. Therefore, its operations are comparatively difficult to implement than that of stacks.

The following steps should be taken to enqueue (insert) data into a queue − **Step 1** −

Check if the queue is full.

- **Step 2** − If the queue is full, produce overflow error and exit.
- **Step 3** − If the queue is not full, increment **rear**pointer to point the next empty space.
- **Step 4** − Add data element to the queue location, where the rear is pointing.
- **Step 5 –** return success



Queue Enqueue

Sometimes, we also check to see if a queue is initialized or not, to handle any unforeseen situations.

Algorithm for enqueue operation

```
procedure enqueue(data) if
   queue is full return
   overflow
   endif rear ← rear + 1
   queue[rear] ← data
    return true end
procedure
```

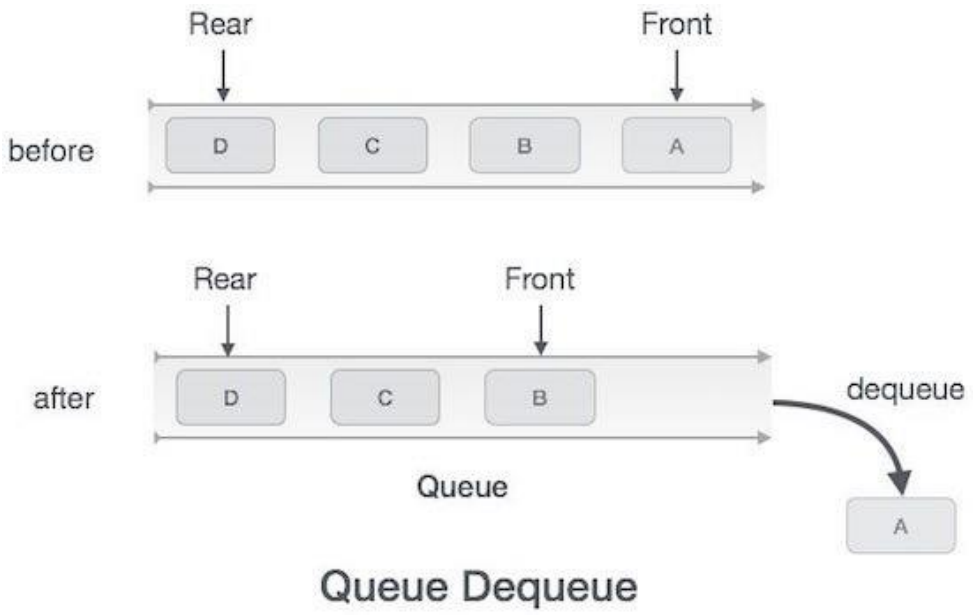
Implementation of enqueue() in C programming language − **Example**

```
int enqueue(int data)

if(isfull()) return 0;

rear = rear + 1;

queue[rear] = data;

return 1; end

procedure
```

Dequeue Operation

Accessing data from the queue is a process of two tasks − access the data where **front**is pointing and remove the data after access. The following steps are taken to perform **dequeue**operation − **Step 1** − Check if the queue is empty.

- **Step 2** – If the queue is empty, produce underflow error and exit.
- **Step 3** - If the queue is not empty, access the data where **front** is pointing.
- **Step 4** − Increment **front**pointer to point to the next available data element.
- **Step 5** − Return success.



Queue Dequeue

Algorithm for dequeue operation

```
procedure dequeue if queue is empty
return underflow
end if

data = queue[front]
front ← front + 1
return true

end procedure
```

Implementation of dequeue() in C programming language −

```
int dequeue() {
if(isempty())
 return 0; int
 data =
 queue[front];
 front = front +
 1;
```

```
return data;
}
```

Implementation of QUEUEusing Array in C language

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <stdbool.h>
#define MAX 6 int
intArray[MAX]; int
front = 0; int rear = -1;
int itemCount = 0;

int peek()
{ return intArray[front];
}

bool isEmpty()
{       return itemCount == 0;
}

bool isFull()
{        return itemCount == MAX;
}

int size()
{      return itemCount;
```

```
}

void insert(int data)
{
   if(!isFull())
  { if(rear == MAX-1)
       { rear = -1;
       }
       intArray[++rear] = data; itemCount++;
   }
}

int removeData()
{ int data = intArray[front++];
   if(front == MAX)
       front = 0;
   itemCount--; return
   data;
} int main() {
   /* insert 5 items */
   insert(3); insert(5);
   insert(9); insert(1);
   insert(12);
   // front : 0
   // rear  : 4
   // ----------------//
   index : 0 1 2 3 4
```

```
// ------------------
// queue : 3 5 9 1 12

insert(15); // front : 0

// rear  : 5
// --------------------// index
: 0 1 2 3 4  5
// ---------------------
// queue : 3 5 9 1 12 15

if(isFull()){ printf("Queue is

full!\n");

}

// remove one item

int num = removeData(); printf("Element
removed: %d\n",num);

// front : 1
// rear  : 5
// ------------------// index
: 1 2 3 4  5

// -------------------

// queue : 5 9 1 12 15 //

insert more items

insert(16); // front : 1

// rear  : -1

// ---------------------// index

: 0  1 2 3 4  5

// ----------------------

// queue : 16 5 9 1 12 15


// As queue is full, elements will not be inserted.

insert(17);
```

```
   insert(18);
   // ----------------------// index : 0  1 2 3 4  5
   // ----------------------
   // queue : 16 5 9 1 12 15
   printf("Element at front: %d\n",peek()); printf("----------------------\n"); printf("index : 5 4 3
   2  1  0\n"); printf("----------------------\n"); printf("Queue:  "); while(!isEmpty()) {
      int n = removeData(); printf("%d ",n);
   }
}
```

If we compile and run the above program, it will produce the following result −

Output

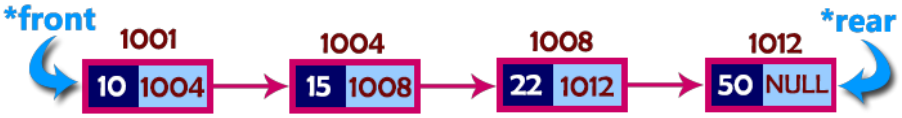
```
Queue is full!
Element removed: 3
Element at front: 5 ----------------------index : 5 4 3 2 1 0
----------------------
Queue: 5 9 1 12 15 16
```

QUEUE USING LINKED LIST

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**'.

Example



In above example, the last inserted node is 50 and it is pointed by '**rear**' and the first inserted node is 10 and it is pointed by '**front**'. The order of elements inserted is 10, 15, 22 and 50.

Operations

To implement queue using linked list, we need to set the following things before implementing actual operations.

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**.

Step 4: Implement the **main**method by displaying Menu of list of operations and make suitable function calls in the **main**method to perform user selected operation.

enQueue(value) - Inserting an element 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**.

deQueue() - Deleting an Element from 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)**).

display() - Displaying the elements of 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**'.

Implementation of QUEUE using Link List in C language

```
#include<stdio.h>
#include<conio.h>
struct Node
{
  int data;
  struct Node *next;
}*front = NULL,*rear = NULL;

void insert(int); void
delete(); void
display(); void
main()
{
  int choice, value; clrscr(); printf("\n:: Queue Implementation using
  Linked List ::\n"); while(1){
    printf("\n****** MENU ******\n"); printf("1. Insert\n2.
    Delete\n3. Display\n4. Exit\n"); printf("Enter your choice:
    "); scanf("%d",&choice); switch(choice){
      case 1: printf("Enter the value to be insert: ");
              scanf("%d", &value); insert(value); break;
      case 2: delete(); break; case 3:
      display(); break; case 4:
      exit(0);
default: printf("\nWrong selection!!! Please try again!!!\n");
    }
  }
}
void insert(int value)
{ struct Node *newNode; newNode = (struct
  Node*)malloc(sizeof(struct Node)); newNode->data = value;
  newNode -> next = NULL; if(front == NULL)
    front = rear = newNode;
  else{
    rear -> next = newNode;
    rear = newNode;
  } printf("\nInsertion is Success!!!\n");
```

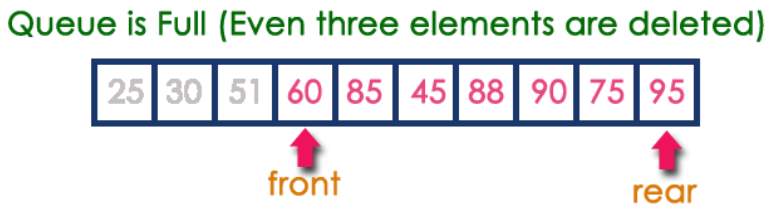
```
}
void delete()
{ if(front == NULL)
      printf("\nQueue is Empty!!!\n");
   else{
      struct Node *temp = front; front = front -> next;
      printf("\nDeleted element: %d\n", temp->data);
      free(temp);
   }
}
void display()
{ if(front == NULL)
      printf("\nQueue is Empty!!!\n");
   else{
      struct Node *temp = front; while(temp->next !=
      NULL)
        { printf("%d--->",temp->data);
          temp = temp -> next;
         } printf("%d--->NULL\n",temp->data);
   }
}
```

CIRCULAR QUEUE

In a normal Queue Data Structure, we can insert elements until queue becomes full. But once if queue becomes full, we cannot insert the next element until all the elements are deleted from the queue. For example consider the queue below...

Ater inserting all the elements into the queue.

Now consider the following situation after deleting three elements from the queue...

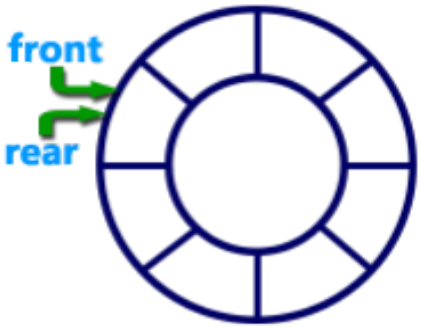


This situation also says that Queue is Full and we can not insert the new element because, '**rear**' is still at last position. In above situation, even though we have empty positions in the queue we can not make use of them to insert new element. This is the major problem in normal queue data structure. To overcome this problem we use circular queue data structure.

What is Circular Queue?

Circular Queue is a linear data structure in which the operations are performed based on FIFO (First In First Out) principle and the last position is connected back to the first position to make a circle.

Graphical representation of a circular queue is as follows...



DOUBLE ENDED QUEUE (DEQUEUE)

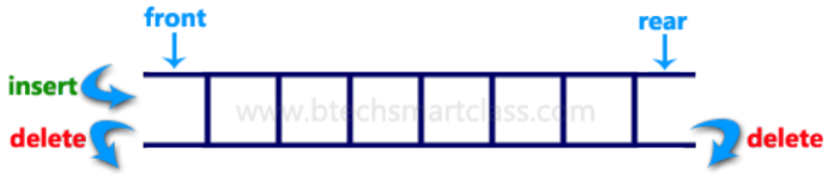
Double Ended Queue is also a Queue data structure in which the insertion and deletion operations are performed at both the ends (**front**and **rear**). That means, we can insert at both front and rear positions and can delete from both front and rear positions.

Double Ended Queue can be represented in TWO ways, those are as follows...

1. Input Restricted Double Ended Queue
2. Output Restricted Double Ended Queue

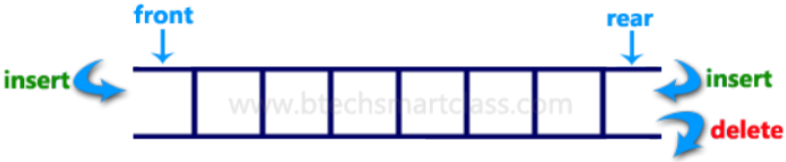
Input Restricted Double Ended Queue

In input restricted double ended queue, the insertion operation is performed at only one end and deletion operation is performed at both the ends.



Output Restricted Double Ended Queue

In output restricted double ended queue, the deletion operation is performed at only one end and insertion operation is performed at both the ends.



Priority queue

A priority queue is an abstract data type that behaves similarly to the normal queue except that each element has some priority, i.e., the element with the highest priority would come first in a priority queue. The priority of the elements in a priority queue will determine the order in which elements are removed from the priority queue.

The priority queue supports only comparable elements, which means that the elements are either arranged in an ascending or descending order.

For example, suppose we have some values like 1, 3, 4, 8, 14, 22 inserted in a priority queue with an ordering imposed on the values is from least to the greatest. Therefore, the 1 number would be having the highest priority while 22 will be having the lowest priority.

Characteristics of a Priority queue

A priority queue is an extension of a queue that contains the following characteristic

- Every element in a priority queue has some priority associated with it.
- An element with the higher priority will be deleted before the deletion of the lesser priority.
- If two elements in a priority queue have the same priority, they will be arranged using the FIFO principle.

Priority Queue using Linked List

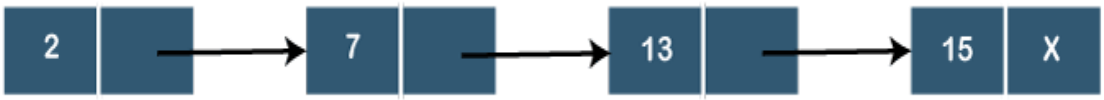
Implement Priority Queue using Linked Lists.

- **push():** This function is used to insert a new data into the queue.
- **pop():** This function removes the element with the highest priority from the queue.
- **peek() / top():** This function is used to get the highest priority element in the queue without removing it from the queue.

Priority Queues can be implemented using common data structures like arrays, linked-lists, heaps and binary trees.

Let's understand through an example.

Consider the below-linked list that consists of elements 2, 7, 13, 15.



Suppose we want to add the node that contains the value 1. Since the value 1 has more priority than the other nodes so we will insert the node at the beginning of the list shown as below:

Now we have to add 7 element to the linked list. We will traverse the list to insert element 7. First, we will compare element 7 with 1; since 7 has lower priority than 1, so it will not be inserted before 7. Element 7 will be compared with the next node, i.e., 2; since element 7 has a lower priority than 2, it will not be inserted before 2.. Now, the element 7 is compared with a next element, i.e., since both the elements have the same priority so they will be served based on the first come first serve. The new element 7 will be added after the element 7 shown as below: