JEPPIAAR INSTITUTE OF TECHNOLOGY

"Self Belief | Self Discipline | Self Respect"

DEPARTMENT OF

COMPUTER SCIENCE AND ENGINEERING

LECTURE NOTES

CS8391 / DATA STRUCTURES

(2017 Regulation)

Year/Semester: II / III

Prepared by

Dr. K. Tamilarasi, Professor / Dept. of CSE.

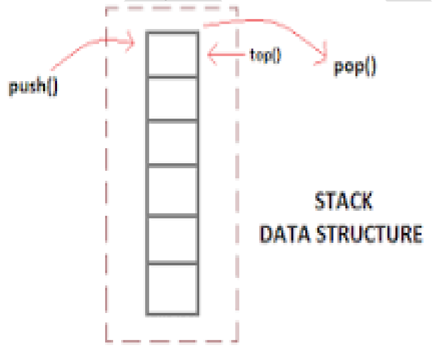
CS8391 DATA STRUCTURES

UNIT II LINEAR DATA STRUCTURES – STACKS, QUEUES

Stack ADT – Operations - Applications - Evaluating arithmetic expressions- Conversion of Infix to postfix expression - Queue ADT – Operations - Circular Queue – Priority Queue - deQueue – applications of queues.

Stack ADT:

- **Stack is a Linear Data Structure that follows Last in First Out (LIFO) principle.**
- **Stack is an ordered list in which, insertion and deletion can be performed only at one end that is called top**
- i.e. the element which is inserted last in the stack, will be deleted first from the stack.
- Example: - Pile of coins, stack of trays



TOP pointer

- It will always point to the last element inserted in the stack. For empty stack, top will be pointing to -1. (TOP = -1) .

Operations On stack :

- Two fundamental operations performed on the stack are **PUSH and POP**.
- Return the top element , **stack overflow, and stack underflow** are other additional functions.

(a) PUSH:

- It is the process of inserting a new element at the Top of the stack.

For every push operation:
1. Check for Full stack (overflow).
2. Increment Top by 1. (Top = Top + 1)
3. Insert the element X in the Top of the stack.

Procedure for PUSH

```
void push ()
{   int val;
   if (top == n)
   printf("\n Overflow");
   else
   {
      printf("Enter the value?");
      scanf("%d",&val);
      top = top +1;
      stack[top] = val;
   }
}
```

(b)POP:

- It is the process of deleting the Top element of the stack.

For every pop operation:
1. Check for Empty stack (underflow).
2. Delete (pop) the Top element X from the stack
3. Decrement the Top by 1. (Top = Top - 1)

Procedure for POP

```
void pop ()
{
   if(top == -1)
   printf("Underflow");
   else
   top = top -1;
}
```

(C) Return or Peek

- Peek operation involves returning the element which is present at the top of the stack without deleting it.

- Underflow condition can occur if we try to return the top element in an empty stack.

Procedure for return or peek

```
void show()
{
   for (i=top;i>=0;i--)
   {
      printf("%d\n",stack[i]);
   }
   if(top == -1)
   {
      printf("Stack is empty");
   }
}
```

Exceptional Conditions of stack

1. **Stack Overflow**

- An Attempt to insert an element X when the stack is Full, is said to be stack overflow.
- For every Push operation, we need to check this condition.

2. **Stack Underflow:**

- An Attempt to delete an element when the stack is empty, is said to be stack underflow.
- For every Pop operation, we need to check this condition.

Implementation of Stack

Stack can be implemented in 2 ways.
1. Static Implementation (Array implementation of Stack)

2. Dynamic Implementation (Linked List Implementation of Stack)

Array Implementation of Stack

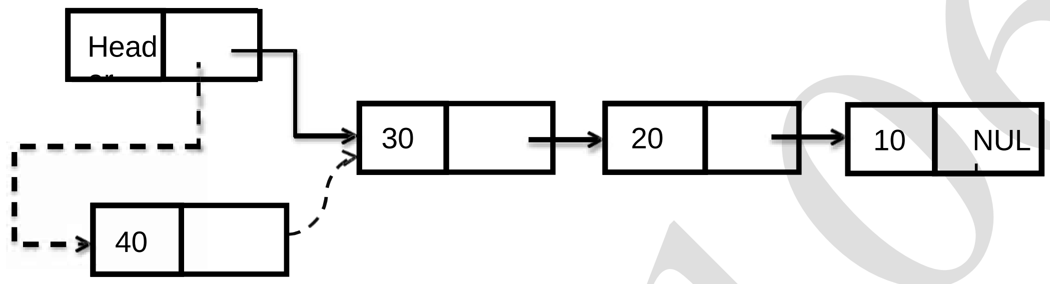
```
#include <stdio.h>
int stack[100],i,j,choice=0,n,top=-1;
void push();
void pop();
void show();
void main ()
{
   printf("Enter the number of elements in the stack ");
   scanf("%d",&n);
   while(choice != 4)
   {  printf("Chose one from the below options...\n");
      printf("\n1.Push\n2.Pop\n3.Show\n4.Exit");
      printf("\n Enter your choice \n");
      scanf("%d",&choice);
      switch(choice)
      {     case 1:    push();
                       break;
            case 2:    pop();
                       break;
           case 3:    show();
                      break;
          case 4:   printf("Exit....");
                    break;
         }
         default:  printf("Please Enter valid choice ");
        };
   }  }
```

Linked List Implementation of Stack

- In linked list implementation of stack, the nodes are maintained non-contiguously in the memory.
- Each node contains a pointer to its immediate successor node in the stack.
- Stack is said to be overflown if the space left in the memory heap is not enough to create a node.

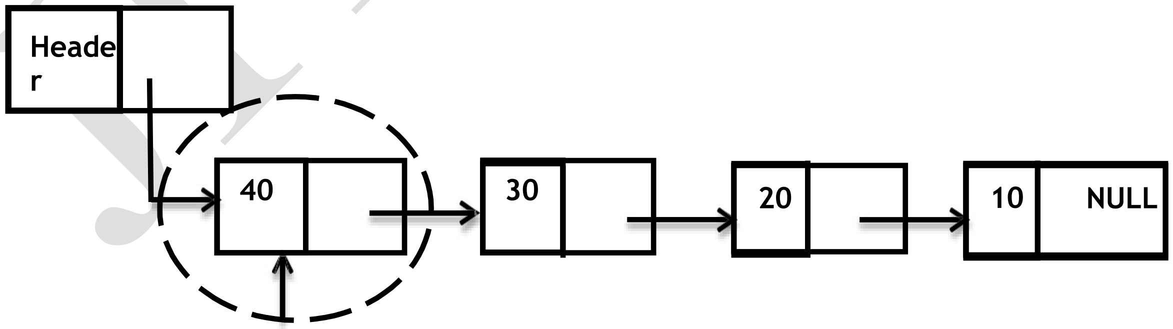
PUSH Operation using Linked List

- Inserting a node from the top of stack is referred to as push operation.
- Create a node first and allocate memory to it.
- If the list is empty then the item is to be pushed as the start node of the list.
- If there are some nodes in the list already, then we have to add the new element in the **beginning of the list**
- For this purpose, **assign the address of the starting element to the address field of the new node and make the new node, the starting node of the list**.



(POP operation) Deleting a node from the stack

- Deleting a node from the top of stack is referred to as pop operation.
- Check for the underflow condition: The underflow condition occurs when we try to pop from an already empty stack. The stack will be empty if the head pointer of the list points to null.
- Adjust the head pointer accordingly: In stack, the elements are popped only from one end, therefore, the value stored in the head pointer must be deleted and the node must be freed. The next node of the head node now becomes the head node.



Program for Linked list implementation stack

```
#include <stdio.h>
```

```
#include <stdlib.h>
void push();
void pop();
void display();
struct node
{   int val;
struct node *next;
};   struct node *head;

void main ()
{     int choice=0;
   printf("\n*********Stack operations using linked list*********\n");
   printf("\n----------------------------------------------\n");
   while(choice != 4)
   {  printf("\n\nChose one from the below options...\n");
      printf("\n1.Push\n2.Pop\n3.Show\n4.Exit");
      printf("\n Enter your choice \n");
      scanf("%d",&choice);
      switch(choice)
      {  case 1:  push();
                  break;
         case 2:  pop();
                  break;
         case 3:  display();
                  break;          }
         case 4:   printf("Exiting....");
                  break;

         default:  printf("Please Enter valid choice ");
  };
}
}
void push ()
{  int val;
   struct node *ptr = (struct node*)malloc(sizeof(struct node));
   if(ptr == NULL)
   {        printf("not able to push the element");      }
   else
   {
      printf("Enter the value");
```

```
        scanf("%d",&val);
        if(head==NULL)
        {
           ptr->val = val;
           ptr -> next = NULL;
           head=ptr;
        }
        else
        {
           ptr->val = val;
           ptr->next = head;
           head=ptr;          }
        printf("Item pushed");
      }
}

void pop()
{
    int item;
    struct node *ptr;
    if (head == NULL)
    {          printf("Underflow");      }
    else
    {
        item = head->val;
        ptr = head;
        head = head->next;
        free(ptr);
        printf("Item popped");
    }
}
void display()
{
    int i;
    struct node *ptr;
    ptr=head;
    if(ptr == NULL)
    {          printf("Stack is empty\n");       }
    else
    {
        printf("Printing Stack elements \n");
        while(ptr!=NULL)
        {
           printf("%d\n",ptr->val);
           ptr = ptr->next;
        }
    }
}
```

Applications of Stack

The following are some of the applications of stack:

1. Evaluating the arithmetic expressions
    - Conversion of Infix to Postfix Expression
    - Evaluating the Postfix Expression
    - Balancing the Symbols
2. Function Call
3. Tower of Hanoi
4. 8 Queen Problem

Evaluating the Arithmetic Expression:
1.Conversion of Infix to Postfix Expression

There are 3 types of Expressions

- Infix Expression
- Postfix Expression
- Prefix Expression

INFIX: The arithmetic operator appears between the two operands to which it is being applied.

Example1 : A + B
Example2 : A / B + C

POSTFIX: The arithmetic operator appears directly after the two operands to which it applies. Also called reverse polish notation.

Example1 : A B +
Example2 : A B / C +

PREFIX: The arithmetic operator is placed before the two operands to which it applies. Also called polish notation
Example1 : + A B
Example2 :   + / A B C

Algorithm to convert Infix Expression to Postfix Expression:

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 the output.

Step 2: If the Character is Operator,
Step 2.1: If the Top of Stack operator has a **higher or equal priority than input operator, then pop that top of stack operator** and place it onto the output. Then push the input operator.
Step 2.2: If the Top of Stack operator has a **less priority than input operator,** then push the input operator.

Step 3: If the character is '(' , push it onto the stack

Step 4: If the character is a ')' , pop all the operators from the stack till it encounters '(', and discard both the parenthesis in the output.
Step 5: If the Charater is '#' pop all the character from stack

Eg: Consider the following Infix expression: - A / (B + C) * D
Convert A / (B + C) * D to postfix notation.

Input : add # in input expression and write A / (B + C) * D #

Input Character	Stack (Note :Top of Stack is bold)	Postfix Expression	Action Taken
A		A	Output A
/	/	A	Push /
(	/ **(**	A	Push (
B	/ **(**	AB	Output B

+	/ (**+**	AB	Push +
C	/ (**+**	ABC	Output C
)	**/**	ABC+	Pop until (, write into output except parenthesis
*	*	ABC+ /	ToS is EQP, So pop ToS and push *
D	*	ABC+ / D	Output D
#		ABC+ / D *	Pop *

Output : A B C + / D *

Note : *, / , and % (modulus) are equal in precedence and higher precedence than + and - .
^ is an exponential power operator. ^ has highest precedence than *, / , and %.

BODMAS: Its letters stand for Brackets, Order (meaning powers), Division, Multiplication, Addition, Subtraction.

Eg 2: Convert A*B+(C-D/E)# into postfix notation.

Eg 3: Outline the algorithm to check if the given parenthesized arithmetic expression contains balanced parenthesis and to convert such expression to postfix form and illustrate the steps for the expression A +(B*C-(D/E^F) *G)*H. Evaluate the given postfix expression 9 3 4 * 8 - 4 * +

Input : add # in input expression and write A +(B*C-(D/E^F) *G)*H #

Input Character	**Stack (Note :Top of Stack is bold)**	**Postfix Expression**
A		A
+	**+**	A
(	+ **(**	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
E	+ (- (/	A B C *DE
^	+ (- (/ ^	A B C *DE
F	+ (- (/ ^	A B C *DEF
)	+ (-	A B C *DEF^/
*	+ (- *	A B C *DEF^/
G	+ (- *	A B C *DEF^/G
)	+	A B C *DEF^/G* -
*	+ *	A B C *DEF^/G* -
H	+ *	A B C *DEF^/G* - H
#		A B C *DEF^/G* - H * +

Output:
Resultant Postfix Expression: ABC*DEF^/G*-H*+

Eg 4: Write the procedure to convert the infix expression to postfix expression and steps involved in evaluating the postfix expression. Convert the expression A-(B/C+(D%E*F)/G)*H to postfix form. Evaluate the given postfix expression 9 3 4 * 8 + 4 / -.

Advantage of Postfix Expression over Infix Expression

- An infix expression is difficult for the machine to know and keep track of precedence of operators.
- On the other hand, a postfix expression itself determines the precedence of operators
- Therefore, for the machine it is easier to carry out a postfix expression than an infix expression.

2. Evaluating the Postfix Expression

The postfix expression is evaluated easily by the use of a stack.

Procedure or Algorithm to evaluate the Postfix Expression:

Read the postfix expression one character at a time until it reaches "#" symbol.

Step 1: If the character is an operand, it is pushed onto the stack.
Step 2: If the character is an operator, POP two values from the stack apply the operator to them and push the result onto the stack.
Step 3 : If the character is an #, Then write the result from stack into output.

Exxample: Infix expression A / (B + C) * D its equivalent postfix expression is A B C + / D *, evaluate the postfix expression for A= 20, B = 6, C= 4 , D = 3.

Input: 20 6 4 + / 3*# .

Input Character	Stack (Note :Top of Stack is bold)	Output	Action Taken
20	**20**		Push 2
6	20 **6**		Push 6
4	20 6 **4**		Push 4
+	20 **10**		Pop 6 and 4 , perform +, and push the result
/	**2**		Pop 20and 10 , perform /, and push the result
3	2 **3**		Push 3
*	**6**		Pop 2 and 3 , perform *, and push the result
#		**6**	**Output**

3. Balancing the symbols

Compilers check the programs for errors, a lack of one symbol will cause an error. Every right parenthesis should have its left parenthesis. Check for balancing the parenthesis and ignore any other character.

Procedure to balance the symbols
the Postfix Expression:

Read one character at a time until it encounters the delimiter. "#"
Step 1 : If the Character is an Operand or operator , ignore it.
Step 2: If the character is an '(', push it on to the stack.
Step 3: If the character is a ')' and if the stack is empty, then report an error as missing '('.
Step 4: If the character is a ')' and if it has a corresponding '(' in the stack, then pop it from the stack.
Step 5: If the character is a '#', and if the stack is not empty, report an error as missing closing symbol. Otherwise output **as balanced symbols.**

Example: Let us consider the expression ((a+b) * (d / e)) # Check the expression has

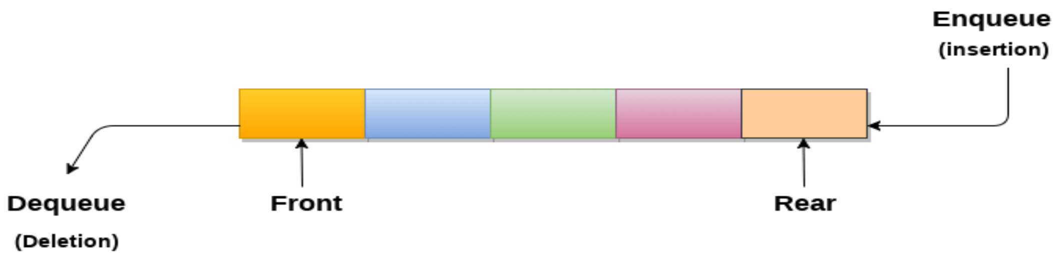
Balanced symbol.

Input : ((a+b) * ((d /e)) #

Input Character	Stack (Note :Top of Stack is bold)	Output	Action Taken
(	**(**		Push (
(	(**(**		Push (
a	(**(**		Ignore a
+	(**(**		Ignore +
b	(**(**		Ignore b
)	**(**		Pop (
*	**(**		Ignore *
(	(**(**		Push (
d	(**(**		Ignore d
/	(**(**		Ignore /
e	(**(**		Ignore e
)	**(**		Pop (
)			Pop (
#		**Balanced Symbol**	**Empty stack, output**

Queue ADT:

- Queue is a Linear Data Structure that follows **First in First out (FIFO)** principle.
- Insertion of element is done at one end of the Queue called **"Rear "**end of the Queue.
- Deletion of element is done at other end of the Queue called **"Front "**end of the Queue.
- Example: - Waiting line in the ticket counter.



Front Pointer:- It always points to the first element inserted in the Queue.

Rear Pointer:- It always points to the last element inserted in the Queue.

For Empty Queue:-

Front (F) = - 1
Rear (R) = - 1

Operations on Queue

Fundamental operations performed on the queue are

1. EnQueue / insert
2. DeQueue / delete
3. Queue Overflow
4. Queue Underflow
5. Display / front

(i) EnQueue operation:-

- It is the process of inserting a new element at the rear end of the Queue.
- For every EnQueue operation
  - o Check for Full Queue
  - o If the Queue is full, Insertion is not possible.
  - o Otherwise, increment the rear end by 1 and then insert the element in the rear end of the Queue.

(ii) DeQueue Operation:-

- It is the process of deleting the element from the front end of the queue.
- For every DeQueue operation

o Check for Empty queue
o If the Queue is Empty, Deletion is not possible.
o Otherwise, delete the first element inserted into the queue and then increment the front by 1.

Exceptional Conditions of Queue

- Queue Overflow
- Queue Underflow

(iii) Queue Overflow: (rear == maxsize-1)

- An Attempt to insert an element X at the Rear end of the Queue when the Queue is full is said to be Queue overflow.
- For every Enqueue operation, we need to check this condition.

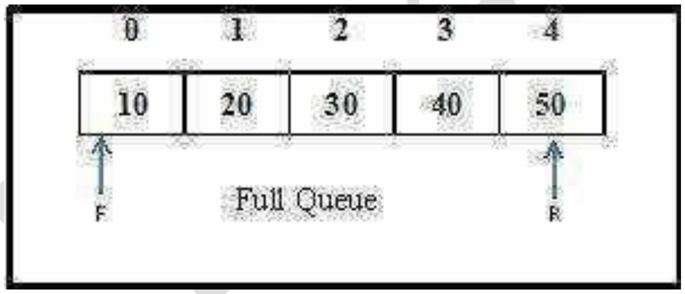
(iv) Queue Underflow: (front == -1 || front > rear)

- An Attempt to delete an element from the Front end of the Queue when the Queue is empty is said to be Queue underflow.
- For every DeQueue operation, we need to check this condition.

Queue can be implemented in two ways.

1. Implementation using Array (Static Queue)
2. Implementation using Linked List (Dynamic Queue)

Implementation using Array (Static Queue)



```
// Enqueue Operation
void enqueue()
{
   int item;
   printf("\nEnter the element\n");
   scanf("%d",&item);
   if(rear == maxsize-1)
   {  printf("OVERFLOW");     return;     }
   if(front == -1 && rear == -1)
   {        front = 0;         rear = 0;      }
   else
```

```
    {         rear = rear+1;      }
    queue[rear] = item;
    printf("Value inserted ");

}

// Dequeue Operation
void dequeue ()
{     int item;
    if (front == -1 || front > rear)
    {         printf("\nUNDERFLOW\n");          return;       }
    else
    {
        item = queue[front];
        if(front == rear)
        {             front = -1;             rear = -1 ;         }
        else
        {             front = front + 1;         }
        printf("\nvalue deleted ");
    }
}
//Display / front Operation

void display()
{    int i;
    if(rear == -1)
    {         printf("\nEmpty queue\n");      }
    else
    {   printf("\nprinting values .....\n");
        for(i=front;i<=rear;i++)
        {             printf("\n%d\n",queue[i]);         }
    }
}

#include<stdio.h>
#include<stdlib.h>
#define maxsize 5
void enqueue ();
void dequeue ();
void display();
```

```
int front = -1, rear = -1;
int queue[maxsize];
void main ()
{
   int choice;
   while(choice != 4)
   {
      printf("\n*************************Main Menu*****************************\n");
      printf("\n1.insert an element\n2.Delete an element\n3.Display the
queue\n4.Exit\n");
      printf("\nEnter your choice ?");
      scanf("%d",&choice);
      switch(choice)
      {
         case 1:  enqueue();  break;
         case 2:  dequeue ();  break;
         case 3:  display(); break;
         case 4:  exit(0);    break;
         default:   printf("\nEnter valid choice??\n");        }
   }
}
```

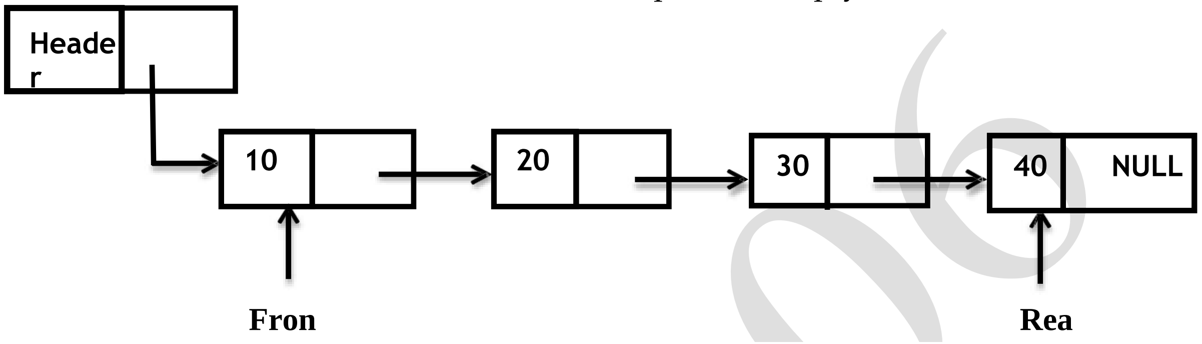
Drawback of array implementation

- **Memory wastage**: The space of the array, which is used to store queue elements, can never be reused to store the elements of that queue because the elements can only be inserted at front end and the value of front might be so high so that, all the space before that, can never be filled.
- **Deciding the array size: T**he extension in the array size is a time taking process and almost impossible to be performed at runtime since a lot of reallocations take place. Due to this reason, the array is declared as large enough, but most of the array slots can never be reused. It will again lead to memory wastage.
- The array implementation of queue cannot be used for the large scale applications.

Linked List implementation of Queue

- In a linked queue, each node of the queue consists of two parts i.e. data part and the link part. Each element of the queue points to its immediate next element in the memory.

- In the linked queue, there are two pointers maintained in the memory i.e. front pointer and rear pointer. The front pointer contains the address of the starting element of the queue while the rear pointer contains the address of the last element of the queue.
- Insertion and deletions are performed at rear and front end respectively. If front and rear both are NULL, it indicates that the queue is empty. The linked



representation of queue is shown in the following figure.

```
#include<stdio.h>
#include<stdlib.h>
struct node
{
   int data;
   struct node *next;
};
struct node *front;
struct node *rear;
void insert();
void delete();
void display();
void main ()
{
   int choice;
   while(choice != 4)
   {printf("\n1.insert an element\n2.Delete an element\n3.Display the queue\n4.Exit\n");
      printf("\nEnter your choice ?");
      scanf("%d",& choice);
      switch(choice)
      {
         case 1:        insert();        break;
```

```
            case 2:         delete();       break;
            case 3:         display();       break;
            case 4:         exit(0);       break;
            default:        printf("\nEnter valid choice??\n");
        }
    }
}

void insert()
{
    struct node *ptr;
    int item;

    ptr = (struct node *) malloc (sizeof(struct node));
    if(ptr == NULL)
    {
        printf("\nOVERFLOW\n");
        return;
    }
    else
    {
        printf("\nEnter value?\n");
        scanf("%d",&item);
        ptr -> data = item;
        if(front == NULL)
        {
            front = ptr;
            rear = ptr;
            front -> next = NULL;
            rear -> next = NULL;
        }
        else
        {
            rear -> next = ptr;
            rear = ptr;
            rear->next = NULL;
        }
    }
}
```

```
void delete ()
{
   struct node *ptr;
   if(front == NULL)
   {        printf("\nUNDERFLOW\n");         return;     }
   else
   {       ptr = front;
      front = front -> next;
      free(ptr);
   }
}


void display()
{
   struct node *ptr;
   ptr = front;
   if(front == NULL)
   {
      printf("\nEmpty queue\n");
   }
   else
   {   printf("\nprinting values .....\n");
      while(ptr != NULL)
      {
         printf("\n%d\n",ptr -> data);
         ptr = ptr -> next;
      }
   }
}
```

Applications of Queue

1. Serving requests on a single shared resource, like a printer, CPU task scheduling etc.
2. In real life, Call Center phone systems will use Queues, to hold people calling them in an order, until a service representative is free.
3. Handling of interrupts in real-time systems. The interrupts are handled in the same order as they arrive, First come first served.
4. Batch processing in operating system.

5. Job scheduling Algorithms like Round Robin Algorithm uses Queue.

Drawbacks of Queue (Linear Queue)

- With the array implementation of Queue, the element can be deleted logically only by moving Front = Front + 1.
- Here the Queue space is not utilized fully.

To overcome the drawback of this linear Queue, we use **Circular Queue**.

Circular Queue

A circular queue uses its storage array as if it were a circle instead of a linear list. In Circular Queue, elements are added at the rear end and the items are deleted at front end of the circular queue. if the last location of the queue is full, then first element comes just after the last element.

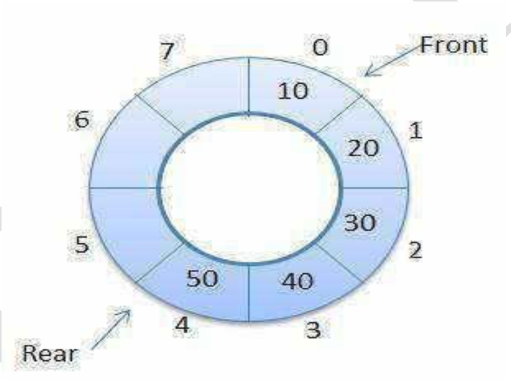


Fig: Circular Queue

Operations on Circular Queue

Fundamental operations performed on the Circular Queue are

- Circular Queue Enqueue
- Circular Queue Dequeue

Formula to be used in Circular Queue

- For Enqueue Rear = (Rear + 1) % ArraySize
- For Dequeue Front = (Front + 1) % ArraySize

ROUTINE TO INSERT AN ELEMENT IN CIRCULAR QUEUE

```
void CEnqueue (int X)
{
```

```
if (front = = (rear + 1) % Maxsize)
print ("Queue is overflow");
else
{ if (front = = -1)
front = rear = 0;
else rear = (rear + 1)% Maxsize;
CQueue [rear] = X; }
 }
```

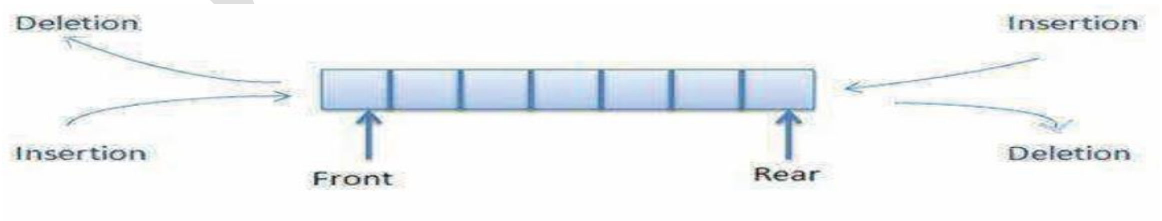
ROUTINE TO DELETE AN ELEMENT FROM CIRCULAR QUEUE

```
int CDequeue ( )
{ if (front = = -1)
print ("Queue is underflow");
else { X = CQueue [Front];
if (front = = rear)
     front = rear = -1;
else
front = (front + 1)% maxsize;
}
return (X);
}
```

ADVANTAGES
It is a structure that allows data to be passed from one process to another while making the most efficient use of memory.

Double Ended Queues (DEQUE ADT)

In DEQUE, insertion and deletion operations are performed at both ends of the Queue.



Exceptional Condition of DEQUE
(i) Input Restricted DEQUE
Here **insertion is allowed at one end** and deletion is allowed at both ends.

(ii) **Output Restricted DEQUE**
Here insertion is allowed at both ends and **deletion is allowed at one end**.

Operations on DEQUE
Four cases for inserting and deleting the elements in DEQUE are
1. Insertion At Rear End [same as Linear Queue]
2. Insertion At Front End
3. Deletion At Front End [same as Linear Queue]
4. Deletion At Rear End

/* The following program in C, implements the logic of Double ended queue, in which the insertion & deletion from end as well as starting is allowed(circular array) */

```
#include<stdio.h>
#include<stdlib.h>
#include<conio.h>
 #define SIZE 100 int queue[SIZE];
 int F = -1; int R = -1;
```
/* To insert element in the rear of Double ended queue*/
```
void insert_r(int x)
 { if(F == (R+1)%SIZE)
{ printf("\nQueue Overflow"); }
else if(R == -1)
{ F = 0; R = 0; queue[R] = x; }
 else
{ R = (R+1) %SIZE; queue[R] = x; } }
```

/* To insert element in the front of Double ended queue*/

```
void insert_f(int x)
{ if(F == (R+1)%SIZE)
{ printf("\nQueue Overflow"); }
else if(R == -1) { F = 0; R = 0;
queue[R] = x; }
else { F = (F+SIZE-1) %SIZE; queue[F] = x; }
 }
```

/* To delete element in the rear of Double ended queue*/
```
int delete_r()
{ int x; if(F == -1)
{ printf("\nQueue Underflow"); }
else if(F == R)
{ x = queue[F];
F = -1; R = -1; }
else
{ x = queue[R];
R = (R+SIZE-1)%SIZE;
}
return x; }
```

/* To delete element in the front of Double ended queue*/

```
 int delete_f()
{ int x; if(F == -1)
{ printf("\nQueue Underflow"); }
 else if(F == R)
{ x = queue[F];
F = -1; R = -1; }
else { x = queue[F];
F = (F+1)%SIZE;
}
return x;
}

void main()
{ char choice;     int x;
while(1)
{ printf("1: Insert From Front\n");
 printf("2: Insert From Rear\n");
 printf("3: Delete From Front\n");
printf("4: Delete From Rear\n");
printf("5: Exit Program\n");
printf("Enter Your Choice:");
choice = getche();
switch(choice) { case '1': printf("\nEnter Integer Data :");
                                    scanf("%d",&x); insert_f(x); break;
                       case '2': printf("\nEnter Integer Data :");
                                   scanf("%d",&x); insert_r(x); break;
                      case '3': printf("\nDeleted Data From Front End: %d",delete_f()); break;
                      case '4': printf("\nDeleted Data From Back End: %d",delete_r()); break;
                      case '5': exit(0); break; } }}
```

Priority Queue (Heap or Binary Heap)

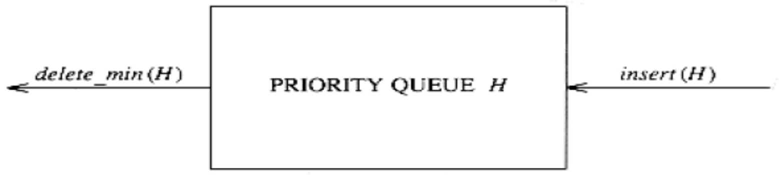
Need for priority queue

- The operating system uses Queue Data structures to schedule the jobs to the CPU.
- Jobs are initially placed at the end of the queue.
- The scheduler will repeatedly take the first job on the queue, run it until either it finishes or its time limit is up
- Sometime, some jobs are still very important have precedence over others and need to be executed first.
- This particular application seems to require a special kind of queue, known as priority queue. Priority queue is also called as Heap or Binary Heap.

Definition:

A priority queue is a data structure that allows two operations:

- insert (enqueue operation) and
- delete_min (queue's dequeue operation with priroity), which finds, returns and removes the minimum element in the heap.



Implementation of priority Queue : Binary Heap implementation

(Refer Unit3 Notes)