UNIT 5

STACKS AND QUEUES

STACKS

- Stacks are a specific kind of list.
- A stack is simply a list of elements with insertion and deletion permitted at one end called as stack top.
- A **stack** is a temporary abstract data type and data structure based on the principle of Last In First Out (LIFO).
- Stacks are used extensively at every level of a modern computer system.

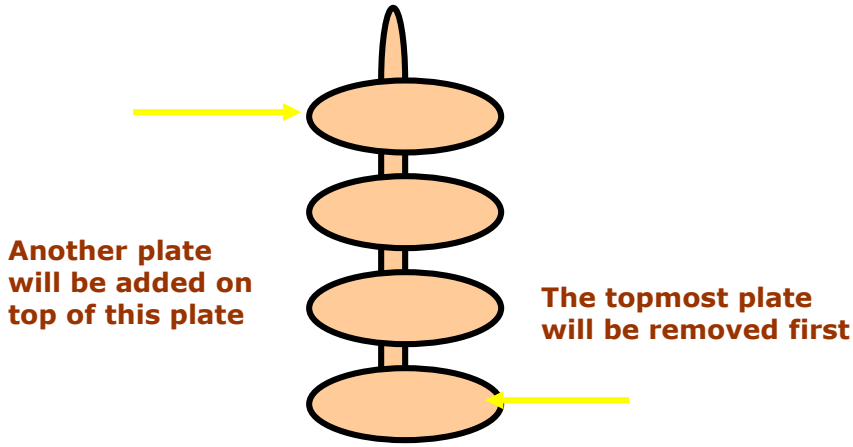
Cont…

- A modern PC uses stacks at the architecture level.
- Used in the basic design of an operating system for interrupt handling and operating system function calls.
- The stack is ubiquitous.
- Stacks have specific adds and removes called

  Push and pop.

INTRODUCTION TO STACKS

- Stack is an important data structure which stores its elements in an ordered manner. Take an analogy of a pile of plates where one plate is placed on top of the other. A plate can be removed from the topmost position. Hence, you can add and remove the plate only at/from one position that is, the topmost position.



Same is the case with stack. A stack is a linear data structure which can be implemented either using an array or a linked list. The elements in a stack are added and removed only from one end, which is called *top*. Hence, a stack is called a LIFO (Last In First Out) data structure as the element that was inserted last is the first one to be taken out.

ARRAY REPRESENTATION OF STACKS

•In computer's memory stacks can be represented as a linear array.

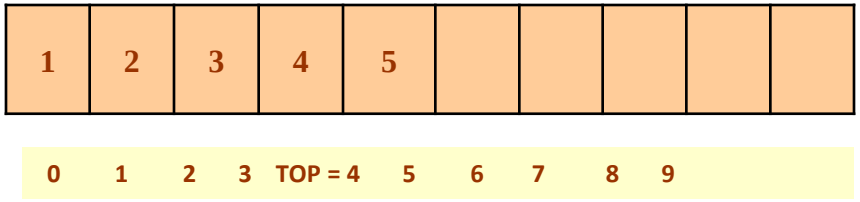
• Every stack has a variable TOP associated with it. TOP is used to store the address of the topmost element of the stack. It is this position from where the element will be added or deleted.

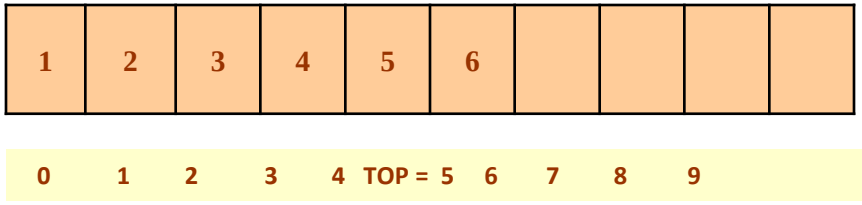
•There is another variable MAX which will be used to store the maximum number of elements that the stack can hold.

•If TOP = NULL, then it indicates that the stack is empty and if TOP = MAX -1, then the stack is full.

Push Operation

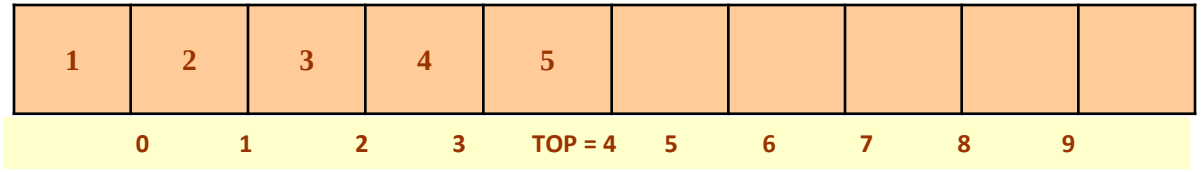
- The push operation is used to insert an element in to the stack. The new element is added at the topmost position of the stack. However, before inserting the value, we must first check if TOP=MAX-1, because if this is the case then it means the stack is full and no more insertions can further be done. If an attempt is made to insert a value in a stack that is already full, an OVERFLOW message is printed.

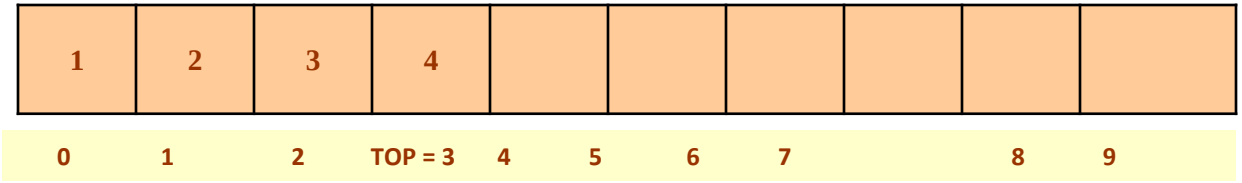




POP OPERATION

- The pop operation is used to delete the topmost element from the stack. However, before deleting the value, we must first check if TOP=NULL, because if this is the case then it means the stack is empty so no more deletions can further be done. If an attempt is made to delete a value from a stack that is already empty, an UNDERFLOW message is printed.





PEEP OPERATION

- Peep is an operation that returns the value of the topmost element of the stack without deleting it from the stack.
- However, the peep operation first checks if the stack is empty or contains some elements. For this, a condition is checked. If TOP = NULL, then an appropriate message is printed else the value is returned.

```
Algorithm to PUSH an element in to the stack

Step 1: IF TOP = MAX-1, then
                    PRINT "OVERFLOW"
          [END OF IF]
Step 2: SET TOP = TOP + 1
Step 3: SET STACK[TOP] = VALUE
Step 4: END
```

```
Algorithm to POP an element from the stack

Step 1: IF TOP = NULL, then
                    PRINT "UNDERFLOW"
          [END OF IF]
Step 2: SET VAL = STACK[TOP]
Step 3: SET TOP = TOP - 1
Step 4: END
```

```
Algorithm for Peep Operation

Step 1: IF TOP =NULL, then
          PRINT "STACK IS EMPTY"
                    Go TO Step 3
Step 2: RETURN STACK[TOP]
Step 3: END
```

PUSH() FUNCTION

```
void push()
{
   if(top>=n-1)
   {
      printf("\n\tSTACK is over flow");

   }
   else
   {
      printf(" Enter a value to be pushed:");
      scanf("%d",&x);
      top++;
      stack[top]=x;
   }
}
```

POP() FUNCTION

```
void  pop()
{
   if(top<=-1)
   {
      printf("\n\t Stack is under flow");
}
   else
   {
      printf("\n\t The popped elements is %d",stack[top]);
      top--;
}
}
```

TRAVERSE AND DISPLAY STACK DATA

```
void display()
{
   if(top>=0)
   {
      printf("\n The elements in STACK \n");
      for(i=top; i>=0; i--)
         printf("\n%d",stack[i]);

   }
   else
   {
      printf("\n The STACK is empty");
   }
```

PROGRAM TO PERFORM STACK OPERATIONS:PUSH, POP, TRAVERSE

```
#include<stdio.h>
int stack[100],choice,n,top,x,i;
void push(void);
void pop(void);
void display(void);
int main()
{
   //clrscr();
   top=-1;
   printf("\n Enter the size of STACK[MAX=100]:");
   scanf("%d",&n);
   printf("\n\t STACK OPERATIONS USING
   ARRAY");
   printf("\n\t--------------------------------");
   printf("\n\t 1.PUSH\n\t 2.POP\n\t
   3.DISPLAY\n\t 4.EXIT");
do
   {
      printf("\n Enter the Choice:");
      scanf("%d",&choice);
      switch(choice)
      {
         case 1: {
            push(); break;   }
         case 2:  {
            pop();    break; }
         case 3:    {
            display(); break;     }
         case 4:  {
            printf("\n\t EXIT POINT ");  break;   }
         default:
         {
            printf ("\n\t not a Valid Choice”);
         }
                  }
   }
   while(choice!=4);
   return 0;
}
```

Stack Applications

Evaluating arithmetic expressions.

- Prefix: + a b
- Infix: a +
- Postfix: a b +

INFIX, POSTFIX AND PREFIX NOTATION

- Infix, Postfix and Prefix notations are three different but equivalent notations of writing algebraic expressions.
- While writing an arithmetic expression using infix notation, the operator is placed in between the operands. For example, A+B;
- Although for us it is easy to write expressions using infix notation but computers find it difficult to parse as the computer needs a lot of information to evaluate the expression. Information is needed about operator precedence, associativity rules and brackets which overrides these rules. So, computers work more efficiently with expressions written using prefix and postfix notations.

POSTFIX NOTATION

- Postfix notation was given by Jan Łukasiewicz who was a Polish logician, mathematician, and philosopher. His aim was to develop a parenthesis-free prefix notation (also known as Polish notation) and a postfix notation which is better known as Reverse Polish Notation or RPN.
- In postfix notation, as the name suggests, the operator is placed after the operands. For example, if an expression is written as A+B in infix notation, the same expression can be written AB+ in postfix notation. The order of evaluation of a postfix expression is always from left to right. Even brackets can not alter the order of evaluation.
- Similarly, the expression- (A + B) * C is written as –
- [AB+]*C
- AB+C* in the postfix notation.
- A postfix operation does not even follow the rules of operator precedence. The operator which occurs first in the expression is operated first on the operands. For example, given a postfix notation AB+C*. While evaluation, addition will be performed prior to multiplication.

PREFIX NOTATION

- Although a Prefix notation is also evaluated from left to right but the only difference between a postfix notation and a prefix notation is that in a prefix notation, the operator is placed before the operands. For example, if A+B is an expression in infix notation, then the corresponding expression in prefix notation is given by +AB.
- While evaluating a prefix expression, the operators are applied to the operands that are present immediately on the right of the operator. Like postfix, prefix expressions also do not follow the rules of operator precedence, associativity and even brackets cannot alter the order of evaluation.

Convert the following infix expressions into prefix expressions

- (A + B) * C
- (+AB)*C
- *+ABC

EVALUATION OF AN INFIX EXPRESSION

- STEP 1: Convert the infix expression into its equivalent postfix expression

```
Algorithm to convert an Infix notation into postfix notation

Step 1: Add ')" to the end of the infix expression
Step 2: Push "(" on to the stack
Step 3: Repeat until each character in the infix notation is scanned
          IF a "(" is encountered, push it on the stack
          IF an operand (whether a digit or an alphabet) is encountered, add it to
     the postfix expression.
          IF a ")" is encountered, then;
          aRepeatedly pop from stack and add it to the postfix expression until a
          "(" is encountered.
bDiscard the "(". That is, remove the "(" from stack and do not add it to the
postfix expression
     IF an operator O is encountered, then;
          aRepeatedly pop from stack and add each operator (popped from the stack)
          to the postfix expression until it has the same precedence or a higher
          precedence than O
bPush the operator O to the stack
     [END OF IF]
     Step 4: Repeatedly pop from the stack and add it to the postfix expression
     until the stack is empty
Step 5: EXIT
```

Exercise: Convert the following infix expression into postfix expression using the algorithm given in figure 9.21.

A – (B / C + (D % E * F) / G)* H

A – (B / C + (D % E * F) / G)* H)

Infix Character Scanned	STACK	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 / D E
*	(- (+ (% *	A B C / D E
F	(- (+ (% *	A B C / D E F
)	(- (+	A B C / D E F * %
/	(- (+ /	A B C / D E F * %
G	(- (+ /	A B C / D E F * % G
)	(-	A B C / D E F * % G / +
*	(- *	A B C / D E F * % G / +
H	(- *	A B C / D E F * % G / + H
)		A B C / D E F * % G / + H * -

STEP 2: Evaluate the postfix expression

```
Algorithm to evaluate a postfix expression

Step 1: Add a ")" at the end of the postfix expression
Step 2: Scan every character of the postfix expression and
repeat steps 3 and 4 until ")"is encountered
Step 3: IF an operand is encountered, push it on the stack
IF an operator O is encountered, then
         pop the top two elements from the stack as A and B
         Evaluate B O A, where A was the topmost element and B
was the element below A.
        Push the result of evaluation on the stack
       [END OF IF]
Step 4: SET RESULT equal to the topmost element of the stack
Step 5: EXIT
```

Let us now take an example that makes use of this algorithm. Consider the infix expression given as "9 - ((3 * 4) + 8) / 4". Evaluate the expression.

The infix expression "9 - ((3 * 4) + 8) / 4" can be written as "9 3 4 * 8 + 4 / -" using postfix notation. Look at table I which shows the procedure.

Character scanned	Stack
9	9
3	9, 3
4	9, 3, 4
*	9, 12
8	9, 12, 8
+	9, 20
4	9, 20, 4
/	9, 5
-	4

CONVERT INFIX EXPRESSION TO PREFIX EXPRESSION

```
Step 1: Reverse the infix string. Note that while reversing the
string you must interchange left and right parenthesis.
Step2: Obtain the corresponding postfix expression of the infix
expression obtained as a result of Step1.
Step 3: Reverse the postfix expression to get the prefix
expression
```

For example, given an infix expression- (A – B / C) * (A / K – L)

Step 1: Reverse the infix string. Note that while reversing the string you must interchange left and right parenthesis.

(L – K / A) * (C / B – A)

Step2: Obtain the corresponding postfix expression of the infix expression obtained as a result of Step1.

The expression is: (L – K / A) * (C / B – A)

Therefore, [L – (K A /)] * [(C B /) - A]

= [LKA/-] * [CB/A-]

= L K A / - C B / A - *

Step 3: Reverse the postfix expression to get the prefix expression

Therefore, the prefix expression is * - A / B C - / A K L

Linked Implementation of Stack – push()

- **Step 1 -** Define a '**Node**' structure with two members **data** and **next**.
- **Step 2 -** Define a **Node** pointer '**top**' and set it to **NULL**.
- **Step 3 -** Implement the **main** method by displaying Menu with list of operations and make suitable function calls in the **main** method.

Linked Implementation of Stack- push()

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

Linked Implementation of Stack- pop()

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**'. Temp=top
- **Step 4 -** Then set '**top** = **top → next**'.
- **Step 5 -** Finally, delete '**temp**'. (**free(temp)**).

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**'.

Implementation of Stack using Linked List in C

```
#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 *;
   = (struct
Node*)malloc(sizeof(struct Node));
  ->data = value;
  if(top == NULL)
    ->next = NULL;
  else
    ->next = top;
  top = ;
  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);
  }
}
```

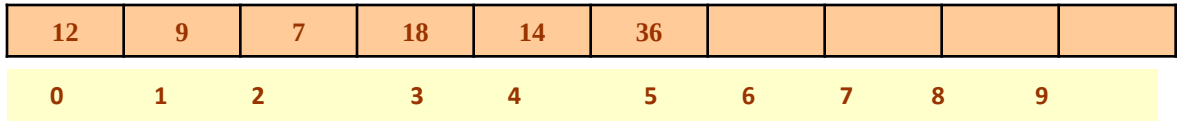
QUEUES

- Queue is an important data structure which stores its elements in an ordered manner. Take for example the analogies given below.
- People moving on an escalator. The people who got on the escalator first will be the first one to step out of it.
- People waiting for bus. The first person standing in the line will be the first one to get into the bus.

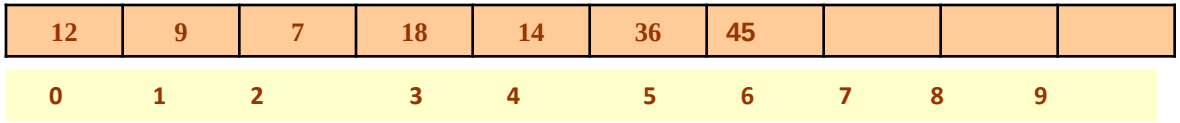
- A queue is a FIFO (First In First Out) data structure in which the element that was inserted first is the first one to be taken out. The elements in a queue are added at one end called the rear and removed from the other one end called front.

ARRAY REPRESENTATION OF QUEUE

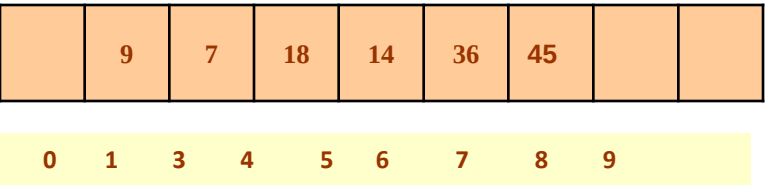
- Queues can be easily represented using linear arrays. As stated earlier, every queue will have front and rear variables that will point to the position from where deletions and insertions can be done respectively.



Here, front = 0 and rear = 5. If we want to add one more value in the list say with value 45, then rear would be incremented by 1 and the value would be stored at the position pointed by rear.



- Here, front = 0 and rear = 6. Now, if we want to delete an element from the queue, then the value of front will be incremented. Deletions are done from only this end of the queue.



However, before inserting an element in the queue we must check for overflow conditions. An overflow will occur when we will try to insert an element into a queue that is already full. When Rear = MAX – 1, where MAX specifies the maximum number of elements that the queue can hold.

Similarly, before deleting an element from the queue, we must check for underflow condition. An underflow condition occurs when we try to delete an element from a queue that is already empty. If front = -1 and rear = -1, this means there is no element in the queue.

QUEUE OPERATIONS

Enqueue: Adds an item to the queue. If the queue is full, then it is said to be an Overflow condition.

Dequeue: Removes an item from the queue. The items are popped in the same order in which they are pushed. If the queue is empty, then it is said to be an Underflow condition.

Front: Get the front item from queue.

Rear: Get the last item from queue.

ALGORITHMS TO INSERT AND DELETE AN ELEMENT FROM THE QUEUE

```
Algorithm to insert an element in the queue

Step 1: IF REAR=MAX-1, then;
                    Write OVERFLOW
          [END OF IF]
Step 2: IF FRONT == -1 and REAR = -1, then;
                    SET FRONT = REAR = 0
          ELSE
                    SET REAR = REAR + 1
          [END OF IF]
Step 3: SET QUEUE[REAR] = NUM
Step 4: Exit
```

```
Algorithm to delete an element from the queue

Step 1: IF FRONT = -1 OR FRONT > REAR, then;
                   Write UNDERFLOW
          ELSE
                   SET FRONT = FRONT + 1
                   SET VAL = QUEUE[FRONT]
          [END OF IF]
Step 2: Exit
```

C PROGRAM FOR QUEUE OPERATIONS

```
#include<stdio.h>
#include<conio.h>
#define SIZE 10
void enQueue(int);
void deQueue();
void display();
int queue[SIZE], front = -1,
rear = -1;
```

```
void main()
{
  int value, choice;
  clrscr();
  while(1){
     printf("1. Insertion\n2. Deletion\n3. Display\n4. Exit");
    printf("\nEnter your choice: ");
    scanf("%d",&choice);
    switch(choice){
            case 1: printf("Enter the value to be insert: ");
                    scanf("%d",&value);
                    enQueue(value);
                    break;
            case 2: deQueue();
                    break;
            case 3: display();
                    break;
            case 4: exit(0);
            default: printf("\nWrong selection!!! Try again!!!");
    }
  }
}
```

C PROGRAM FOR QUEUE OPERATIONS

```
void enQueue(int value){
   if(rear == SIZE-1)
      printf("\nQueue is Full!!! Insertion is not possible!!!");
   else{
      if(front == -1)
            front = 0;
      rear++;
      queue[rear] = value;
      printf("\nInsertion success!!!");
   }
}
```

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

C PROGRAM FOR QUEUE OPERATIONS

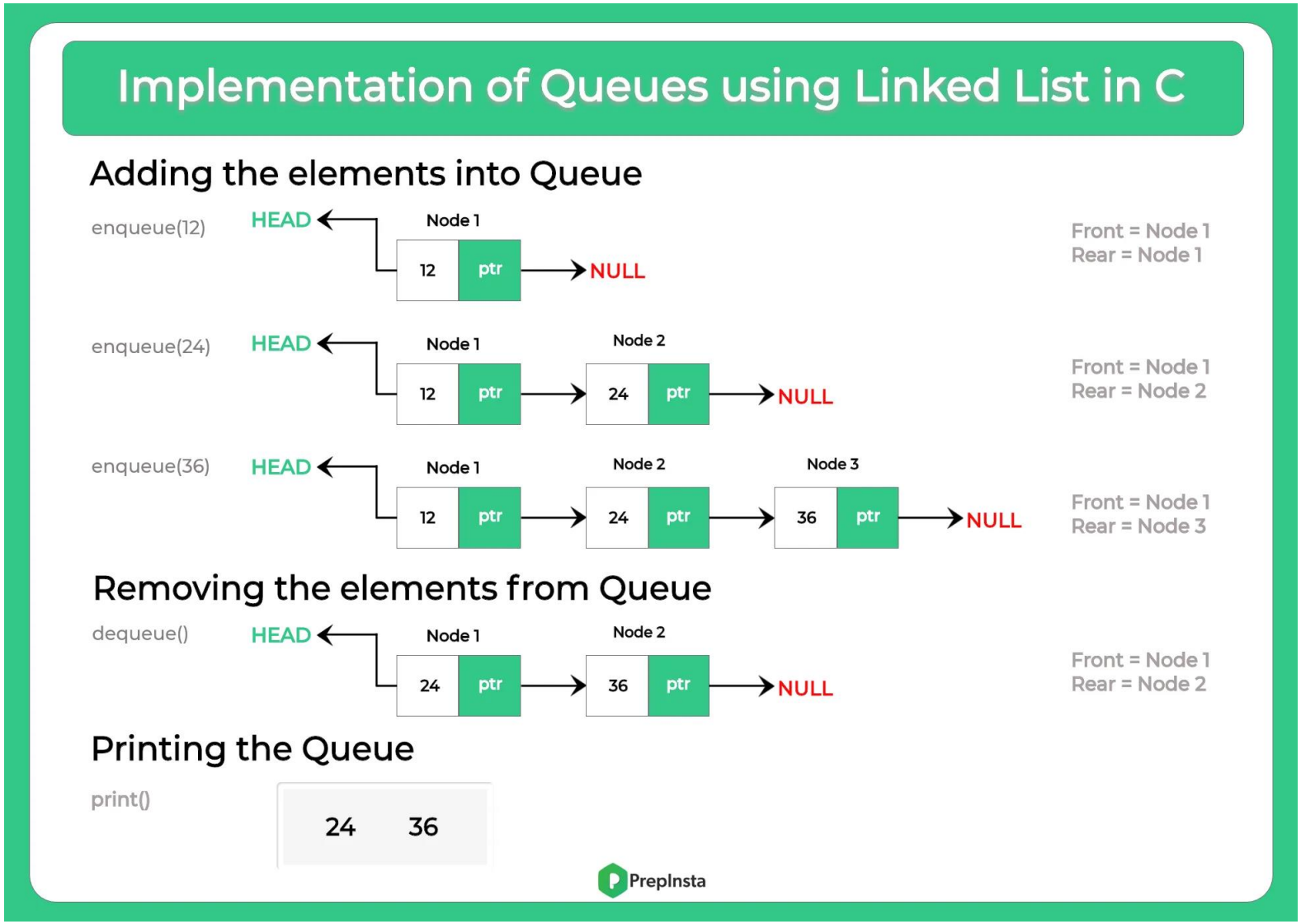
```
void display(){
  if(rear == -1)
    printf("\nQueue is Empty!!!");
  else{
    int i;
    printf("\nQueue elements are:\n");
    for(i=front; i<=rear; i++)
        printf("%d\t",queue[i]);
  }
}
```

Applications of Queues

Queue is used when things have to be processed in First In First Out order like Breadth First Search. This property of Queue makes it also useful in following kind of scenarios.

1) When a resource is shared among multiple consumers. Examples include CPU scheduling, Disk Scheduling.
2) When data is transferred asynchronously (data not necessarily received at same rate as sent) between two processes. Examples include IO Buffers, pipes, file IO, etc.
3) In Operating systems:
    a) Semaphores
    b) FCFS (first come first serve) scheduling, example: FIFO queue
    c) Spooling in printers
    d) Buffer for devices like keyboard
4) In Networks:
    a) Queues in routers/ switches
    b) Mail Queues

Linked implementation of Queue

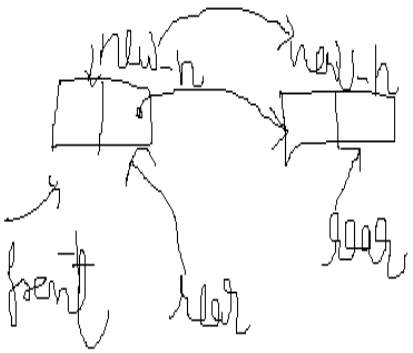


Add node in a queue

enqueue(int d)

{

```
struct node* new_n;
new_n->data = d;
new_n->next = null;
if((front == null)&&(rear == null))
front = rear = new_n;
else
rear->next = new_n
rear = new_n
}
```



Delete node from a queue

```
dequeue()
{
struct node *temp;
temp = front;
if((front == null)&&(rear == null))
return;
else
front = front->next;
free(temp);
}
```

Traverse a Queue

```
print()
{
struct node* temp;
if((front == null)&&(rear == null))
return;
else
temp = front;
while(temp)
{
printf("%d", temp->data);
temp = temp->next
}
}
```

```
#include <stdio.h>
#include <stdlib.h>
struct node
{
        int data;
        struct node* next;
};//defining linked list to implement queue
struct node *front = NULL;
struct node *rear = NULL;
int main()//main function to use all our declared function
{
        int opt,n,i,data;
        printf("Enter Your Choice:-");
        while(opt!=0){
printf("\n\1.Insert Queue\n2.show  Queue     \n3. Delete from the Queue\n 0 for Exit");
                scanf("%d",&opt);
                switch(opt){
                        case 1:
                                printf("\nEnter the size: "); scanf("%d",&n);
                                printf("\nEnter your data\n");
                                i=0;
                                while(i<n){
                                        scanf("%d",&data);
                                        enqueue(data);
                                        i++;
                                }
                                break;
                        case 2:
                                print();
                                break;
                        case 3:
                                dequeue();
                                break;
                        case 0:
                                break;
                        default:
                                printf("\nIncorrect Choice");

                }
        }
}
```

```
#include <stdio.h>
#include <stdlib.h>
struct node
{
        int data;
        struct node* next;
};//defining linked list to implement queue
struct node *front = NULL;
struct node *rear = NULL;
void enqueue(int d)//function to insert a node in queue
{
        struct node* new_n;
        new_n = (struct node*)malloc(sizeof(struct node));
        new_n->data = d;
        new_n->next = NULL;
        if((front == NULL)&&(rear == NULL)){
                front = rear = new_n;
        }
        else{
                rear->next = new_n;
                rear = new_n;
        }

}
```

```
void print()//function to display the queue
{
        struct node* temp;
        if((front == NULL)&&(rear == NULL)){
                printf("\nQueue is Empty");
        }
        else{
                temp = front;
                while(temp){
                        printf("\n%d",temp->data);
                        temp = temp->next;
                }
        }
}
```

```
void dequeue()//function to delete an element from a queue
{
        struct node *temp;
        temp = front;
        if((front == NULL)&&(rear == NULL)){
                printf("\nQueue is Empty");
        }
        else{
                front = front->next;
                free(temp);
        }
}
```