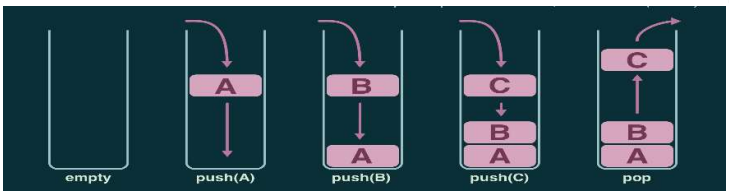
Stack

- It is a linear data structure that follows a particular order in which the operations are performed.
- The order may be LIFO(Last In First Out) or FILO(First In Last Out).
- The insertion and deletion operation in stack are known as PUSH and POP operations.
- Push and pop done at the top of the stack .
- operation can be performed on stack :
  - push () : insert item in stack
  - pop() : delete top item in stack
  - peek() : access the top item of stack
- All operation done in constant time O(1) time complexity



Implementation of stack using array

```
#include <stdio.h>
#define MAX 100
int stack[MAX],top = -1;
void push(int val)
{
   if (top == MAX)
      printf("\n Overflow");
      return;
   top = top + 1;
   stack[top] = val;
}
int peek(){
   return stack[top];
}
void pop()
{
   if (top == -1)
      printf("Underflow");
   else
      top = top - 1;
}
void show()
{
   for (int i = top; i >= 0; i--)
      printf("%d\n", stack[i]);
   if (top == -1)
      printf("Stack is empty");
}
void main()
{
   push(4); // insert the item
   push(2); // insert the item
   show(); // show the items
   pop();  // insert the item
   show();  //show the items of the stack
  int val =peek(); // get the top value
 printf("top element %d " ,val); // print value

}
```

Implementation of stack using Linked List

```
#include<stdio.h>
#include<stdlib.h>
struct Stack
{
   int data;
   struct Stack* next;
};
struct Stack* Node(int data){
  struct Stack * newNode=(struct Stack* ) malloc(sizeof(struct Stack));
   newNode->data=data;
   newNode->next=NULL;
   return newNode;
}
struct Stack* push(struct Stack * top , int data){
   struct Stack* newNode=Node(data);
    if(top!=NULL){
       newNode->next=top;
    }
    return newNode;
 }
 int peek(struct Stack * top){
    return top->data;
}
struct Stack * pop(struct Stack * top ){
    return top->next;
}
void main(){
   struct Stack * top=NULL;
   top=push(top,5);
   top=push(top,15);
   int val=peek(top); // 15
   top=pop(top);
   int val=peek(top); // 5
   top=push(top,3);
   val=peek(3); // 3
}
```

Algorithm of push or pop operation

```
PUSH(stack, data): //algo of push() operation
1. top ==Max-1 then print "stack overflow " stop
2. top increment by 1
3. stack[top]=data
4. stop

POP(stack): //algo of pop() operation
1. top<0 then print "stack underflow " stop
2. top decrement by 1
3. stop
```

Application of Stack

- Back and forward buttons in a web browser:
- Undo/redo functionality in text editors a
- Expression conversion (postfix , infix)
- Parenthesis checking
- String reversal
- **Infix and postfix notation**

  (a+b) * c -- infix notation
  ab+c* -- postfix notation

Conversion of Infix to postfix using stack

Infix notation : A + (B*C – (D/E ^ F)*H)

Expression	stack	postfix
A		A
+	+	A
(	+(	A
B	+(	AB
*	+(*	AB
C	+(*	ABC
-	+(-	ABC*
(	+(-(	ABC*
D	+(-(	ABC*D
/	+(-(/	ABC*D
E	+(-(/	ABC*DE
^	+(-(/^	ABC*DE
F	+(-(/^	ABC*DEF
)	+(-	ABC*DEF^/
*	+(-*	ABC*DEF^/
H	+(-*	ABC*DEF^/H
)	+	ABC*DEF^/H*-
		ABC*DEF^/H*-+

Infix to postfix using arithmetic expression

Infix notation : A + (B * C + D)/E.

A + (B * C + D)/E	T1=BC*
A + (T1+ D)/E	T2=T1D+
A+T2/E	T3=T2E/
A+T3	T4=AT3+
T4	PUT T4
AT3+	PUT T3
AT2E/+	PUT T2
AT1D+E/+	PUT T1
ABC*D+E/+	

postfix to infix Notation

- postfix notation : 752+*

7	7
5	7,5
2	7,5,2
+	7,(5+2)
*	7*(5+2)

Iteration

- Iteration is when same procedure is repeated multiple times
- Each repetition of process is a single iteration
- Result of each iteration is starting point of next iteration.
- Iteration allows us to simplify our algorithm .
- Iteration done by using loop of the languages
- Example : factorial , fibonocci , sum of array etc

```
int arr[5]={1,2,3,3,4} , sum =0 ;
for (int i = 0; i < 5; i++)
{
sum+=arr[i];
}
printf("%d",sum);
```

Recursion

- Recursion is the technique of making a function call itself.
- This provides to break problems into sum problems which are easier to solve.
- Recursion may be a bit difficult to understand.
- A simple ***base case*** **(or cases)** — it tells the function when to stop. if we fail to include this condition it will result in infinite recursions.
- A *recursive step* — a set of rules that reduces problems to subproblems.
- Example:

```
int recursion(int n){
    if(n==0) return 1; //base case
    return n*recursion(n-1); //recursive step
}
```

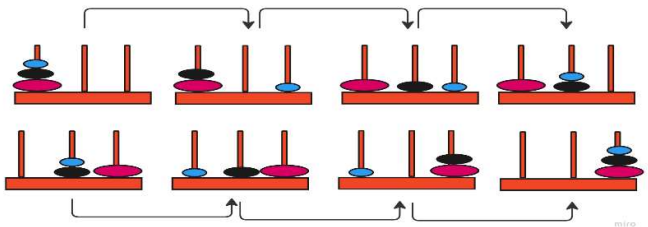
Types of Recursion

- **Direct recursion:** A function is directly recursive if it contains an explicit call to itself.
- **Indirect recursion:** A function is indirectly recursive if it contains a call to another function
- **Tail recursion:** is defined as a recursive function in which the recursive call is the last statement that is executed by the function.
- **Tree recursion :** In which we call multiple recursive call like fibo(n) =fibo(n-1) + fibo(n-2)
- **Example of type of recursion**

int foo(int n){
if(n==0)
return 1;
return foo(n-1);
}
	int foo (int x)
	{
	if (x <= 0)
	return x;
	return bar (x) ;
	}
	int bar (int y)
	{ return foo (y – 1) ; }
		int recursion(int n){
		if(n==0) return 1;
		return n*recursion(n-1);
		}

direct recursion	indirect recursion	tail recursion

Tower of Hnoi



- Tower of Hanoi is a mathematical puzzle where we have three rods (**A**, **B**, and **C**) and **N** disks. Initially, all the disks are stacked in decreasing value of diameter
- Only one disk can be moved at a time.
- Each move consists of taking the upper disk from one of the stacks
- No disk may be placed on top of a smaller disk.
- Total no. of steps to solve of n disk = 2n – 1 = 2*3 – 1 = 7

Algorithm of Tower of Hnoi

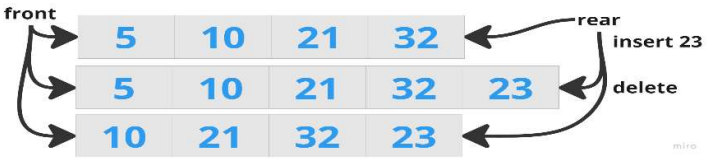
```
void TOH(n , s , a , d):
1.  if n==0
2.    return
3.  TOH(n-1,s,d,a)  //recursive call
4.  print(s+"to"+d)
5.  TOH(n-1,a,s,d)  // recursive call
```

Tower of Hnoi program in C

```c
#include<stdio.h>
void towers( int num, char  S, char A, char D)
{
    if (num == 0)
      return;
    towers (num - 1, S,D , A);
    printf ("\n Move disk %d from peg %c to peg %c", num, S, D);
    towers (num - 1, A, S, D);
}
int main()
{
  int num;
  printf ("Enter the number of disks : ");
  scanf ("%d", &num);
  printf ("The sequence of moves :\n");
  towers (num, 'A', 'B', 'C');
  return 0;
}
```

Queue

- A *Queue* is defined as a linear data structure
- *Queue* uses two pointers − front and rear.
- Deletion done using front pointer.insertion done using rear pointer.
- Queue follows the **First In First Out (FIFO) rule .**
- all operation of done at constant O(1) time
- operation can be performed on queue :
  - insert () : insert item in queue
  - delete () : delete top item in queue



Implementation of queue using array

```c
#include <stdio.h>
#include<stdlib.h>
# define SIZE 100
int queue[SIZE];
int Rear = - 1,Front=-1;
void insert(int data)
{
  if (Rear == SIZE - 1){
    printf("Overflow \n");
    return ;
  }
    if (Front == - 1){
       Front = 0;
    }
    queue[++Rear] = data;
}
void delete ()
{
  if(Front==Rear){
    Front=Rear=-1;
  }
  if (Front == - 1 )
  {
    printf("Underflow \n");
    return ;
  }
  Front = Front + 1;
}
void show()
{
  if (Front == - 1){
    printf("Empty Queue \n");
    return ;
  }
  for (int i = Front; i <= Rear; i++){
    printf("%d ", queue[i]);
  }
}
int main()
{
   show();  // show the items of the queue
  insert(4); // insert the item on the top of queue
  insert(2); // insert the item on the top of queue
  show(); // show the items of the queue
  delete();  // insert the item on the top of queue
  show();  //show the items of the queue
}
```

Implementation of queue using Linked List

```c
#include<stdio.h>
#include <stdlib.h>
struct Queue {
  int data;
  struct Queue* next;
};
struct Queue* front = NULL;
struct Queue* rear = NULL;

void insert(int data) {
  struct Queue* newQueue = (struct Queue*)malloc(sizeof(struct Queue));
  newQueue->data = data;
  newQueue->next = NULL;
  if (front == NULL && rear == NULL) {
    front = rear = newQueue;
    return;
  }
  rear->next = newQueue;
  rear = newQueue;
}

int delete() {
  if (front == NULL) {
    printf("Queue is empty");
    return -1;
  }
  int data = front->data;
  if (front == rear)
    front = rear = NULL;
  else
    front = front->next;

  return data;
}

void  main() {
  insert(10);
  insert(20);
  printf("%d ", delete());
  printf("%d ", delete());
  printf("%d ", delete());
}
```

Algorithm of insert & delete operation in queue

```
function insert(data , queue,rear ,front ,size ):
  1. if rear==size -1 then print "queue overflow" stop
  2. else
  3.  check if front ==-1 then set front =0
  4. set rear=rear+1 and queue[rear]=data
  5. endif
```

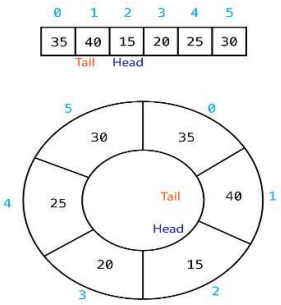
```
function delete ( queue,rear ,front  ):
1.if rear=front then set front=rear=1  endif
2. if front==-1 then print "queue underflow " stop endif
3.set front =front +1
```

Application of Queue

- Add a song into playlist
- Printers
- Used in graph traversal bfs algorithm
- Ticket windows
- Bus stop

Circular queue

- A Circular Queue is an extended version of a normal queue
- Last element is connected to the first element of the queue forming a circle.
- new element is done at the very first location of the queue if the last location at the queue is full.




```
Q = circularQueue(6)
Q.Enqueue(5)
Q.Enqueue(10)
Q.Enqueue(15)
Q.Enqueue(20)
Q.Enqueue(25)
Q.Enqueue(30)
Q.Dequeue()
Q.Dequeue()
Q.Enqueue(35)
Q.Enqueue(40)
```

Implementation of circular queue using array

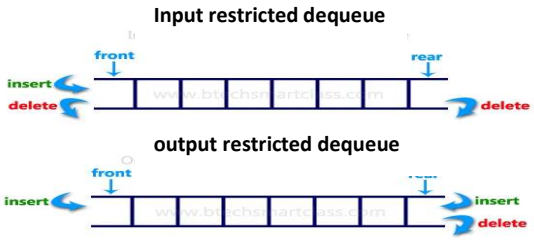
```
#include<stdio.h>
#define SIZE 5
int cqueue[SIZE],front=-1,rear=-1;
void insert(int value ){
  if((front==0 && rear ==SIZE-1) || (front==rear+1)){
    printf("queue is full ");
  }
  else{
    if(rear==SIZE-1 && front!=0){
      rear=-1;
    }
    cqueue[++rear]=value;
    if(front==-1){
      front=0;
    }
  } }
void delete(){
  if(front==-1 && rear==-1){
    printf("queue is empty ");
  }
  else{
    front=front+1;
    if(front==SIZE){
      front=0;
    }
    if(front-1==rear){
      front=rear=-1;
    }
  }  }
void display(){
  if(front==-1){
    printf("queue is empty");
  }
  else{
    int i =front;
    if(front<=rear){
      while (i<=rear)
      {
        printf("%d ", cqueue[i]);
        i++;  }
    }
    else{
      while (i<=SIZE-1)
      {
        printf("%d ", cqueue[i]);
        i++;
      }
      i=0;
      while (i<=rear)
      {
        printf("%d ", cqueue[i]);
        i++;
      }
    }  } }
int main(int argc, char const *argv[])
{
  insert(5);
  insert(6);
  display();
  delete();
  display();
  return 0;
}
```

Dequeue

- Insertion and deletion operations are performed at both ends .
- This dequeue can be used both as a stack and as a queue

Types of Dequeue

- **input restricted queue**, insertion operation can be performed at only one end, while deletion can be performed from both ends.
- **output restricted queue**, deletion operation can be performed at only one end, while insertion can be performed from both ends



Priority Queue

- It is data structure that behaves like a normal queue except that each element has some priority,
- elements are either arranged in an ascending or descending order.
- It has 2 type:
- 1. Ascending PQueue 2. Descending PQueue

Application priority Queue

- Optimization problems
- Heap sort using priority queue
- Dijkstra shortest path find using priority queue
- Scheduling the jobs in OS