**What is Recursion?**

When in any program a function calls itself, it is called recursion. It can happen directly or indirectly. The method of call leads to different types of recursion. Recursion is a problem-solving technique that contains some special properties. In the process of recursion, the function breaks down into different parts to solve the problem.

## Types of Recursion

There are two types of Recursion.

1. Direct Recursion
2. Indirect Recursion

## What is a Stack?

A stack is a **linear data structure** where elements are stored in the LIFO (Last In First Out) principle where the last element inserted would be the first element to be deleted. A stack is an Abstract Data Type (ADT), that is popularly used in most programming languages. It is named stack because it has the similar operations as the real-world stacks, for example − a pack of cards or a pile of plates, etc.



Stack is considered a complex data structure because it uses other data structures for implementation, such as Arrays, Linked lists, etc.

## Stack Representation

A stack allows all data operations at one end only. At any given time, we can only access the top element of a stack.

The following diagram depicts a stack and its operations –

**Algorithms – PUSH() - insert**

* PUSH (Stack, TOP, Item)
* This procedure will insert an element Item into the stack which is represented by array containing N elements with the pointer TOP denoting top element in the stack.

Step 1: [Check for stack overflow?]

If TOP >= N then

write(‘stack overflow’)

Exit

Step 2: [Increment the TOP pointer]

TOP <-TOP +1

Step 3: [Insert an element into the stack]

Stack[TOP] <-Item

Step 4: [Finished]

Return

**Algorithms – POP()**

* POP(Stack, TOP, Item)
* This algorithm deletes an element Item from the top of a stack S containing N elements.

Step 1: [Check for stack underflow?]

if TOP == 0 then

write (‘Stack Underflow’)

Exit

Step 2: [Accessing the value to be deleted]

Item <- Stack[TOP]

Step 3: [Decrement the stack pointer]

TOP<-TOP-1

Step 4: [Return deleted element Item ]

Return Item

**Algorithms – PEEP()**

* PEEP(Stack, TOP, I,item)
* Given an array Stack containing N elements. Pointer TOP elements top element of the stack. This algorithm fetched the ith element in the value item.

Step 1: [Check for stack underflow?]

If (TOP-i+1) <= 0 then

write(‘Stack underflow’)

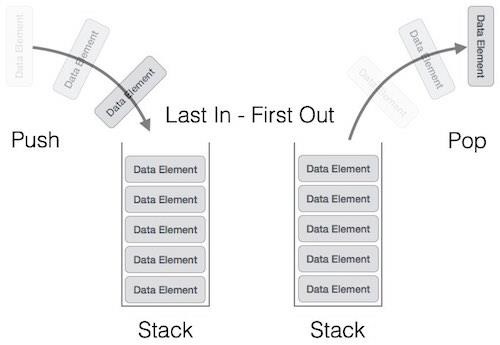
Exit

Step 2: [Storing the ith element in item]

item <- Stack[TOP-i+1]

Step 3: [Finished]

Return item



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.

**What is a Queue?**

The Queue is a linear data structure or an abstract data type. Queue follows the FIFO – “first in, first out” method to process the data. The data which is inserted first will be accessed first. Unlike Stack, Queue has two ends REAR and FRONT. The REAR end is always used to insert the data i.e., enqueue, and the FRONT end is used to remove the data inserted i.e. dequeue.

Representation of Queue

Linear array is the best and easiest way to represent the Queue. The Queue contains two ends, REAR and FRONT. These ends point to the position from where the insertion and deletion take place. The REAR points to the variable from where the insertion takes place and the FRONT points to the variable from where deletion takes place.

**Types of Queue:**

* 1. Simple Queue
  2. Circular Queue
  3. Double ended Queue
  4. Priority Queue

**Algorithm to insert an element in a simple queue**

**QINSERT(Q,F,N,R,Y)**

1. Step 1 : [Check for an overflow]
2. if R>=N then
3. write(‘Overflow’)
4. return
5. Step 2 : [increment rear pointer]
6. R<-R+1
7. Step 3 : [insert an element at rear position]
8. Q[R]<-Y
9. Step 4 : [is front pointer properly set ? ]
10. if F=0 then
11. F<-1
12. return
14. **Algorithm to delete an element in a simple queue**
15. **QDELETE(Q,F,R,Y)**
16. Step 1 : [Check for an underflow]
17. if F=0then
18. write(‘Underflow’)
19. return
20. Step 2 : [delete an element]
21. Y<-Q[F]
22. Step 3 : [is the queue empty ?]
23. if F=R then
24. F<-R<-0
25. else
26. F<-F+1
27. Step 4 : [Return deleted element ]
28. return (Y)

**Circular Queue**

The queue in which elements are arranged in a **circular fashion**

In a circular queue any element is accessible from any position but **only** in a **forward** **manner**.

**Algorithm to insert an element in a circular queue**

QINSERT(F,R,N,Q,Y)

Step 1: [Reset rear pointer ?]

if R=N

then R<-1

Else

R<-R+1

Step 2 : [Check for overflow]

if R=F

then write(‘OVERFLOW’)

Return

Step 3 : [Insert element]

Q[R]<-Y

Step 4 : [Is front pointer properly set?]

if F=0

then F<-1

Return

**Algorithm to delete an element in a circular queue**

QDELETE(F,R,Q,Y)

Step 1 : [Underflow ?]

if F=0

then write(‘Underflow … No elements to delete’)

Return

Step 2 : [Delete an element]

Y<-Q[F]

Step 3 : [Queue empty]

if R=F

then R<-F<-0

return (Y)

Step 4 : [Increment front pointer]

if F=N

then f<-1

else

F<-F+1

Return (Y)

**Double Ended queue (D-queue)**

* A D-queue is a linear list in which elements can be inserted or deleted from either end of a queue.
* **There are two variations in D-queue:**

1. An input restricted D-queue
2. An output restricted D-queue

* An **Input restricted D-queue** allows insertion only at one end of the queue but deletion at both the ends of a queue.
* **Algorithm for Insertion at rear end**
* Step-1: [Check for overflow]
* if(rear==MAX)
* Print("Queue is Overflow”);
* return;
* Step-2: [Insert Element]
* else
* rear=rear+1;
* q[rear]=no;
* [Set rear and front pointer]
* if front=0
* front=1;
* Step-3: return
* **Algorithm for Insertion at front end**
* Step-1 : [Check for the front position]
* if(front<=1)
* Print("Cannot add item at the front”);
* return;
* Step-2 : [Insert at front]
* else
* front=front-1;
* q[front]=no;
* Step-3 : Return
* **Algorithm for Deletion from front end**
* Step-1 [ Check for front pointer]
* if front=0
* print(" Queue is Underflow”);
* return;
* Step-2 [Perform deletion]
* else
* no=q[front];
* print(“Deleted element is”,no);
* [Set front and rear pointer]
* if front=rear
* front=0;
* rear=0;
* else
* front=front+1;
* Step-3 : Return
* **Algorithm for Deletion from rear end**
* Step-1 : [Check for the rear pointer]
* if rear=0
* print(“Cannot delete value at rear end”);
* return;
* Step-2: [ perform deletion]
* else
* no=q[rear];
* [Check for the front and rear pointer]
* if front= rear
* front=0;
* rear=0;
* else
* rear=rear-1;
* print(“Deleted element is”,no);
* Step-3 : Return
* **Priority Queue**

A queue in which we are able to insert items or remove items from any position based on some property is (based on the priority assigned to the tasks) is knows as Priority Queue

**Linked liner list :**

A linked list is a linear data structure that includes a series of connected nodes. Here, each node stores the data and the address of the next node.

Linked List is a sequence of nodes which contains items. Each link contains a connection to another nodes. Linked list is the second most-used data structure after array.

Following are the important terms to understand the concept of Linked List.

**Let's see how each node of the linked list is represented. Each node consists:**

**A data item**

**An address of another node**

**Types of Linked List**

Following are the various types of linked list.

**Simple Linked List** − Item navigation is forward only.

**Doubly Linked List** − Items can be navigated forward and backward.

**Circular Linked List** − Last item contains link of the first element as next and the first element has a link to the last element as previous.

**Basic Operations**

Following are the basic operations supported by a list.

**Insertion** − Adds an element at the beginning of the list.

**Deletion** − Deletes an element at the beginning of the list.

**Display** − Displays the complete list.

**Search** − Searches an element using the given key.

**Delete** − Deletes an element using the given key.

**Algorithm for insertion of a node at the first position**

**Function :** INSERT(X,FIRST). Given X, a new element , and FIRST , a pointer to the first element of a linked list whose typical node contains INFO and LINK fields as previously described , this function inserts X.AVAIL is a pointer to the top element of the availability stack; NEW is a temporary pointer variable. It is required that X precede the node whose address is given by FIRST.

1. [Underflow?]

If AVAIL = NULL

then Write(‘AVAILABILITY STACK UNDERFLOW’)

Return(FIRST)

2. [Obtain address of next free node]

NEW <- AVAIL

3. [Remove free node from availability stack]

AVAIL <- LINK(AVAIL)

4. [Initialise fields of new node and its link to the list]

INFO(NEW) <- X

LINK(NEW) <- FIRST

5.[Return address of new node]

Return(NEW)

**Algorithm for insertion of node at the end**

**Function:** INSEND(X,FIRST).Given X, a new element , and FIRST , a pointer to the first element of a linked linear list whose typical node contains INFO and LINK fields as previously described , this function inserts X.AVAIL is a pointer to the top element of the availability stack; NEW and SAVE are temporary pointer variables. It is required that X be inserted at the end of the list.

1.[Underflow?]

If AVAIL = NULL

then Write(‘AVAILABILITY STACK UNDERFLOW’)

Return(FIRST)

2. [Obtain address of next free node]

NEW <- AVAIL

3. [Remove free node from availability stack]

AVAIL <- LINK(AVAIL)

4. [Initialise fields of new node]

INFO(NEW) <- X

LINK(NEW) <- NULL

5. [Is the list empty?]

If FIRST = NULL

then Return(NEW)

6. [Initiate search for the last node]

SAVE <- FIRST

7. [Search for end of list]

Repeat while LINK(SAVE) NULL

SAVE <- LINK(SAVE)

8.[Set LINK field of last node to NEW]

LINK(SAVE) <- NEW

9.[Return first node pointer]

Return(FIRST)

**Algorithm for deletion of element from a linear list**

1.[Empty list?]

If FIRST = NULL

then Write(‘UNDERFLOW’)

Return

2. [Initialise search for X]

TEMP <- FIRST

3. [FIND X]

Repeat through step 5 while TEMP X and LINK(TEMP) NULL

4. [Update predecessor marker]

PRED <- TEMP

5. [Move to next node]

TEMP <- LINK(TEMP)

6. [End of the list?]

If TEMP X

then Write(‘NODE NOT FOUND’)

Return

7. [Delete X]

If X = FIRST (Is X the first node?)

then FIRST <- LINK(FIRST)

else LINK(PRED) <- LINK(X)

8. [Return node to availability area]

AVAIL <- X

Return

**Doubly linked lists**

**An algorithm for inserting a node to the left of a given node “M” in a doubly linked linear list.**

1. [Obtain new node from availability stack]

NEW <- NODE

2. [Copy information field]

INFO(NEW) <- X

3. [Insertion into an empty list?]

If R = NULL

then LPTR(NEW) <- RPTR(NEW) <- NULL

L <- R <- NEW

Return

1. [Obtain new node from availability stack]

NEW <- NODE

2. [Copy information field]

INFO(NEW) <- X

3. [Insertion into an empty list?]

If R = NULL

then LPTR(NEW) <- RPTR(NEW) <- NULL

L <- R <- NEW

Return

**An algorithm for deletion of a node from a doubly linked linear list**

1. [Underflow?]

If R=NULL

then Write(‘Underflow’)

Return

2. [Delete node]

If L=R (Single node in list)

then L <- R <- NULL

else If OLD = L (Left-most node being deleted)

then L <- RPTR(L)

LPTR(L) <- NULL

else If OLD = R (Right-most node being deleted)

then R <- LPTR(R)

RPTR(R) <- NULL

else RPTR(LPTR(OLD)) <- RPTR(OLD)

LPTR(RPTR(OLD)) <- LPTR(OLD)

3. [Return deleted node]

Restore(OLD)

Return