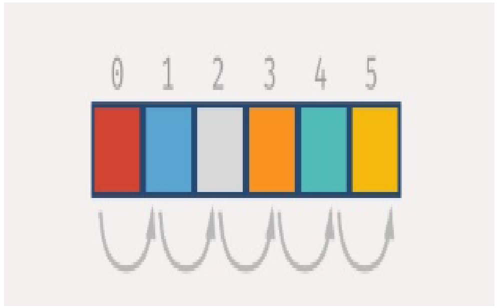
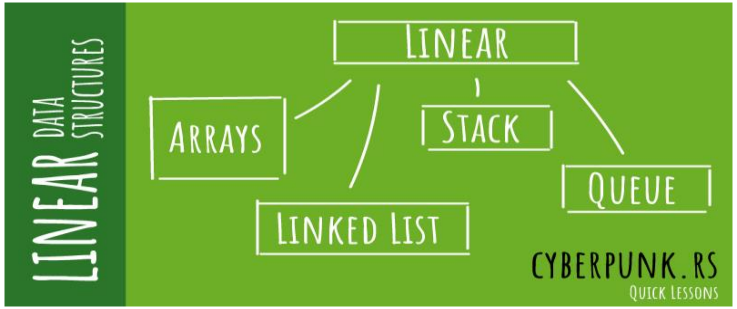
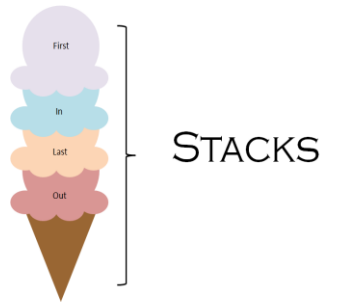
CHAPTER 2
LINEAR DATA STRUCTURE







SUBJECT:DATA STRUCTURE CODE:3130702	PREPARED BY: ASST.PROF.PARAS NARKHEDE (CSE DEPARTMENT,ACET)	

ARRAY

AMIRAJ
COLLEGE OF ENGINEERING & TECHNOLOGY

TOPICS TO BE COVERED

- Array:
- Representation of arrays
    - One dimensional array
    - Two dimensional array
- Applications of arrays
    - Symbol Manipulation (matrix representation of polynomial equation)
    - Sparse matrix
- Sparse matrix and its representation

AMIRAJ
COLLEGE OF ENGINEERING & TECHNOLOGY

REPRESENTATION OF ARRAY

➔ •An array is a group of consective memory locations with same name and data type.

➔ •Simple variable is a single memory location with unique name and a type. But an Array is collection of different adjacent memory locations. All these memory locations have one collective name and type.

➔ •The memory locations in the array are known as **elements** of array. The total number of elements in the array is called **length.**

➔ •The elements of array is accessed with reference to its position in array, that is call **index** or **subscript.**

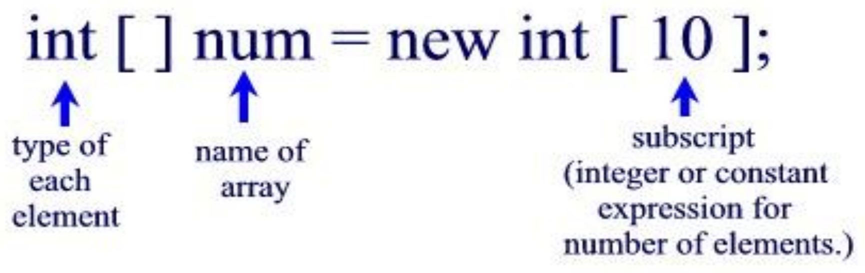
AMIRAJ COLLEGE OF ENGINEERING & TECHNOLOGY

ONE DIMENSIONAL ARRAY

➔ A type of array in which all elements are arranged in the form of a list is known as **1-D array** or **single dimensional array** or **linear list.**

➔ **Declaring 1-D Array:**

- data_type identifier[length];
- e.g: int marks[5];


int [ ] num = new int [10];
type of each element
name of array
subscript (integer or constant expression for number of elements.)

Initialization of 1-D array

Elements of an array can be initialized after an array is declared. An array can be declared at Compile time or at Run time

Compile time initialization

```
int age[5] = {20,21,19,23,25}
```

Here first element of age is initialized to 20, second to 21 and so on.

Run time initialization

```
for( int i =0; i<5;i++){
age[i]= 20;
   }
```

TWO DIMENSIONAL ARRAY

- Two-D array can be considered as table that consists of rows and columns. Each element in 2-D array is refered with the help of two indexes. One index indicates row and second indicates the column.
- **Declaring 2-D Array:**

  Data_type
  Identifier[row][colu
  mn]; e.g: int
  arr[4][3];

AMIRAJ
COLLEGE OF ENGINEERING & TECHNOLOGY

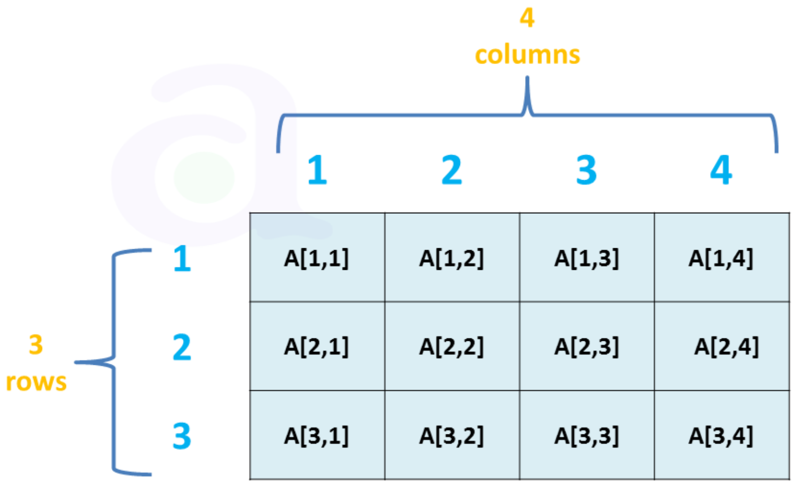
➔ The two-D array can also be initialized at the time of declaration. Initialization is performed by assigning the initial values in braces seperated by commas.

- Some important points :

➔ oThe elements of each row are enclosed within braces and seperated by comma.

➔ oAll rows are enclosed within braces.

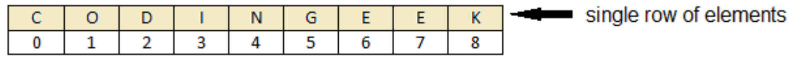
➔ oFor number arrays, if all elements are not specified , the un specified elements are initialized by zero.



	1	2	3	4
1	A[1,1]	A[1,2]	A[1,3]	A[1,4]
2	A[2,1]	A[2,2]	A[2,3]	A[2,4]
3	A[3,1]	A[3,2]	A[3,3]	A[3,4]

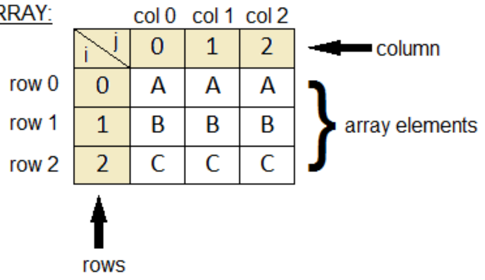
Fig: Two-dimension 3x4 array

1 D ARRAY:



C	O	D	I	N	G	E	E	K
0	1	2	3	4	5	6	7	8

2 D ARRAY:



		col 0	col 1	col 2
	i \ j	0	1	2
row 0	0	A	A	A
row 1	1	B	B	B
row 2	2	C	C	C

APPLICATION OF ARRAY

- Symbol Manipulation (matrix representation of polynomial equation)
  - Sparse Matrix

- Matrix representation of polynomial equation
  - We can use array for different kind of operations in polynomial equation such as addition, subtraction, division, differentiation etc…
  - We are interested in finding suitable representation for polynomial so that different operations like addition, subtraction etc… can be performed in efficient manner
  - Array can be used to represent Polynomial equation

AMIRAJ
COLLEGE OF ENGINEERING & TECHNOLOGY

SPARSE MATRIX

- An mXn matrix is said to be ***sparse*** if “many” of its elements are zero.
- A matrix that is not sparse is called a ***dense matrix***.
- We can device a simple representation scheme whose space requirement equals the size of the non-zero elements.

$$\begin{bmatrix} 0 & 0 & 3 & 0 & 4 \\ 0 & 0 & 5 & 7 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 2 & 6 & 0 & 0 \end{bmatrix} \Longrightarrow$$

Row	0	0	1	1	3	3
Column	2	4	2	3	1	2
Value	3	4	5	7	2	6

AMIRAJ
COLLEGE OF ENGINEERING & TECHNOLOGY

SPARSE MATRIX

- ➔ To construct matrix structure from liner representation we need to record
- ➔ Original row and columns of each non zero entries
- ➔ No of rows and columns in the matrix
- ➔ So each element of the array into which the sparse matrix is mapped need to have three fields: ***row*****,** ***column*** *and* ***value***

AMIRAJ
COLLEGE OF ENGINEERING & TECHNOLOGY

SPARSE MATRIX

- ➔ A super market conducting a study of the mix items purchased by its customers.
- ➔ For this study data are gathered for the purchase made by 1000 customers.
- ➔ These data are organized into a matrix, purchases with purchases(i,j) being the quantity of item i purchased by customer j.
- ➔ Suppose that the super market has an inventory of 10,000 different items.
- ➔ The purchase matrix is therefore a 10,000 x 1,000 matrix
- ➔ If the average customer buys 20 different items only about 20,000 of 1,00,000,000 matrix entries are nonzero

STACK

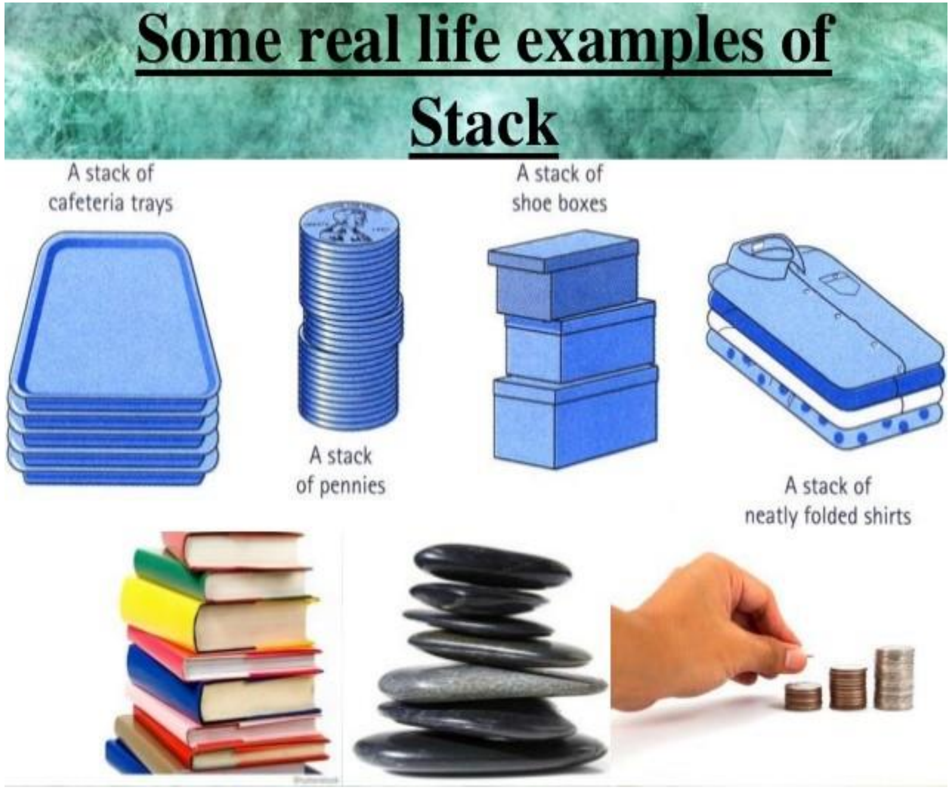
STACK DEFINITION & CONCEPT

A stack is a data structure in which items can be inserted only from one end and get items back from the same end. There , the last item inserted into stack, is the the first item to be taken out from the stack. In short its also called Last in First out [LIFO].

AMIRAJ
COLLEGE OF ENGINEERING & TECHNOLOGY

EXAMPLE OF STACK

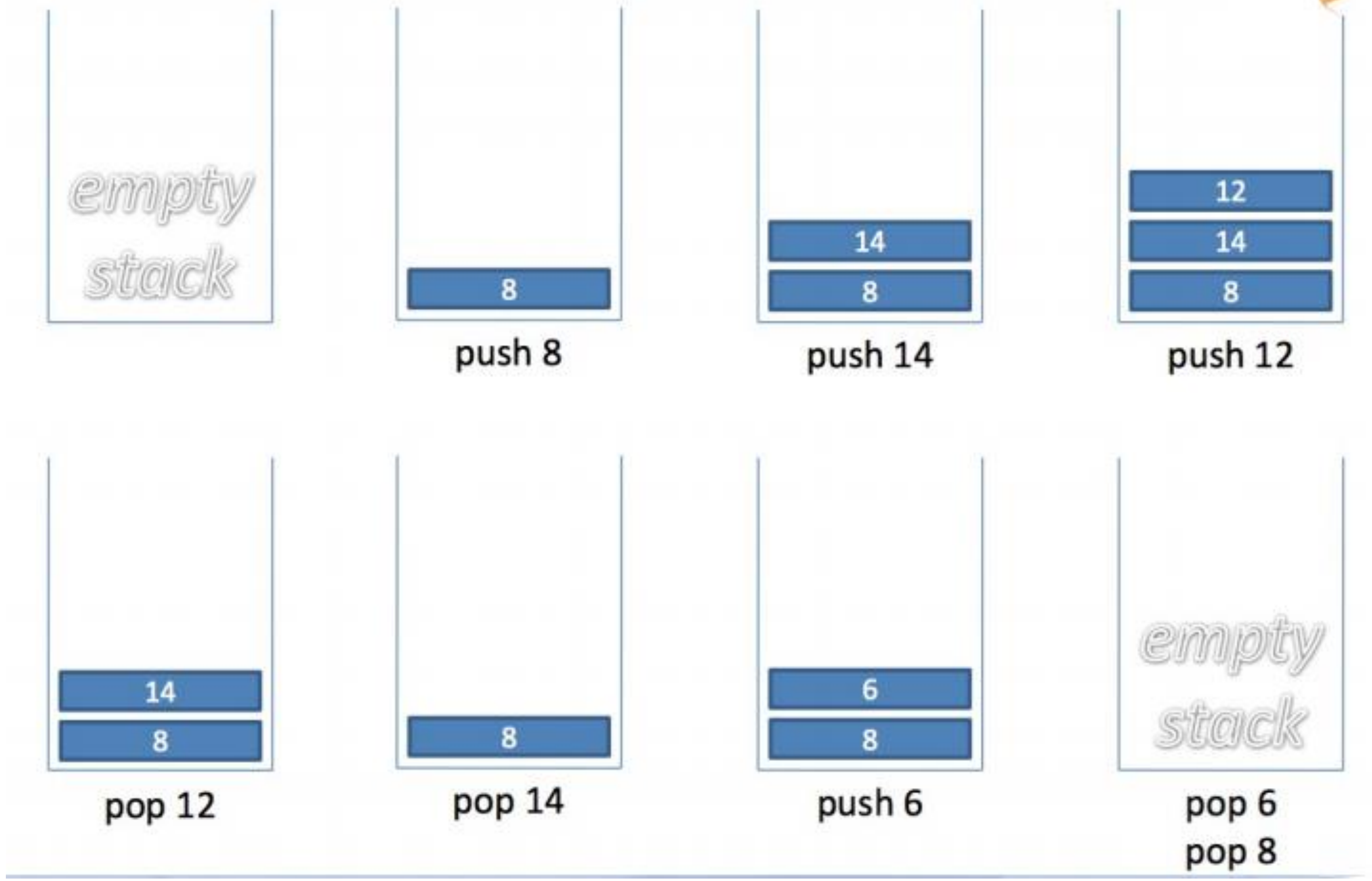
- ➔ A Stack of book on table.
- ➔ Token stack in Bank.
- ➔ Stack of trays and plates.


Some real life examples of Stack
A stack of cafeteria trays
A stack of pennies
A stack of shoe boxes
A stack of neatly folded shirts

AMIRAJ COLLEGE OF ENGINEERING & TECHNOLOGY

STACK OPERATION

- Top: Open end of the stack is called Top, From this end item can be inserted.
- Push: To insert an item from Top of stack is called push operation. The push operation change the position of Top in stack.
- POP: To put-off, get or remove some item from top of the stack is the pop operation, We can POP only only from top of the stack.
- IsEmpty: Stack considered empty when there is no item on Top. IsEmpty operation return true when no item in stack else false.
- IsFull: Stack considered full if no other element can be inserted on top of the stack. This condition normally occur when stack implement ed through array.


empty stack
8
push 8
14
8
push 14
12
14
8
push 12
14
8
pop 12
8
pop 14
6
8
push 6
empty stack
pop 6
pop 8

PUSH

- This procedure inserts an element **X** to the top of a stack
- Stack is represented by a vector **S** containing **N** elements
- A pointer **TOP** represents the top element in the stack.

```
1. [Check for stack underflow]
        If      TOP = 0
        Then    write ('STACK
UNDERFLOW')
                Return (0)
2. [Decrement TOP]
        TOP ← TOP - 1
3. [Return former top element
    of stack]
        Return(S[TOP + 1])
```

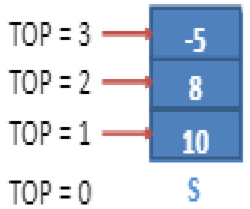
POP(S, TOP)

POP(S, TOP)

POP(S, TOP)

POP(S, TOP)

Underflow



POP

- This function ***removes & returns*** the top element from a stack.
- Stack is represented by a vector **S** containing **N** elements.
- A pointer **TOP** represents the top element in the stack.

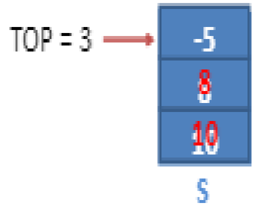
PEEP

- This function returns the value of the **Ith** element from the **TOP** of the stack. The element is not deleted by this function.
- Stack is represented by a vector **S** containing **N**

```
1. [Check for stack underflow]
        If      TOP-I+1 ≤ 0
        Then    write ('STACK UNDERFLOW')
                Return (0)
2. [Return Ith element from top of the stack]
        Return(S[TOP-I+1])
```

PEEP (S, TOP, 2)
PEEP (S, TOP, 3)
PEEP (S, TOP, 4)

Underflow



AMIRAJ COLLEGE OF ENGINEERING & TECHNOLOGY

APPLICATION OF STACK

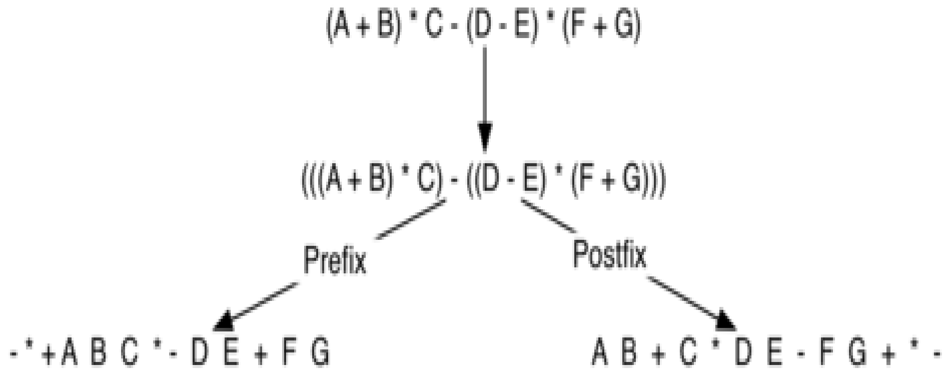
- **Expression Evaluation**
  - Stack is used to evaluate prefix, postfix and infix expressions.
- **Expression Conversion**
  - An expression can be represented in prefix, postfix or infix notation. Stack can be used to convert one form of expression to another.
- **Syntax Parsing**
  - Many compilers use a stack for parsing the syntax of expressions, program blocks etc. before translating into low level code.

AMIRAJ COLLEGE OF ENGINEERING & TECHNOLOGY

- **Backtracking**
    - Suppose we are finding a path for solving maze problem. We choose a path and after following it we realize that it is wrong. Now we need to go back to the beginning of the path to start with new path. This can be done with the help of stack.
- **Parenthesis Checking**
    - Stack is used to check the proper opening and closing of parenthesis.
- **String Reversal**
    - Stack is used to reverse a string. We push the characters of string one by one into stack and then pop character from stack.
- **Function Call**
    - Stack is used to keep information about the active functions or subroutines.

POLISH EXPRESSION

Infix, Postfix and Prefix notations are three different but equivalent ways of writing expressions. It is easiest to demonstrate the differences by looking at examples of operators that take two operands.


(A + B) * C - (D - E) * (F + G)
(((A + B) * C) - ((D - E) * (F + G)))
Prefix
Postfix
- * + A B C * - D E + F G
A B + C * D E - F G + * -

INFIX NOTATION

- Infix notation: X + Y
- Operators are written in-between their operands. This is the usual way we write expressions. An expression such as A * (B + C) / D is usually taken to mean something like: "First add B and C together, then multiply the result by A, then divide by D to give the final answer."
- Infix notation needs extra information to make the order of evaluation of the operators clear: rules built into the language about operator precedence and associativity, and brackets ( ) to allow users to override these rules. For example, the usual rules for associativity say that we perform operations from left to right, so the multiplication by A is assumed to come before the division by D. Similarly, the usual rules for precedence say that we perform multiplication and division before we perform addition and subtraction.

PREFIX NOTATION

- Prefix notation (also known as "Polish notation"): + X Y
- Operators are written before their operands. The expressions given above are equivalent to / * A + B C D
- As for Postfix, operators are evaluated left-to-right and brackets are superfluous. Operators act on the two nearest values on the right. I have again added (totally unnecessary) brackets to make this clear:
- (/ (* A (+ B C)) D)

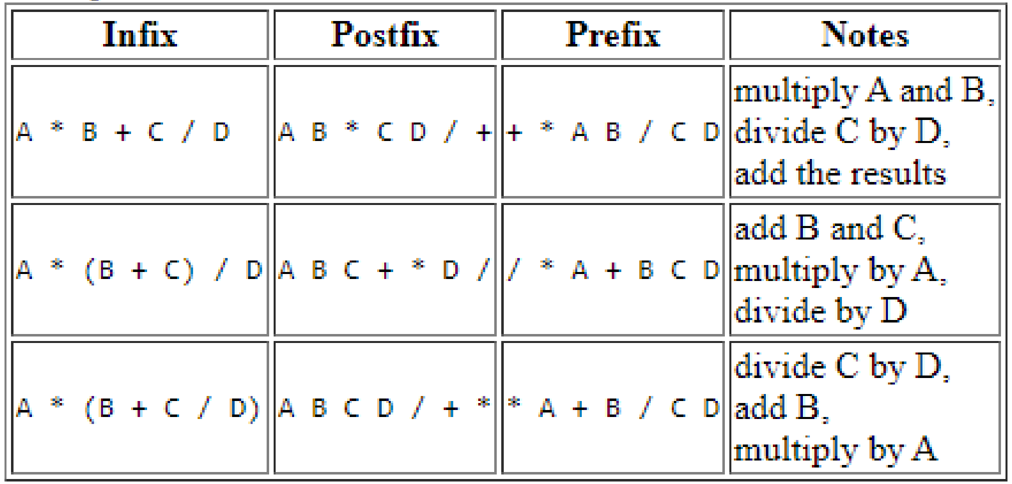
- Although Prefix "operators are evaluated left-to-right", they use values to their right, and if these values themselves involve computations then this changes the order that the operators have to be evaluated in. In the example above, although the division is the first operator on the left, it acts on the result of the multiplication, and so the multiplication has to happen before the division (and similarly the addition has to happen before the multiplication).
- Because Postfix operators use values to their left, any values involving computations will already have been calculated as we go left-to-right, and so the order of evaluation of the operators is not disrupted in the same way as in Prefix expressions.

POSTFIX NOTATION

- Postfix notation (also known as "Reverse Polish notation"): X Y +
- Operators are written after their operands. The infix expression given above is equivalent to A B C + * D /
- The order of evaluation of operators is always left-to-right, and brackets cannot be used to change this order. Because the "+" is to the left of the "*" in the example above, the addition must be performed before the multiplication.
- Operators act on values immediately to the left of them. For example, the "+" above uses the "B" and "C". We can add (totally unnecessary) brackets to make this explicit:
- ((A (B C +) *) D /)
- Thus, the "*" uses the two values immediately preceding: "A", and the result of the addition. Similarly, the "/" uses the result of the multiplication and the "D".

EXAMPLE

In all three versions, the operands occur in the same order, and just the operators have to be moved to keep the meaning correct. (This is particularly important for asymmetric operators like subtraction and division: A - B does not mean the same as B - A; the former is equivalent to A B - or - A B, the latter to B A - or - B A).



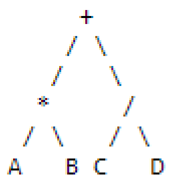
Infix	Postfix	Prefix	Notes
A * B + C / D	A B * C D / +	+ * A B / C D	multiply A and B, divide C by D, add the results
A * (B + C) / D	A B C + * D /	/ * A + B C D	add B and C, multiply by A, divide by D
A * (B + C / D)	A B C D / + *	* A + B / C D	divide C by D, add B, multiply by A

CONVERTING BETWEEN THIS NOTATION

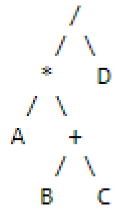
The most straightforward method is to start by inserting all the implicit brackets that show the order of evaluation e.g.:

Infix	Postfix	Prefix
((A * B) + (C / D))	((A B *) (C D /) +)	(+ (* A B) (/ C D))
((A * (B + C)) / D)	((A (B C +) *) D /)	(/ (* A (+ B C)) D)
(A * (B + (C / D)))	(A (B (C D /) +) *)	(* A (+ B (/ C D)))

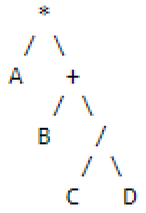
[illegible]



((A*B)+(C/D))



((A*(B+C))/D)



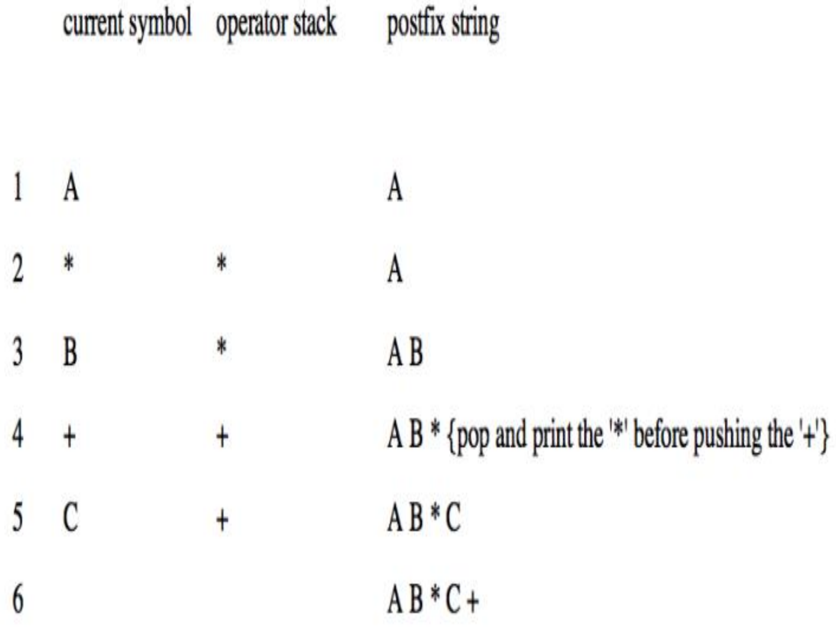
(A*(B+(C/D)))

Operator Precedence Chart

Operator Type	Operator	Associativity
Primary Expression Operators	`() [] . -> expr++ expr--`	left-to-right
Unary Operators	`* & + - ! ~ ++expr --expr (typecast) sizeof`	right-to-left
Binary Operators	`* / %`	left-to-right
	`+ -`	
	`>> <<`	
	`< > <= >=`	
	`== !=`	
	`&`	
	`^`	
	`|`	
	`&&`	
	`||`	
Ternary Operator	`?:`	right-to-left
Assignment Operators	`= += -= *= /= %= >>= <<= &= ^= |=`	right-to-left
Comma	`,`	left-to-right

INFIX TO POSTFIX

- ➔ Infix expression can be directly evaluated but the standard practice in CS is that the infix expression converted to postfix form and then the expression is evaluated. During both processes stack is proved to be a useful data structure.
- ➔ Example A*B+C become AB*C+



	current symbol	operator stack	postfix string
1	A		A
2	*	*	A
3	B	*	A B
4	+	+	A B * {pop and print the '*' before pushing the '+'}
5	C	+	A B * C
6			A B * C +

AMIRAJ
COLLEGE OF ENGINEERING & TECHNOLOGY

ALGORITHM

1. Given a expression in the infix form.

2. Find the highest precedence operator

3.If there are more then one operators with the same precedence check associativity, i.e. pick the left most first.

4.Convert the operator and its operands from infix to postfix A + B --> A B+

5.Repeat steps 2 to 4, until all the operators in the given expression are in the postfix form

INFIX TO PREFIX

1. Given a expression in the infix form.

2. Find the highest precedence operator

3.If there are more then one operators with the same precedence check associativity, i.e. pick the left most first.

4.Convert the operator and its operands from infix to prefix A + B --> +A B

5.Repeat steps 2 to 4, until all the operators in the given expression are in the postfix form.

Contin…..

Conversion of Infix to Postfix

Example to Convert Infix to Postfix using stack

a + (b*c)

Read character	Stack	Output
a	Empty	a
+	+	a
(	+(	a
b	+(	ab
*	+(*	ab
c	+(*	abc
)	+	abc*
		abc*+

COLLEGE OF ENGINEERING & TECHNOLOGY

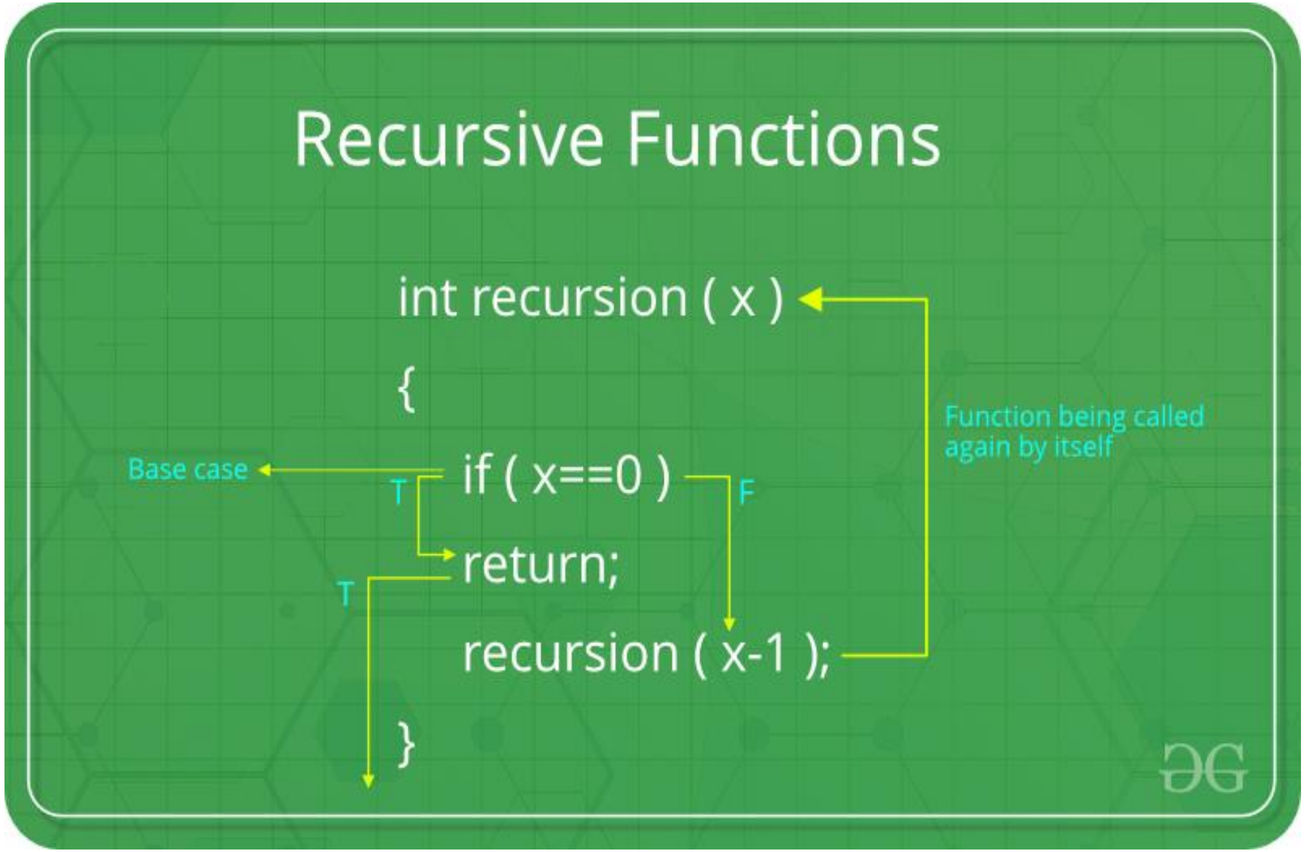
RECURSION

➔ Recursion is one of the most powerful tools in a programming language, but one of the most threatening topics-as most of the beginners and not surprising to even experienced students feel.

➔ When function is called within the same function, it is known as **recursion** in C. The function which calls the same function, is known as **recursive function**.

➔ Recursion is defined as defining anything in terms of itself. Recursion is used to solve problems involving iterations, in reverse order.

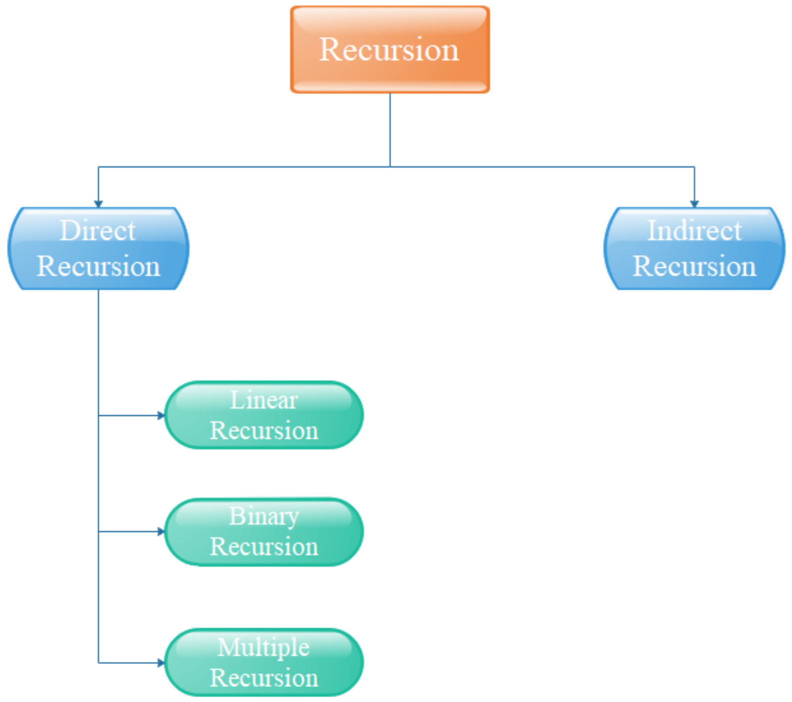
AMIRAJ COLLEGE OF ENGINEERING & TECHNOLOGY



Recursive Functions

```
int recursion ( x )
{
    if ( x==0 )
    return;
    recursion ( x-1 );
}
```

- **Types of Recursion**
  - There are two types of Recursion
    - Direct recursion
    - Indirect recursion
- **Direct Recursion**
- When in the body of a method t is a call to the same method, we that the method is **directly recursive**.
- There are three types of Direct Recursion
  - Linear Recursion
  - Binary Recursion
  - Multiple Recursion



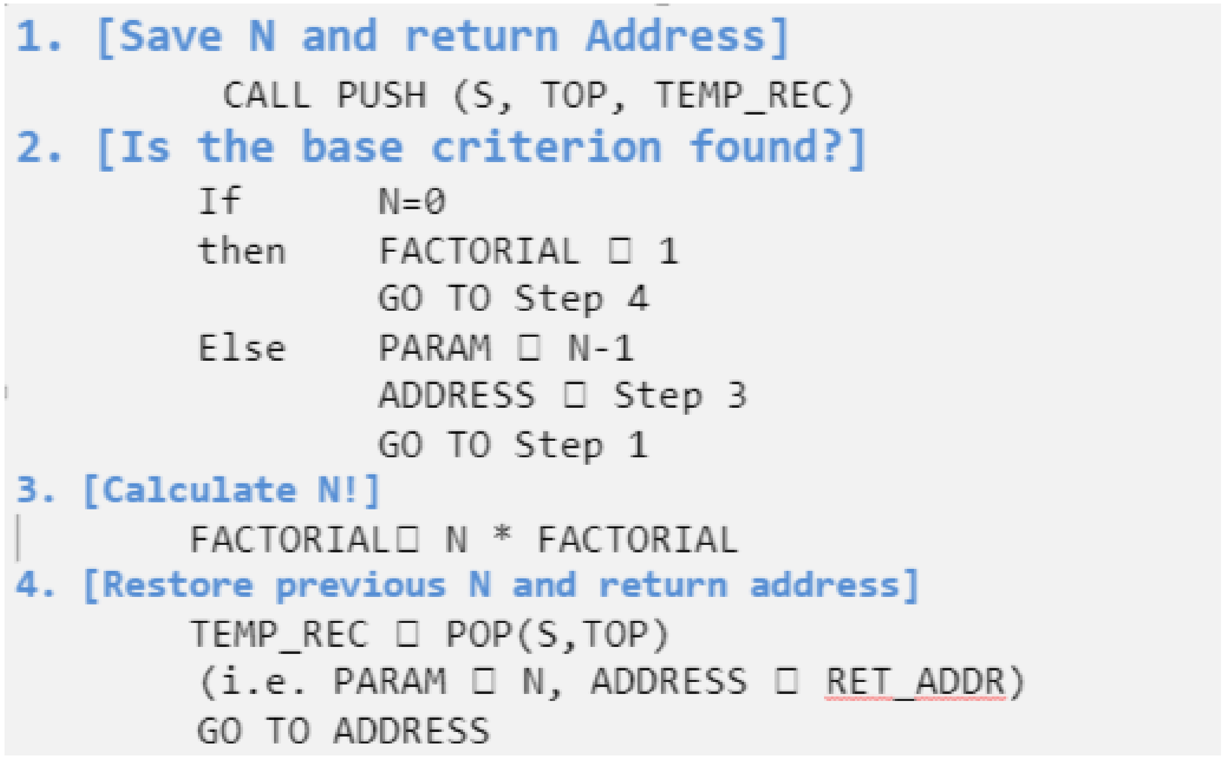
AMIRAJ
COLLEGE OF ENGINEERING & TECHNOLOGY

ALGORITHM TO FIND FACTORIAL USING RECURSION

- ➔ Given integer number **N**
- ➔ This algorithm **computes factorial of N**.
- ➔ **Stack S** is used to store an activation record associated with each recursive call.
- ➔ **TOP** is a pointer to the top element of stack S.
- ➔ Each **activation record contains** the current value of **N** and the current return address **RET_ADDE**.
- ➔ **TEMP_REC** is also a record which contains two variables **PARAM** & **ADDRESS**.
- ➔ **Initially** return address is set to the **main calling address**. PARAM is set to initial value N.

AMIRAJ
COLLEGE OF ENGINEERING & TECHNOLOGY

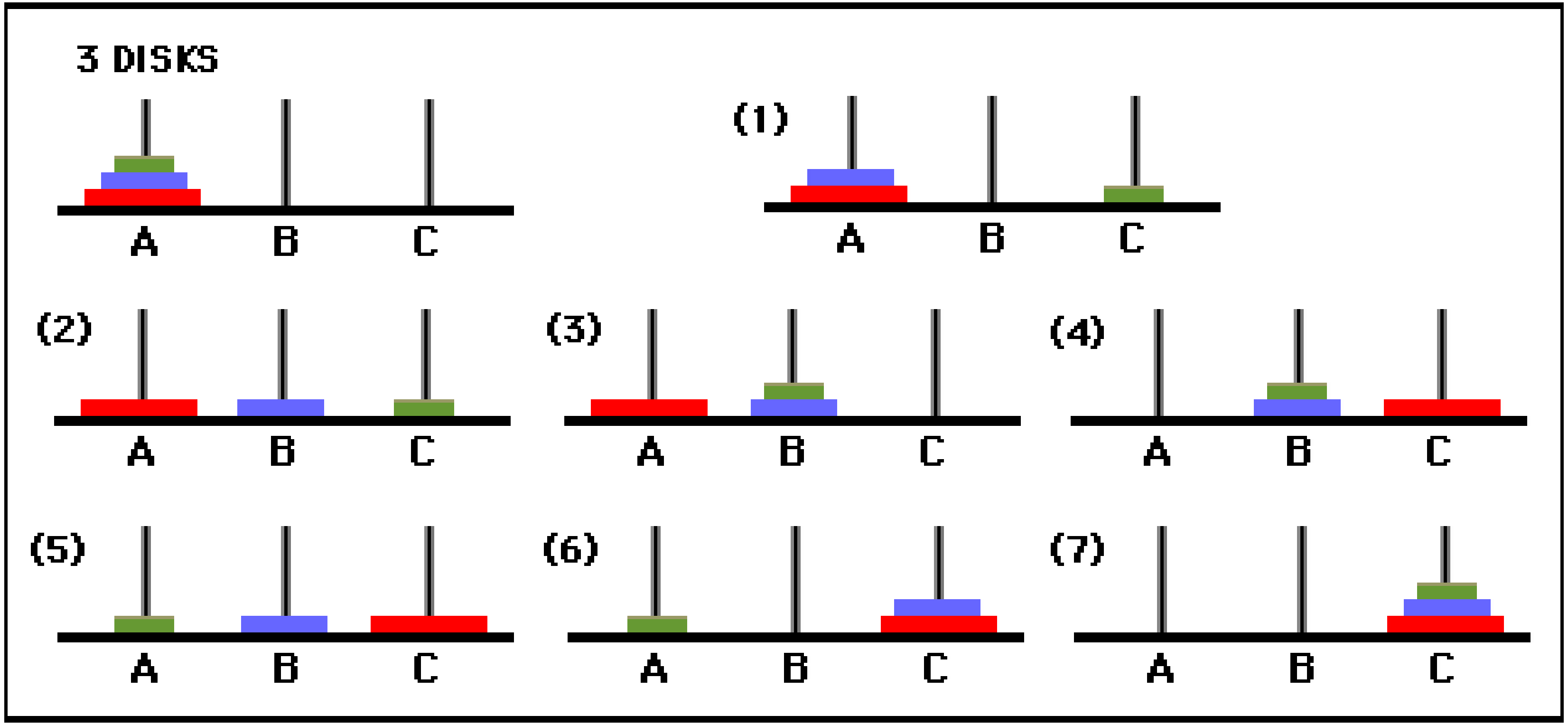
ALGORITHM:FACTORIAL



```
1. [Save N and return Address]
          CALL PUSH (S, TOP, TEMP_REC)
2. [Is the base criterion found?]
         If       N=0
         then     FACTORIAL □ 1
                  GO TO Step 4
         Else     PARAM □ N-1
                  ADDRESS □ Step 3
                  GO TO Step 1
3. [Calculate N!]
         FACTORIAL□ N * FACTORIAL
4. [Restore previous N and return address]
         TEMP_REC □ POP(S,TOP)
         (i.e. PARAM □ N, ADDRESS □ RET_ADDR)
         GO TO ADDRESS
```

TOWER OF HANOI

- These rings are of different sizes and stacked upon in an ascending order, i.e. the smaller one sits over the larger one. There are other variations of the puzzle where the number of disks increase, but the tower count remains the same.
- Rules
  - The mission is to move all the disks to some another tower without violating the sequence of arrangement. A few rules to be followed for Tower of Hanoi are −
  - Only one disk can be moved among the towers at any given time.
  - Only the "top" disk can be removed.
  - No large disk can sit over a small disk.


3 DISKS
A
B
C
(1)
(2)
(3)
(4)
(5)
(6)
(7)

ALGORITHM FOR TOWER OF HANOI

```
START

Procedure Hanoi(disk, source, dest, aux)

  IF disk == 1, THEN

    move disk from source to dest

  ELSE

    Hanoi(disk - 1, source, aux, dest)     // Step 1

    move disk from source to dest          // Step 2

    Hanoi(disk - 1, aux, dest, source)     // Step 3

  END IF

END Procedure

STOP
```

QUEUE

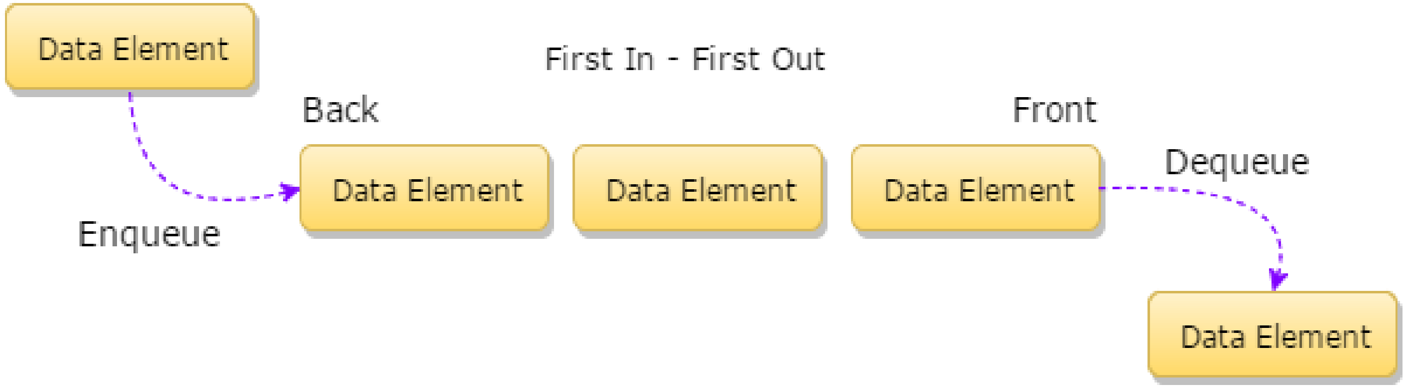
AMIRAJ
COLLEGE OF ENGINEERING & TECHNOLOGY

REPRESENTATION OF QUEUE

Queue is an abstract data structure, somewhat similar to stack. In contrast to stack, queue is opened at both end. One end is always used to insert data enqueue and the other is used to remove data dequeue. Queue follows First-In-First-Out methodology, i.e., the data item stored first will be accessed first.

LAST IN
LAST OUT
ONE WAY
FIRST IN
FIRST OUT

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



OPERATION ON QUEUE

- Queue operations may involve initializing or defining the queue, utilizing it and then completing erasing it from memory. Here we shall try to understand basic operations associated with queues –
  - enqueue – add store an item to the queue.
  - dequeue – remove access an item from the queue.

AMIRAJ
COLLEGE OF ENGINEERING & TECHNOLOGY

➔ Few more functions are required to make above mentioned queue operation efficient. These are –
  - peek – get the element at front of the queue without removing it.
  - isfull – checks if queue is full.
  - isempty – checks if queue is empty.

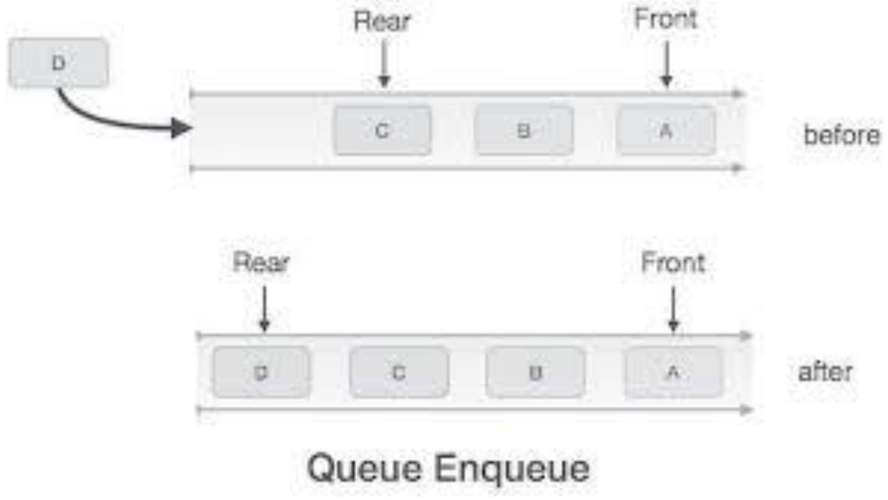
➔ In queue, we always dequeue oraccess data, pointed by front pointer and while enqueing orstoring data in queue we take help of rear pointer.

ENQUEUE OPERATION

- As queue maintains two data pointers, front and rear, its operations are comparatively more difficult to implement than stack. The following steps should be taken to enqueue insert data into a queue −
  - Step 1 − Check if queue is full.
  - Step 2 − If queue is full, produce overflow error and exit.
  - Step 3 − If queue is not full, increment rear pointer to point next empty space.
  - Step 4 − Add data element to the queue location, where rear is pointing.
  - Step 5 − return success.

ALGORITHM FOR ENQUEUE OPERATION

```
procedure enqueue(data)
   if queue is full
      return overflow
   endif

   rear ← rear + 1

   queue[rear] ← data

   return true

end procedure
```



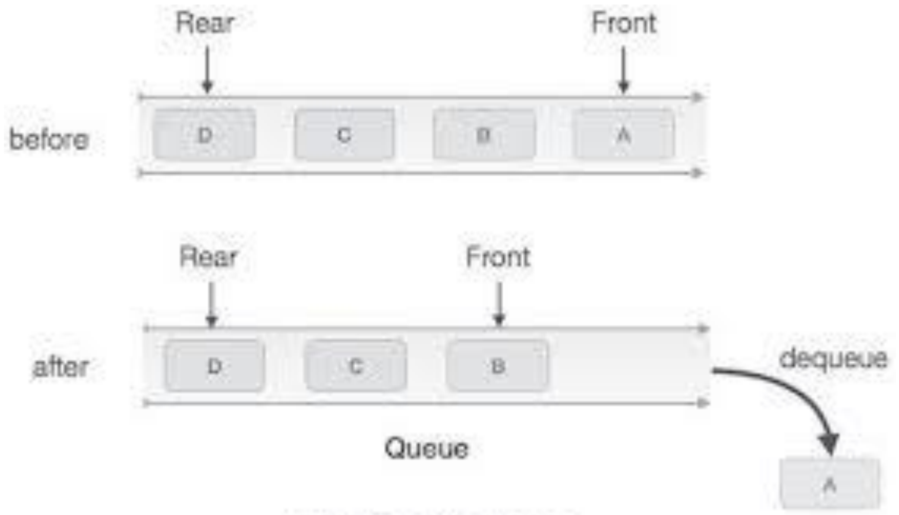
DEQUEUE OPERATION

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

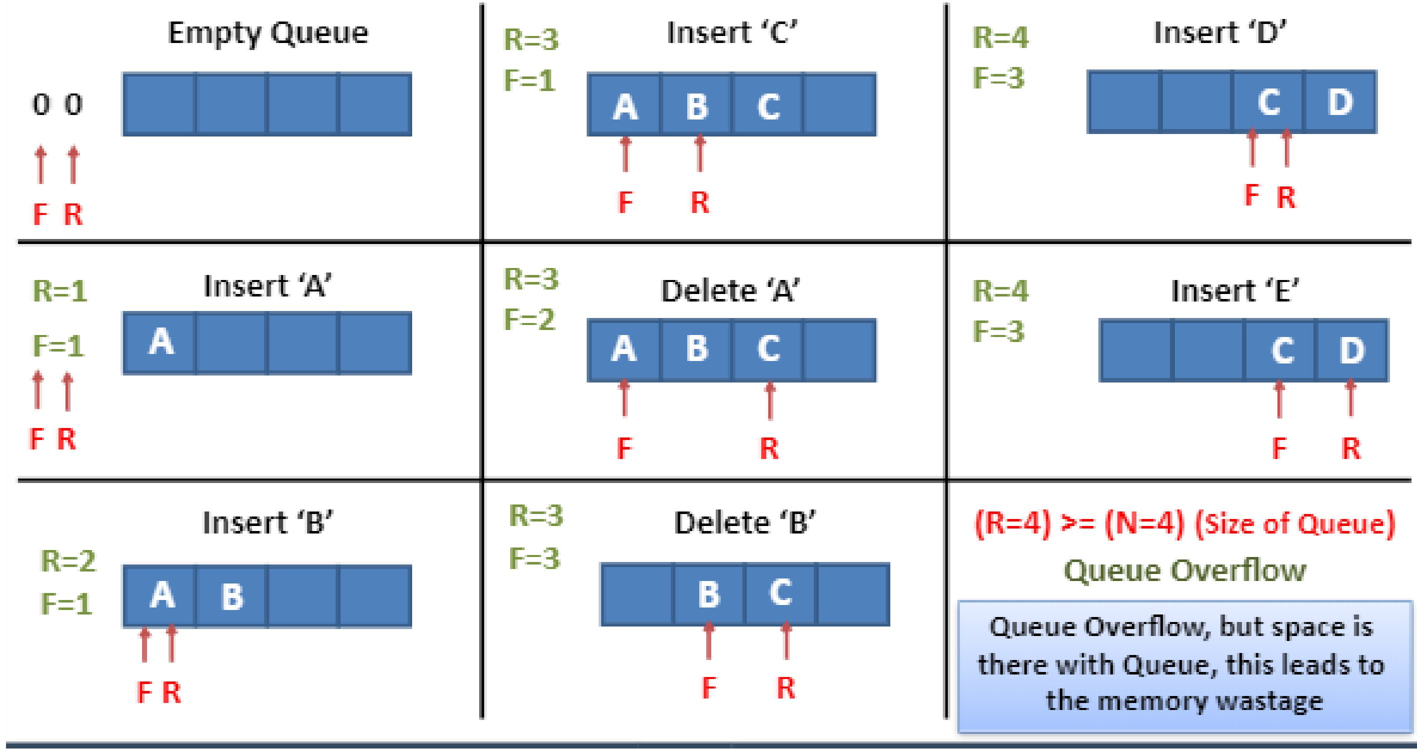
ALGORITHM FOR DEQUEUE OPERATION

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

   data = queue[front]
   front ← front - 1

   return true
end procedure
```



Queue Dequeue

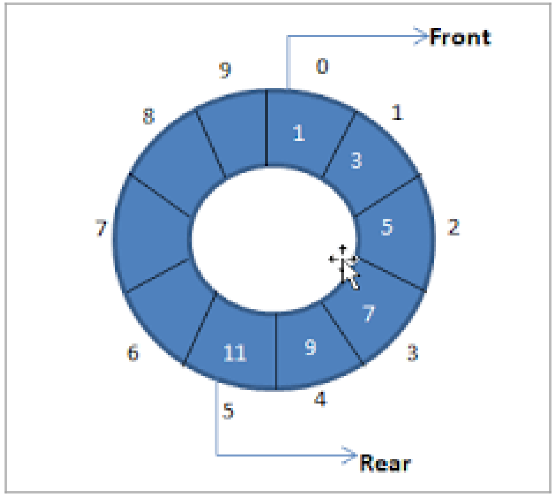
Perform following operations on queue with size 4 & draw queue after each operation
Insert 'A' | Insert 'B' | Insert 'C' | Delete 'A' | Delete 'B' | Insert 'D' | Insert 'E'



Queue Overflow, but space is there with Queue, this leads to the memory wastage

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. It is also called **‘Ring Buffer’**.
- In a normal Queue, we can insert elements until queue becomes full. But once queue becomes full, we can not insert the next element even if there is a space in front of queue.

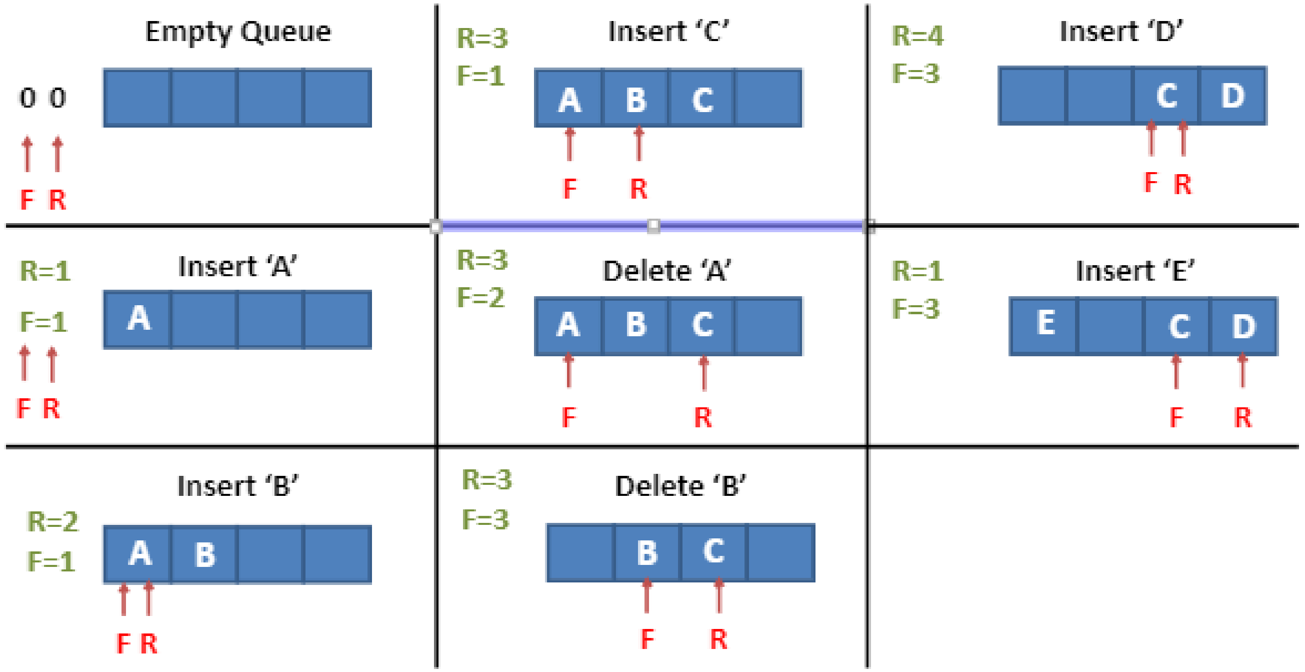


OPERATION ON CIRCULAR QUEUE

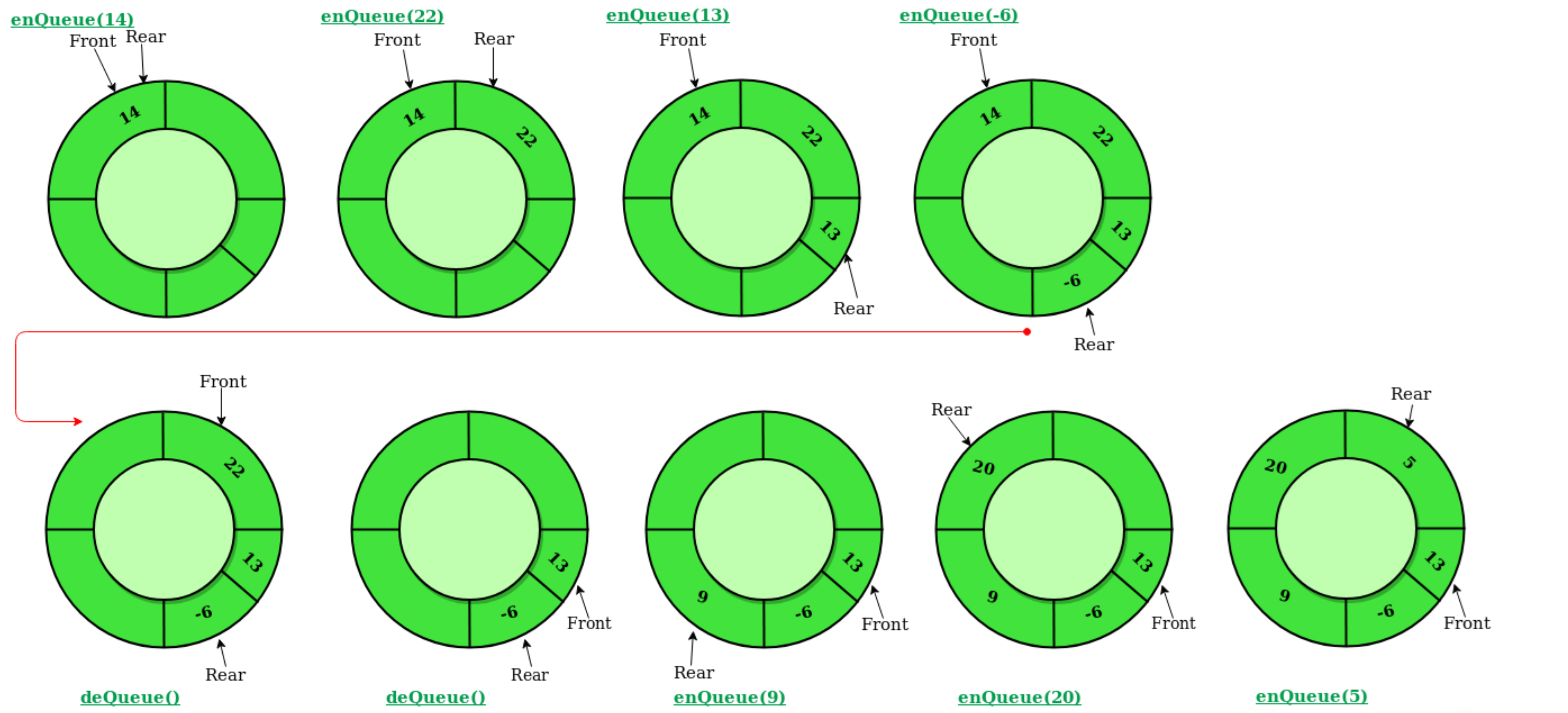
- **Front:** Get the front item from queue.
- **Rear:** Get the last item from queue.
- **enQueue(value)** This function is used to insert an element into the circular queue. In a circular queue, the new element is always inserted at Rear position.
    - **Steps:**Check whether queue is Full – Check ((rear == SIZE-1 && front == 0) || (rear == front-1)).
    - If it is full then display Queue is full. If queue is not full then, check if (rear == SIZE – 1 && front != 0) if it is true then set rear=0 and insert element.
- **deQueue()** This function is used to delete an element from the circular queue. In a circular queue, the element is always deleted from front position.
    - **Steps:**Check whether queue is Empty means check (front==-1).
    - If it is empty then display Queue is empty. If queue is not empty then step 3
    - Check if (front==rear) if it is true then set front=rear= -1 else check if (front==size-1), if it is true then set front=0 and return the element.

EXAMPLE OF CIRCULAR QUEUE

Perform following operations on Circular queue with size 4 & draw queue after each operation Insert 'A' | Insert 'B' | Insert 'C' | Delete 'A' | Delete 'B' | Insert 'D' | Insert 'E'



AMIRAJ COLLEGE OF ENGINEERING & TECHNOLOGY



AMIRAJ
COLLEGE OF ENGINEERING & TECHNOLOGY

enQueue(value) - Inserting value into the Circular Queue

➔ In a circular queue, enQueue() is a function which is used to insert an element into the circular queue. In a circular queue, the new element is always inserted at **rear** position. The enQueue() function takes one integer value as parameter and inserts that value into the circular queue. We can use the following steps to insert an element into the circular queue…

- Step 1 - Check whether **queue** is **FULL**. (**(rear == SIZE-1 && front == 0)** || **(front == rear+1)**)
- Step 2 - If it is **FULL**, then display **"Queue is FULL!!! Insertion is not possible!!!"** and terminate the function.
- Step 3 - If it is **NOT FULL**, then check **rear == SIZE - 1 && front != 0** if it is **TRUE**, then set **rear = -1**.
- Step 4 - Increment **rear** value by one (**rear++**), set **queue[rear]** = **value** and check '**front == -1**' if it is **TRUE**, then set **front = 0**.

AMIRAJ COLLEGE OF ENGINEERING & TECHNOLOGY

deQueue() - Deleting a value from the Circular Queue

➔ In a circular queue, deQueue() is a function used to delete an element from the circular queue. In a circular queue, the element is always deleted from **front** position. The deQueue() function doesn't take any value as a parameter. We can use the following steps to delete an element from the circular queue…

- Step 1 - Check whether **queue** is **EMPTY**. (**front == -1 && rear == -1**)
- Step 2 - If it is **EMPTY**, then display **"Queue is EMPTY!!! Deletion is not possible!!!"** and terminate the function.
- Step 3 - If it is **NOT EMPTY**, then display **queue[front]** as deleted element and increment the **front** value by one (**front ++**). Then check whether **front == SIZE**, if it is **TRUE**, then set **front = 0**. Then check whether both **front - 1** and **rear** are equal (**front -1** == **rear**), if it **TRUE**, then set both **front** and **rear** to '**-1**' (**front** = **rear** = **-1**).

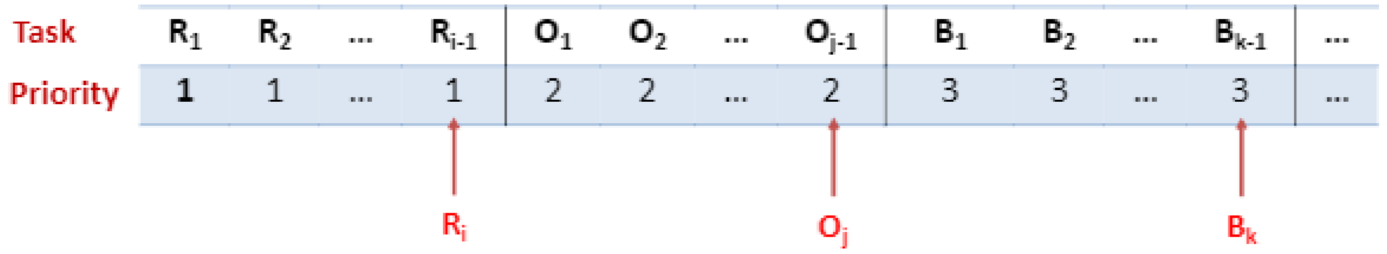
display() - Displays the elements of a Circular Queue

- We can use the following steps to display the elements of a circular queue…
  - Step 1 - Check whether **queue** is **EMPTY**. (**front == -1**)
  - Step 2 - If it is **EMPTY**, then display **"Queue is EMPTY!!!"** and terminate the function.
  - Step 3 - If it is **NOT EMPTY**, then define an integer variable '**i**' and set '**i** = **front**'.
  - Step 4 - Check whether '**front <= rear**', if it is **TRUE**, then display '**queue[i]**' value and increment '**i**' value by one (**i++**). Repeat the same until '**i <= rear**' becomes **FALSE**.
  - Step 5 - If '**front <= rear**' is **FALSE**, then display '**queue[i]**' value and increment '**i**' value by one (**i++**). Repeat the same until'**i <= SIZE - 1**' becomes **FALSE**.
  - Step 6 - Set **i** to **0**.
  - Step 7 - Again display '**cQueue[i]**' value and increment **i** value by one (**i++**). Repeat the same until '**i <= rear**' becomes **FALSE**.

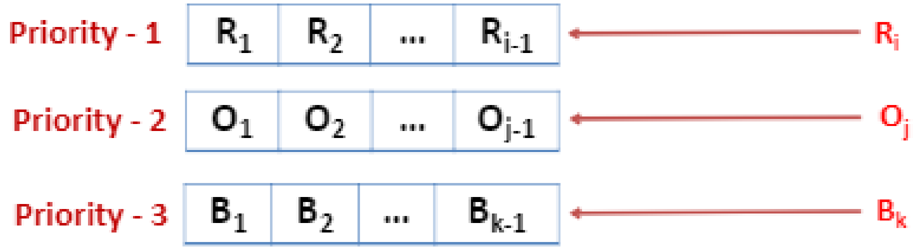
AMIRAJ
COLLEGE OF ENGINEERING & TECHNOLOGY

PRIORITY QUEUE

- A queue in which we are able to **insert & remove items** from **any position based on** some property (such as **priority** of the task to be processed) is often referred as **priority queue**.
- Below fig. represent a priority queue of jobs waiting to use a computer.
- Priorities are attached with each Job
    - **Priority 1** indicates **Real Time Job**
    - **Priority 2** indicates **Online Job**
    - **Priority 3** indicates **Batch Processing Job**
- Therefore if a job is initiated with priority i, it is inserted immediately at the end of list of other jobs with priorities i.
- Here jobs are always removed from the front of queue

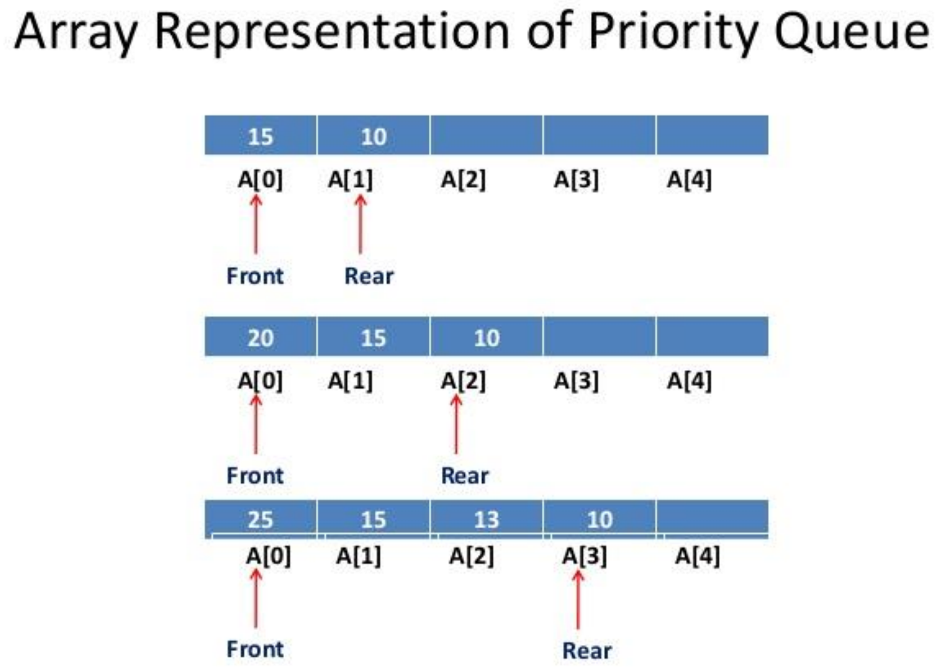

Task
Priority
R_1 R_2 ... R_{i-1} O_1 O_2 ... O_{j-1} B_1 B_2 ... B_{k-1} ...
1 1 ... 1 2 2 ... 2 3 3 ... 3 ...
R_i
O_j
B_k

Priority Queue viewed as a single queue with insertion allowed at any position


Priority - 1
R_1 R_2 ... R_{i-1}
R_i
Priority - 2
O_1 O_2 ... O_{j-1}
O_j
Priority - 3
B_1 B_2 ... B_{k-1}
B_k

Priority Queue viewed as a Viewed as a set of queue

ARRAY REPRESENTATION OF PRIORITY QUEUE



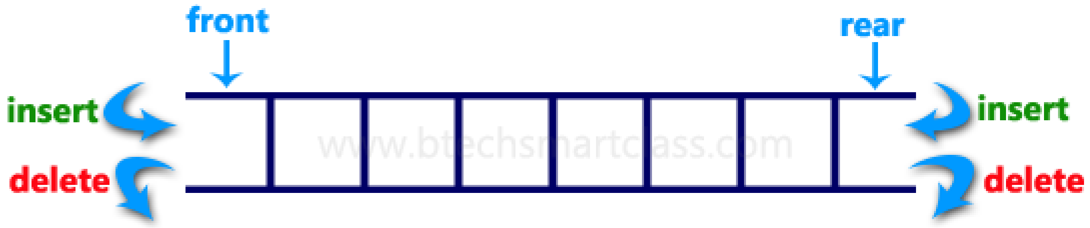
AMIRAJ COLLEGE OF ENGINEERING & TECHNOLOGY

DOUBLY ENDED QUEUE

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.

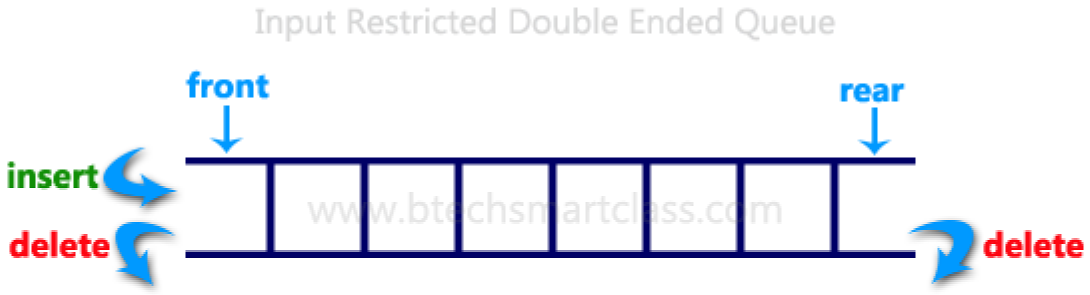
Double Ended Queue can be represented in TWO ways, those are as follows...

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



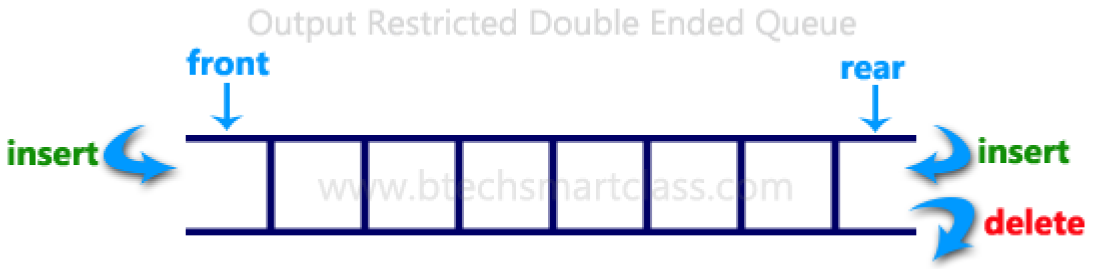
Input Restricted Double Ended Queue

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.



APPLICATION OF QUEUE

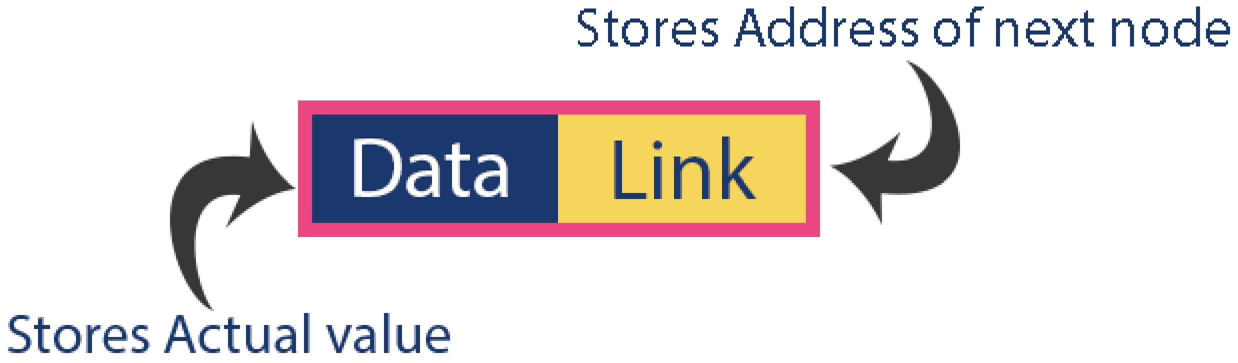
- Queue of people at any service point such as ticketing etc.
- Queue of air planes waiting for landing instructions.
- **Queue of processes** in OS.
- Queue is also used by Operating systems for **Job Scheduling**.
- When a **resource is shared** among multiple consumers. E.g., in case of printers the first one to be entered is the first to be processed.
- 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.
- Queue is used in **BFS (Breadth First Search)** algorithm. It helps in traversing a tree or graph.
- Queue is used in networking to **handle congestion**.

LINKED LIST

AMIRAJ
COLLEGE OF ENGINEERING & TECHNOLOGY

LINKED LIST

When we want to work with an unknown number of data values, we use a linked list data structure to organize that data. The linked list is a linear data structure that contains a sequence of elements such that each element links to its next element in the sequence. Each element in a linked list is called "Node".



SINGLY LINKED LIST

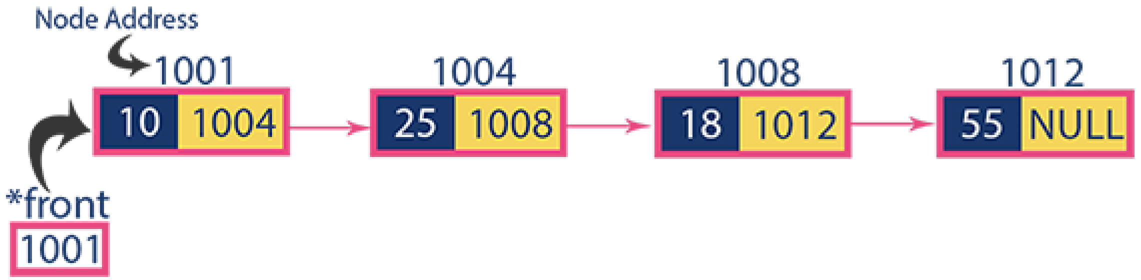
Simply a list is a sequence of data, and the linked list is a sequence of data linked with each other.

The formal definition of a single linked list is as follows...

Single linked list is a sequence of elements in which every element has link to its next element in the sequence.

In any single linked list, the individual element is called as "Node". Every "Node" contains two fields, data field, and the next field. The data field is used to store actual value of the node and next field is used to store the address of next node in the sequence.

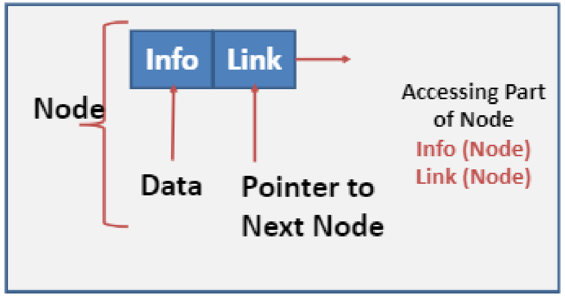
The graphical representation of a node in a single linked list is as follows...



— — —

Node Structure of Singly List

Typical Node



C Structure to represent a node

```
struct node
{
        int info;
        struct node *link;
};
```

AMIRAJ COLLEGE OF ENGINEERING & TECHNOLOGY

Operations on Single Linked List

- The following operations are performed on a Single Linked List
    - **Insertion**
    - **Deletion**
    - **Display**
- Before we implement actual operations, first we need to set up an empty list. First, perform the following steps before implementing actual operations.
    - Step 1 - Include all the **header files** which are used in the program.
    - Step 2 - Declare all the **user defined functions**.
    - Step 3 - Define a **Node** structure with two members **data** and **next**
    - Step 4 - Define a Node pointer **'head'** and set it to **NULL**.
    - Step 5 - Implement the main method by displaying operations menu and make suitable function calls in the main method to perform user selected operation.

Insertion

In a single linked list, the insertion operation can be performed in three ways. They are as follows...

1. Inserting At Beginning of the list
2. Inserting At End of the list
3. Inserting At Specific location in the list

Inserting At Beginning of the list

➔ We can use the following steps to insert a new node at beginning of the single linked list…

- Step 1 - Create a **newNode** with given value.
- Step 2 - Check whether list is **Empty** (**head == NULL**)
- Step 3 - If it is **Empty** then, set **newNode→next** = **NULL** and **head** = **newNode**.
- Step 4 - If it is **Not Empty** then, set **newNode→next** = **head** and **head** = **newNode**.

Inserting At End of the list

- We can use the following steps to insert a new node at end of the single linked list…
  - Step 1 - Create a **newNode** with given value and **newNode** → **next** as **NULL**.
  - Step 2 - Check whether list is **Empty** (**head** == **NULL**).
  - Step 3 - If it is **Empty** then, set **head** = **newNode**.
  - Step 4 - If it is **Not Empty** then, define a node pointer **temp** and initialize with **head**.
  - Step 5 - Keep moving the **temp** to its next node until it reaches to the last node in the list (until **temp** → **next** is equal to **NULL**).
  - Step 6 - Set **temp** → **next** = **newNode**.

Inserting At Specific location in the list (After a Node)

- We can use the following steps to insert a new node after a node in the single linked list…
  - Step 1 - Create a **newNode** with given value.
  - Step 2 - Check whether list is **Empty** (**head** == **NULL**)
  - Step 3 - If it is **Empty** then, set **newNode** → **next** = **NULL** and **head** = **newNode**.
  - Step 4 - If it is **Not Empty** then, define a node pointer **temp** and initialize with **head**.
  - Step 5 - Keep moving the **temp** to its next node until it reaches to the node after which we want to insert the newNode (until **temp1** → **data** is equal to **location**, here location is the node value after which we want to insert the newNode).
  - Step 6 - Every time check whether **temp** is reached to last node or not. If it is reached to last node then display **'Given node is not found in the list!!! Insertion not possible!!!'** and terminate the function. Otherwise move the **temp** to next node.
  - Step 7 - Finally, Set '**newNode** → **next** = **temp** → **next**' and '**temp** → **next** = **newNode**'

AMIRAJ
COLLEGE OF ENGINEERING & TECHNOLOGY

Deletion

In a single linked list, the deletion operation can be performed in three ways. They are as follows...

1. Deleting from Beginning of the list
2. Deleting from End of the list
3. Deleting a Specific Node

Deleting from Beginning of the list

➔ We can use the following steps to delete a node from beginning of the single linked list…

- Step 1 - Check whether list is **Empty** (**head** == **NULL**)
- Step 2 - If it is **Empty** then, display **'List is Empty!!! Deletion is not possible'** and terminate the function.
- Step 3 - If it is **Not Empty** then, define a Node pointer **'temp'** and initialize with **head**.
- Step 4 - Check whether list is having only one node (**temp** → **next** == **NULL**)
- Step 5 - If it is **TRUE** then set **head** = **NULL** and delete **temp** (Setting **Empty** list conditions)
- Step 6 - If it is **FALSE** then set **head** = **temp** → **next**, and delete **temp**.

Deleting from End of the list

- We can use the following steps to delete a node from end of the single linked list…
  - Step 1 - Check whether list is **Empty** (**head** == **NULL**)
  - Step 2 - If it is **Empty** then, display **'List is Empty!!! Deletion is not possible'** and terminate the function.
  - Step 3 - If it is **Not Empty** then, define two Node pointers **'temp1'** and '**temp2'** and initialize '**temp1**' with **head**.
  - Step 4 - Check whether list has only one Node (**temp1** → **next** == **NULL**)
  - Step 5 - If it is **TRUE**. Then, set **head** = **NULL** and delete **temp1**. And terminate the function. (Setting **Empty** list condition)
  - Step 6 - If it is **FALSE**. Then, set '**temp2 = temp1** ' and move **temp1** to its next node. Repeat the same until it reaches to the last node in the list. (until **temp1** → **next** == **NULL**)
  - Step 7 - Finally, Set **temp2** → **next** = **NULL** and delete **temp1**.

Deleting a Specific Node from the list

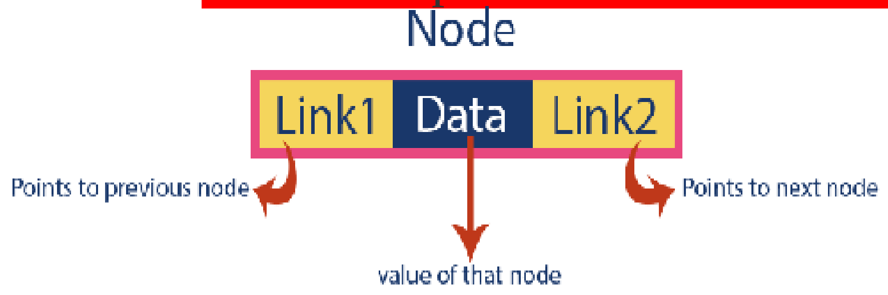
- We can use the following steps to delete a specific node from the single linked list…
  - Step 1 - Check whether list is **Empty** (**head** == **NULL**)
  - Step 2 - If it is **Empty** then, display **'List is Empty!!! Deletion is not possible'** and terminate the function.
  - Step 3 - If it is **Not Empty** then, define two Node pointers **'temp1'** and '**temp2**' and initialize '**temp1**' with **head**.
  - Step 4 - Keep moving the **temp1** until it reaches to the exact node to be deleted or to the last node. And every time set '**temp2 = temp1**' before moving the '**temp1**' to its next node.
  - Step 5 - If it is reached to the last node then display **'Given node not found in the list! Deletion not possible!!!'**. And terminate the function.
  - Step 6 - If it is reached to the exact node which we want to delete, then check whether list is having only one node or not
  - Step 7 - If list has only one node and that is the node to be deleted, then set **head** = **NULL** and delete **temp1** (**free(temp1)**).
  - Step 8 - If list contains multiple nodes, then check whether **temp1** is the first node in the list (**temp1 == head**).
  - Step 9 - If **temp1** is the first node then move the **head** to the next node (**head = head → next**) and delete **temp1**.
  - Step 10 - If **temp1** is not first node then check whether it is last node in the list (**temp1 → next == NULL**).
  - Step 11 - If **temp1** is last node then set **temp2 → next** = **NULL** and delete **temp1** (**free(temp1)**).
  - Step 12 - If **temp1** is not first node and not last node then set **temp2 → next** = **temp1 → next** and delete **temp1** (**free(temp1)**).

AMIRAJ
COLLEGE OF ENGINEERING & TECHNOLOGY

DOUBLY LINKED LIST

In a single linked list, every node has a link to its next node in the sequence. So, we can traverse from one node to another node only in one direction and we can not traverse back. We can solve this kind of problem by using a double linked list. A double linked list can be defined as follows...

Double linked list is a sequence of elements in which every element has links to its previous element and next element in the sequence.



Operations on Double Linked List

In a double linked list, we perform the following operations...

1. Insertion
2. Deletion
3. Display

Insertion

In a double linked list, the insertion operation can be performed in three ways as follows...

1. Inserting At Beginning of the list
2. Inserting At End of the list
3. Inserting At Specific location in the list

Inserting At Beginning of the list

- ➔ We can use the following steps to insert a new node at beginning of the double linked list…
  - Step 1 - Create a **newNode** with given value and **newNode** → **previous** as **NULL**.
  - Step 2 - Check whether list is **Empty** (**head** == **NULL**)
  - Step 3 - If it is **Empty** then, assign **NULL** to **newNode** → **next** and **newNode** to **head**.
  - Step 4 - If it is **not Empty** then, assign **head** to **newNode** → **next** and **newNode** to **head**.

Inserting At End of the list

➔ We can use the following steps to insert a new node at end of the double linked list…

- Step 1 - Create a **newNode** with given value and **newNode** → **next** as **NULL**.
- Step 2 - Check whether list is **Empty** (**head** == **NULL**)
- Step 3 - If it is **Empty**, then assign **NULL** to **newNode** → **previous** and **newNode** to **head**.
- Step 4 - If it is **not Empty**, then, define a node pointer **temp** and initialize with **head**.
- Step 5 - Keep moving the **temp** to its next node until it reaches to the last node in the list (until **temp** → **next** is equal to **NULL**).
- Step 6 - Assign **newNode** to **temp** → **next** and **temp** to **newNode** → **previous**.

Inserting At Specific location in the list (After a Node)

- We can use the following steps to insert a new node after a node in the double linked list…
  - Step 1 - Create a **newNode** with given value.
  - Step 2 - Check whether list is **Empty** (**head** == **NULL**)
  - Step 3 - If it is **Empty** then, assign **NULL** to both **newNode → previous** & **newNode → next** and set **newNode** to **head**.
  - Step 4 - If it is **not Empty** then, define two node pointers **temp1** & **temp2** and initialize **temp1** with **head**.
  - Step 5 - Keep moving the **temp1** to its next node until it reaches to the node after which we want to insert the newNode (until **temp1 → data** is equal to **location**, here location is the node value after which we want to insert the newNode).
  - Step 6 - Every time check whether **temp1** is reached to the last node. If it is reached to the last node then display **'Given node is not found in the list!!! Insertion not possible!!!'** and terminate the function. Otherwise move the **temp1** to next node.
  - Step 7 - Assign **temp1 → next** to **temp2**, **newNode** to **temp1 → next**, **temp1** to **newNode → previous**, **temp2** to **newNode → next** and **newNode** to **temp2 → previous**.

AMIRAJ
COLLEGE OF ENGINEERING & TECHNOLOGY

Deletion

In a double linked list, the deletion operation can be performed in three ways as follows...

1. Deleting from Beginning of the list
2. Deleting from End of the list
3. Deleting a Specific Node

Deleting from Beginning of the list

- We can use the following steps to delete a node from beginning of the double linked list…
  - Step 1 - Check whether list is **Empty** (**head** == **NULL**)
  - Step 2 - If it is **Empty** then, display **'List is Empty!!! Deletion is not possible'** and terminate the function.
  - Step 3 - If it is not Empty then, define a Node pointer **'temp'** and initialize with **head**.
  - Step 4 - Check whether list is having only one node (**temp** → **previous** is equal to **temp** → **next**)
  - Step 5 - If it is **TRUE**, then set **head** to **NULL** and delete **temp** (Setting **Empty** list conditions)
  - Step 6 - If it is **FALSE**, then assign **temp** → **next** to **head**, **NULL** to **head** → **previous** and delete **temp**.

Deleting from End of the list

- We can use the following steps to delete a node from end of the double linked list…
    - Step 1 - Check whether list is **Empty** (**head** == **NULL**)
    - Step 2 - If it is **Empty**, then display **'List is Empty!!! Deletion is not possible'** and terminate the function.
    - Step 3 - If it is not Empty then, define a Node pointer **'temp'** and initialize with **head**.
    - Step 4 - Check whether list has only one Node (**temp** → **previous** and **temp** → **next** both are **NULL**)
    - Step 5 - If it is **TRUE**, then assign **NULL** to **head** and delete **temp**. And terminate from the function. (Setting **Empty** list condition)
    - Step 6 - If it is **FALSE**, then keep moving **temp** until it reaches to the last node in the list. (until **temp** → **next** is equal to **NULL**)
    - Step 7 - Assign **NULL** to **temp** → **previous** → **next** and delete **temp**.

AMIRAJ
COLLEGE OF ENGINEERING & TECHNOLOGY

Deleting a Specific Node from the list

➔ We can use the following steps to delete a specific node from the double linked list…

- Step 1 - Check whether list is **Empty** (**head** == **NULL**)
- Step 2 - If it is **Empty** then, display **'List is Empty!!! Deletion is not possible'** and terminate the function.
- Step 3 - If it is not Empty, then define a Node pointer **'temp'** and initialize with **head**.
- Step 4 - Keep moving the **temp** until it reaches to the exact node to be deleted or to the last node.
- Step 5 - If it is reached to the last node, then display **'Given node not found in the list! Deletion not possible!!!'** and terminate the fuction.

- Step 6 - If it is reached to the exact node which we want to delete, then check whether list is having only one node or not
- Step 7 - If list has only one node and that is the node which is to be deleted then set **head** to **NULL** and delete **temp** (**free(temp)**).
- Step 8 - If list contains multiple nodes, then check whether **temp** is the first node in the list (**temp == head**).
- Step 9 - If **temp** is the first node, then move the **head** to the next node (**head = head → next**), set **head** of **previous** to **NULL** (**head → previous = NULL**) and delete **temp**.

➔ Step 10 - If **temp** is not the first node, then check whether it is the last node in the list (**temp → next == NULL**).

- Step 11 - If **temp** is the last node then set **temp** of **previous** of **next** to **NULL** (**temp → previous → next = NULL**) and delete **temp** (**free(temp)**).
- Step 12 - If **temp** is not the first node and not the last node, then set **temp** of **previous** of **next** to **temp** of **next** (**temp → previous → next = temp → next**), **temp** of **next** of **previous** to **temp** of **previous** (**temp → next → previous = temp → previous**) and delete **temp** (**free(temp)**).

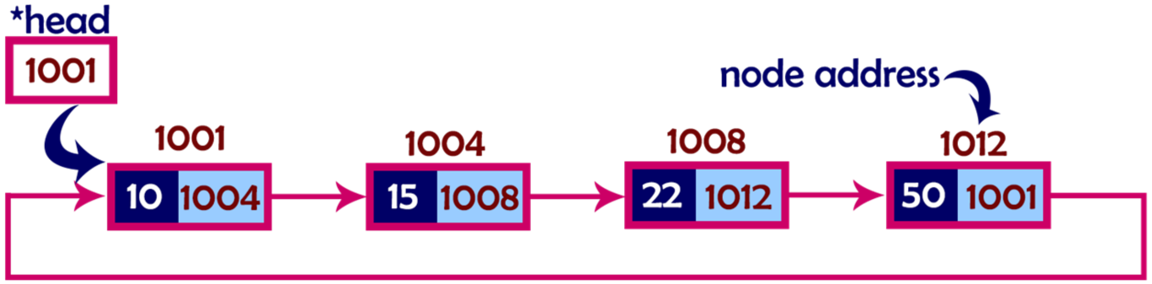
COLLEGE OF ENGINEERING & TECHNOLOGY

CIRCULAR LINKED LIST

In single linked list, every node points to its next node in the sequence and the last node points NULL. But in circular linked list, every node points to its next node in the sequence but the last node points to the first node in the list.

A circular linked list is a sequence of elements in which every element has a link to its next element in the sequence and the last element has a link to the first element.

That means circular linked list is similar to the single linked list except that the last node points to the first node in the list



Operations

In a circular linked list, we perform the following operations...

1. Insertion
2. Deletion
3. Display

Before we implement actual operations, first we need to setup empty list. First perform the following steps before implementing actual operations.

- Step 1 - Include all the **header files** which are used in the program.
- Step 2 - Declare all the **user defined** functions.
- Step 3 - Define a **Node** structure with two members **data** and **next**
- Step 4 - Define a Node pointer '**head**' and set it to **NULL**.
- Step 5 - Implement the **main** method by displaying operations menu and make suitable function calls in the main method to perform user selected operation.

Insertion

In a circular linked list, the insertion operation can be performed in three ways. They are as follows...

1. Inserting At Beginning of the list
2. Inserting At End of the list
3. Inserting At Specific location in the list

Inserting At Beginning of the list

- We can use the following steps to insert a new node at beginning of the circular linked list…
  - Step 1 - Create a **newNode** with given value.
  - Step 2 - Check whether list is **Empty** (**head** == **NULL**)
  - Step 3 - If it is **Empty** then, set **head** = **newNode** and **newNode→next** = **head** .
  - Step 4 - If it is **Not Empty** then, define a Node pointer '**temp**' and initialize with '**head**'.
  - Step 5 - Keep moving the '**temp**' to its next node until it reaches to the last node (until '**temp** → **next** == **head**').
  - Step 6 - Set '**newNode** → **next** =**head**', '**head** = **newNode**' and '**temp** → **next** = **head**'.

Inserting At End of the list

➔ We can use the following steps to insert a new node at end of the circular linked list…
- Step 1 - Create a **newNode** with given value.
- Step 2 - Check whether list is **Empty** (**head** == **NULL**).
- Step 3 - If it is **Empty** then, set **head** = **newNode** and **newNode** → **next** = **head**.
- Step 4 - If it is **Not Empty** then, define a node pointer **temp** and initialize with **head**.
- Step 5 - Keep moving the **temp** to its next node until it reaches to the last node in the list (until **temp** → **next** == **head**).
- Step 6 - Set **temp** → **next** = **newNode** and **newNode** → **next** = **head**.

Inserting At Specific location in the list (After a Node)

➔ We can use the following steps to insert a new node after a node in the circular linked list...

- Step 1 - Create a **newNode** with given value.
- Step 2 - Check whether list is **Empty** (**head** == **NULL**)
- Step 3 - If it is **Empty** then, set **head** = **newNode** and **newNode** → **next** = **head**.
- Step 4 - If it is **Not Empty** then, define a node pointer **temp** and initialize with **head**.
- Step 5 - Keep moving the **temp** to its next node until it reaches to the node after which we want to insert the newNode (until **temp1** → **data** is equal to **location**, here location is the node value after which we want to insert the newNode).
- Step 6 - Every time check whether **temp** is reached to the last node or not. If it is reached to last node then display **'Given node is not found in the list!!! Insertion not possible!!!'** and terminate the function. Otherwise move the **temp** to next node.
- Step 7 - If **temp** is reached to the exact node after which we want to insert the newNode then check whether it is last node (temp → next == head).
- Step 8 - If **temp** is last node then set **temp** → **next** = **newNode** and **newNode** → **next** = **head**.
- Step 8 - If **temp** is not last node then set **newNode** → **next** = **temp** → **next** and **temp** → **next** = **newNode**.

Deletion

In a circular linked list, the deletion operation can be performed in three ways those are as follows...

1. Deleting from Beginning of the list
2. Deleting from End of the list
3. Deleting a Specific Node

Deleting from Beginning of the list

- We can use the following steps to delete a node from beginning of the circular linked list...
  - Step 1 - Check whether list is **Empty** (**head** == **NULL**)
  - Step 2 - If it is **Empty** then, display **'List is Empty!!! Deletion is not possible'** and terminate the function.
  - Step 3 - If it is **Not Empty** then, define two Node pointers **'temp1'** and '**temp2**' and initialize both '**temp1**' and '**temp2**' with **head**.
  - Step 4 - Check whether list is having only one node (**temp1 → next** == **head**)
  - Step 5 - If it is **TRUE** then set **head** = **NULL** and delete **temp1** (Setting **Empty** list conditions)
  - Step 6 - If it is **FALSE** move the **temp1** until it reaches to the last node. (until **temp1 → next** == **head**)
  - Step 7 - Then set **head** = **temp2 → next**, **temp1 → next** = **head** and delete **temp2**.

AMIRAJ
COLLEGE OF ENGINEERING & TECHNOLOGY

Deleting from End of the list

- We can use the following steps to delete a node from end of the circular linked list...
  - Step 1 - Check whether list is **Empty** (**head** == **NULL**)
  - Step 2 - If it is **Empty** then, display **'List is Empty!!! Deletion is not possible'** and terminate the function.
  - Step 3 - If it is **Not Empty** then, define two Node pointers **'temp1'** and '**temp2'** and initialize '**temp1**' with **head**.
  - Step 4 - Check whether list has only one Node (**temp1 → next** == **head**)
  - Step 5 - If it is **TRUE**. Then, set **head** = **NULL** and delete **temp1**. And terminate from the function. (Setting **Empty** list condition)
  - Step 6 - If it is **FALSE**. Then, set '**temp2 = temp1** ' and move **temp1** to its next node. Repeat the same until **temp1** reaches to the last node in the list. (until **temp1** → **next** == **head**)
  - Step 7 - Set **temp2** → **next** = **head** and delete **temp1**.

Deleting a Specific Node from the list

- We can use the following steps to delete a specific node from the circular linked list...
  - Step 1 - Check whether list is **Empty** (**head** == **NULL**)
  - Step 2 - If it is **Empty** then, display **'List is Empty!!! Deletion is not possible'** and terminate the function.
  - Step 3 - If it is **Not Empty** then, define two Node pointers **'temp1'** and '**temp2**' and initialize '**temp1**' with **head**.
  - Step 4 - Keep moving the **temp1** until it reaches to the exact node to be deleted or to the last node. And every time set '**temp2 = temp1**' before moving the '**temp1**' to its next node.
  - Step 5 - If it is reached to the last node then display **'Given node not found in the list! Deletion not possible!!!'**. And terminate the function.
  - Step 6 - If it is reached to the exact node which we want to delete, then check whether list is having only one node (**temp1** → **next** == **head**)

- Step 7 - If list has only one node and that is the node to be deleted then set **head** = **NULL** and delete **temp1** (**free(temp1)**).
- Step 8 - If list contains multiple nodes then check whether **temp1** is the first node in the list (**temp1 == head**).
- Step 9 - If **temp1** is the first node then set **temp2** = **head** and keep moving **temp2** to its next node until **temp2** reaches to the last node. Then set **head = head → next**, **temp2** → **next** = **head** and delete **temp1**.
- Step 10 - If **temp1** is not first node then check whether it is last node in the list (**temp1 → next == head**).
- Step 1 1- If **temp1** is last node then set **temp2** → **next** = **head** and delete **temp1** (**free(temp1)**).
- Step 12 - If **temp1** is not first node and not last node then set **temp2** → **next** = **temp1** → **next** and delete **temp1** (**free(temp1)**).

AMIRAJ
COLLEGE OF ENGINEERING & TECHNOLOGY

LINKED LIST IMPLEMENTATION OF STACK

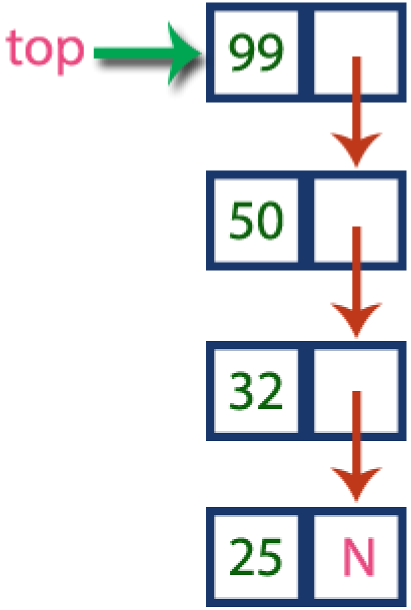
The major problem with the stack implemented using an array is, it works only for a fixed number of data values. That means the amount of data must be specified at the beginning of the implementation itself. Stack implemented using an array is not suitable, when we don't know the size of data which we are going to use. A stack data structure can be implemented by using a linked list data structure. The stack implemented using linked list can work for an unlimited number of values. That means, stack implemented using linked list works for the 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 previous node in the list. The **next** field of the first element must be always **NULL**.

Stack Operations using Linked List

➔ To implement a stack using a 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 a **Node** pointer '**top**' and set it to **NULL**.
- Step 4 - Implement the **main** method by displaying Menu with list of operations and make suitable function calls in the **main** method.



AMIRAJ COLLEGE OF ENGINEERING & TECHNOLOGY

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

AMIRAJ
COLLEGE OF ENGINEERING & TECHNOLOGY

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

display() - Displaying stack of elements

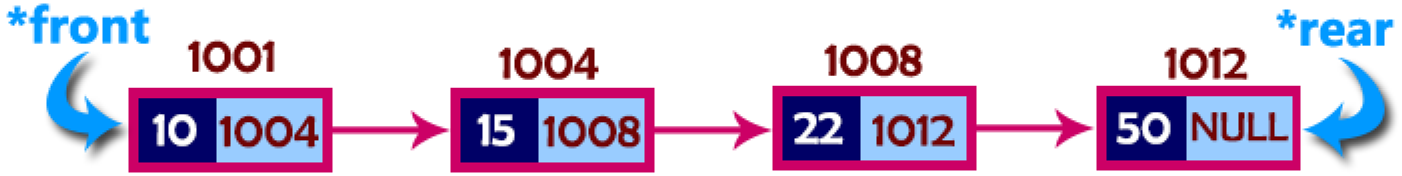
➔ We can use the following steps to display the elements (nodes) of a stack...

- 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**'.

LINKED LIST IMPLEMENTATION OF QUEUE

The major problem with the queue implemented using an array is, It will work for an only fixed number of data values. That means, the amount of data must be specified at the beginning itself. Queue using an 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 a linked list data structure. The queue which is implemented using a linked list can work for an unlimited number of values. That means, queue using linked list can work for the variable size of data (No need to fix the size at the 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**'



AMIRAJ
COLLEGE OF ENGINEERING & TECHNOLOGY

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

➔ We can use the following steps to insert a new node 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

- We can use the following steps to delete a node from the 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

- We can use the following steps to display the elements (nodes) of a 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 5 - Finally! Display '**temp** → **data** ---> **NULL**'.

APPLICATION OF LINKED LIST

- Implementation of stacks and queues.
- Implementation of graphs : Adjacency **list** representation of graphs is most popular which is uses **linked list** to store adjacent vertices.
- Dynamic memory allocation : We use **linked list** of free blocks.
- Maintaining directory of names.
- Performing arithmetic operations on long integers.

AMIRAJ
COLLEGE OF ENGINEERING & TECHNOLOGY