CMR College of Engineering & Technology

Kandlakoya(v), Medchal Road Hyderabad, Telangana, India - 501401,

Telephone: 08418 - 200699. Email: info@cmrcet.ac.in.

EXPLORE TO INVENT

Data Structures Lab Manual

COMPUTER SCIENCE & ENGINEERING DEPARTMENT

CMRCOLLEGE OF ENGINEERING

&

TECHNOLOGY

KANDLAKOYA, MEDCHAL ROAD, HYDERABAD–501401

Lab Manual

Data Structures Laboratory

(Common to EEE, ECE, CSE, IT, CSE-AI&ML&CSE-Cyber Security)
B. Tech. II Semester

Department of Computer Science and Engineering

2022-2023

CMR College of Engineering & Technology
Department of CSE
DATA STRUCTURES LAB
(Common to ECE, CSE, EEE, IT, CSE (AIML)&CSE(Cyber Security))

Reference Books:

1. Ellis Horowitz, Sartaj Sahni, Fundamentals of Data Structures in C, Second Edition Universities Press.
2. Thomas H. Cormen Charles E. Leiserson, Introduction to Algorithms, PHI Learning Pvt. Ltd. Third edition.
3. Algorithms, Data Structures, and Problem Solving with C++", Illustrated Edition by Mark Allen Weiss, Addison-Wesley Publishing Company.
4. E.Balagurusamy Data Structures Using C, McGraw Hill Education; First edition.

Q1: Write a C program to perform the following operations on the given array

(i) Insert element in specific position into array

(ii) Delete random element from array

(iii) Reverse the array elements An array is a collection of items store data contiguous memory locations.

Algorithm:

Step1: Start

Step2: Read number of elements

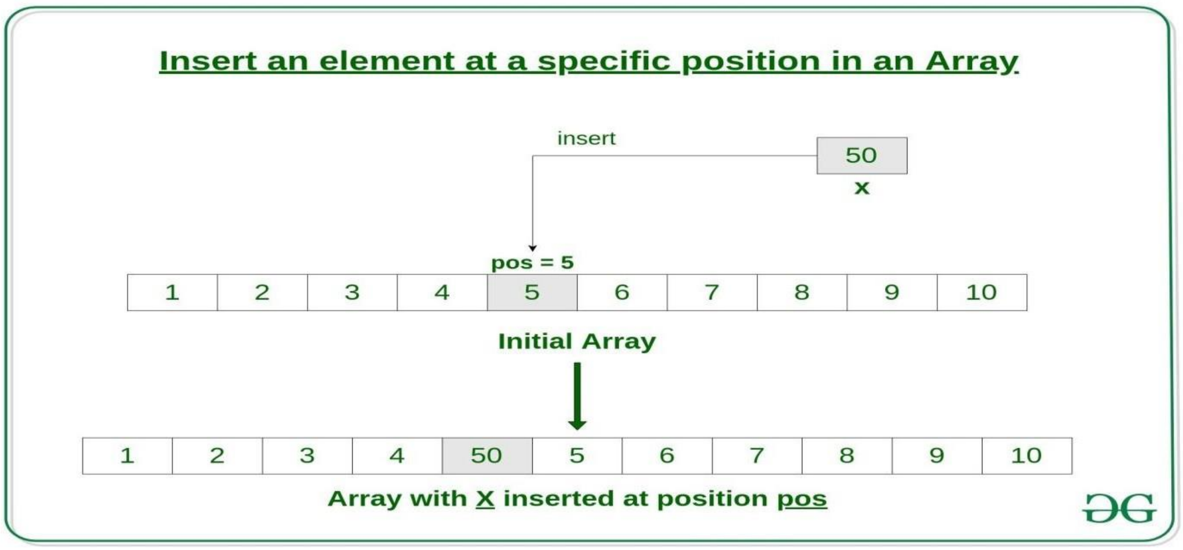
Step3: Read Array of elements

Step4: Enter your choice to insert/delete/reverse the Given array

Step 5: Stop

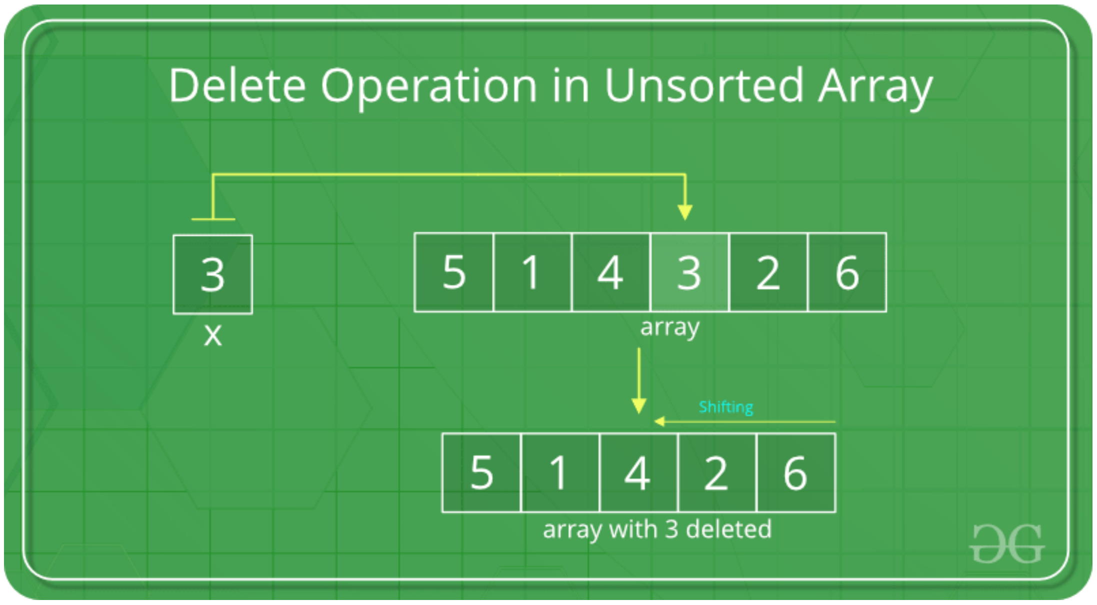
Insert element in specific position in to array:

1) First get the element to be inserted, say
2) Then get the position at which this element is to be inserted, say pos
3) Then shift the array elements from this position to one position forward, and do this for all the other elements next to pos.
4) Insert the element x now at the position pos, as this is now empty.


Insert an element at a specific position in an Array
insert
50
x
pos = 5
1 2 3 4 5 6 7 8 9 10
Initial Array
1 2 3 4 50 5 6 7 8 9 10
Array with X inserted at position pos

Delete random element from array:

In the delete operation, the element to be deleted is searched using the linear search, and then delete operation Is performed followed by shifting the elements.
First search ‘x’ in array, then delete that element and shift all elements to one position back.



Reverse the array elements:

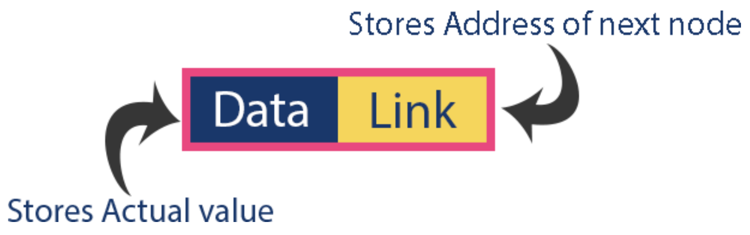
1) Inputsizeandelementsinanarray.Storeitinsomevariablesaysizeandarrrespectively.
2) Declareanotherarraythatwillstorereversedarrayelementsoforiginalarraywithsamesize,say reverse[size].
3) Initialize two variables that will keep track of original and reverse array. Here we will access
    a. Original array from last and reverse array from first. Hence, initializer Index=size-1andrevIndex=0.
4) Run loop fromsize-1to 0 in decremented style .The loop structure should look like while (arrIndex>=0).
5) In side loop copy original array to reverse array i.e. reverse[revIndex]=arr[arrIndex];.
6) After copy, increment rev Index and decrement arr Index.
7) Finally after loop print reverse array.

Q2.WriteaC program to implement Single linked list

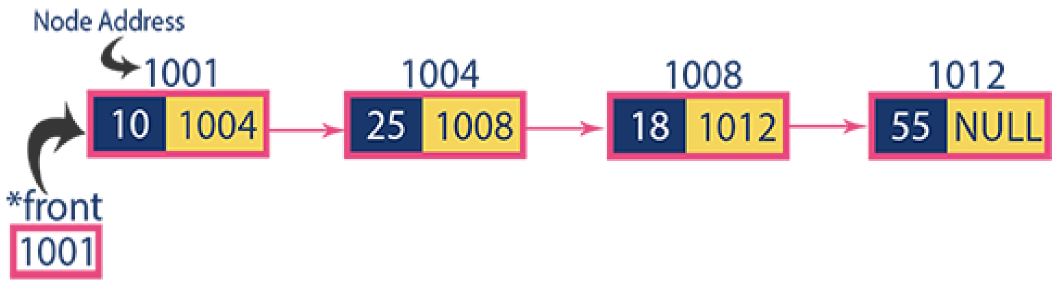
Linked List is a linear data structure in which every data item is represented as Node containing two or more lots.

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 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 node in a single linked list is as follows...



Example



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 setup empty list. First perform the following steps before implementing actual operations.

Step1- Include all the **header files** which are used in the program.

Step2-Declare all the **user defined functions**.

Step3 -Define a **Node** structure with two members **data** and **next**

Step4-Define a Node pointer **‘starts'** and set it to **NULL**.

Step5-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:

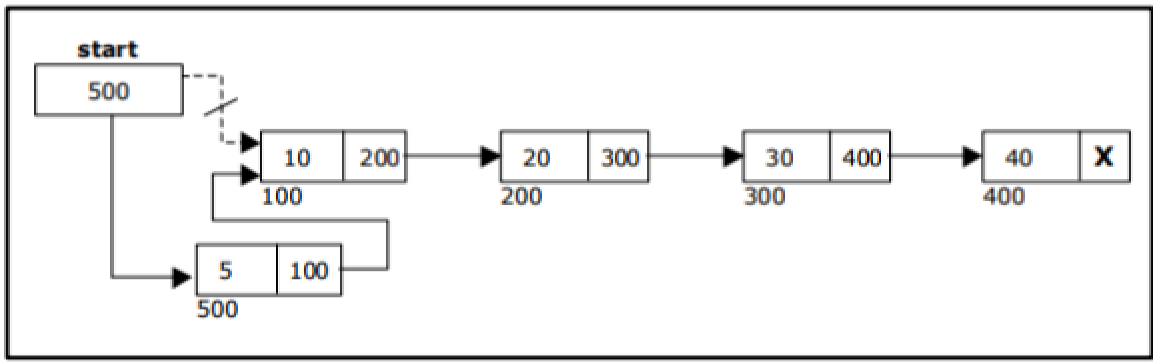
We can use the following steps to insert anew node at beginning of the single linked list...

Step1-Create a **newNode** with given value.

Step2-Check whether list is **Empty** (**start**==**NULL**)

Step3-If it is **Empty** then, set newNode→**next** =**NULL** and **start**=**newNode**.

Step4-IfitisNotEmptythen,set **newNode→next**=**start**and**start**=**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...

Step1-Create a **new Node** with given value and **newNode→next**as**NULL**.

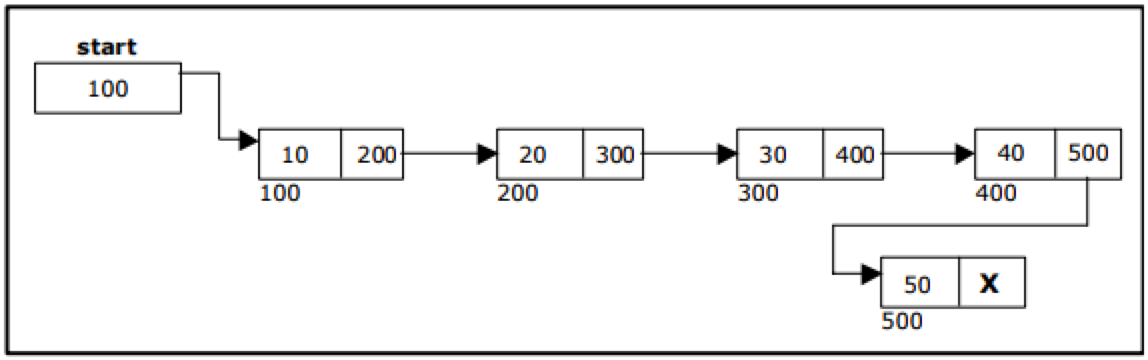
Step2-Check whether list is **Empty** (**start**==**NULL**).

Step3-If it is **Empty** then, set**start**=**newNode**.

Step4- If it is **NotEmpty** then defines a node pointer **temp** and initialize with **start**.

Step5-Keep moving the **temp** to its next node until it reaches to the last node in the list (until **temp** →**next** is not equal to **NULL**).

Step6-Settemp → next=newNode.



Inserting at middle of the list:

Step1-Create a new Node with given value

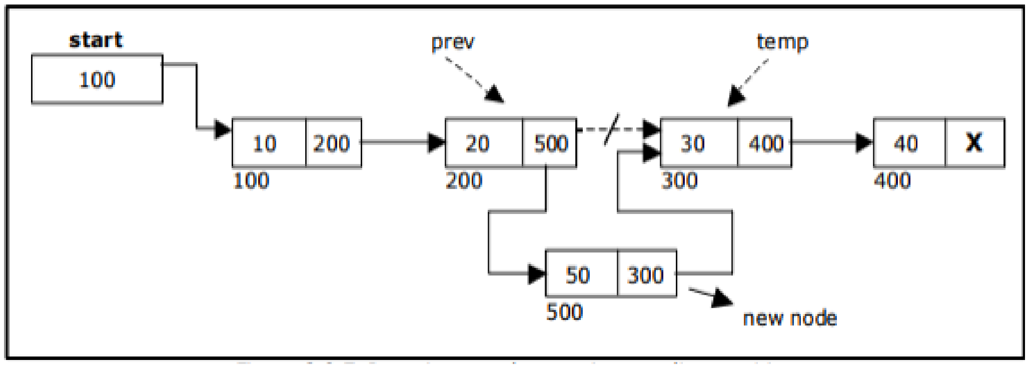
Step2-Checkwhetherlist is **Empty** (**start**==**NULL**)

Step 3 - If it is **Empty** then, set **newNode** → **next** = **NULL** and **start** = **newNode**.

Step4- If it is **NotEmpty** then defines a node pointer **temp** and initialize with **start**.

Step5-Keep moving the **temp** to its next node until it reachest other node after which we want to insert the newNode

Step6-Finally,Setp->next=newnode, newnode->next=temp



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...

Step1-Checkwhetherlist is **Empty** (**start**==**NULL**)

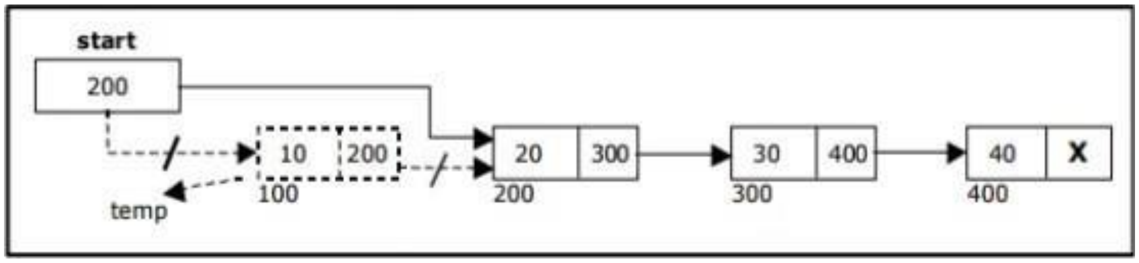
Step2-If it is **Empty** then, display**' List is Empty!!!Deletion is not possible'** and terminate the function.

Step3-Ifit is **Not Empty** then, define a Node pointer **'temp'** and initialize with **start**.

Step 4 - Check whether list is having only one node (**temp →next**==**NULL**)

Step5- If it is **TRUE** then set **start**=**NULL** and delete **temp** (Setting **Empty** list conditions)

Step6- If it is **FALSE** then set **start**=**temp→next**, and delete **temp (free(temp))**.



Deleting from End of the list:

We can use the following steps to delete an odefrom end of the single linked list...

Step1-Checkwhetherlist is **Empty**(**start**==**NULL**)

Step2- If it is **Empty** then, display **'List is Empty!!!Deletion is not possible'** and terminate the function.

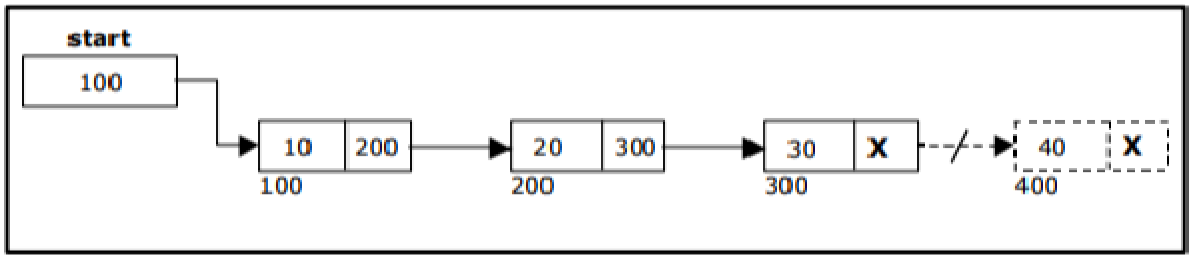
Step 3 - If it is **Not Empty** then, define two Node pointers **'temp'** and'**temp1'** and initialize' **temp**' with **start**.

Step4 –Check whether list has only one Node(**temp→ next** ==**NULL**)

Step5- If it is **TRUE**. Then, set **start**=**NULL** and delete **temp**. And terminate the function. (Setting **Empty** list condition)

Step6- If it is **FALSE**. Then, set '**temp1=temp** 'and move **temp** to its next node. Repeat the same until it reaches to the last node in the list.(until**temp1 →next** ==**NULL**)

Step7-Finally, Set **temp1→ next**=**NULL** and delete **temp(free(temp))**.



Deleting a Specific Node from the list:

Step1-Create a new Node with given value

Step2-Checkwhetherlist is **Empty**(**start**==**NULL**)

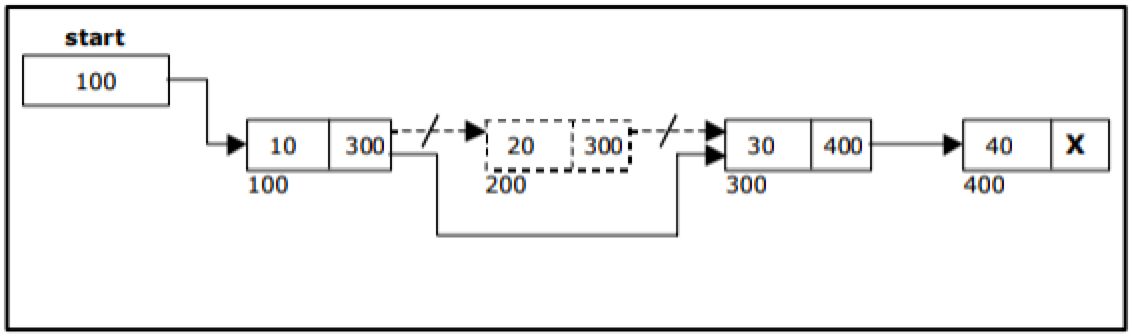
Step 3 - If it is **Empty** then, set **newNode** → **next** = **NULL** and **start** = **newNode**.

Step4- If it is **Not Empty** then defines an order pointer **temp** and initialize with **start**.

Step5-Keep moving the **temp** to its next node until it reaches to the node after which we want to delete the Node

Step6-Finally,Setp->next=temp->next

Step7-deletet emp(free(temp)).



Traversing:

- Assign the address of start pointer to at emp pointer.
- Display the information from the data field of each node.
- The function traverse() is used for traversing and displaying the information stored in the list from left to right.

2B) Write a C program to implement Circular linked list
i) Insertion ii) Deletion. iii) Display

Circular Singly linked list is similar to Single linked list, but the last node of the list contains a pointer o the first node of the list.

Inserting At End of the list:

We can use the following steps to insert anew node at end of the single linked list...

Step1-Create a **newNode** with given value and **newNode→next** as **NULL**.

Step2-Check whether list is **Empty** (**start==NULL**).

Step3-If it is **Empty** then, set **start=new Node**.

Step4- If it is **Not Empty** then defines an order pointer **temp** and initialize with **start**.

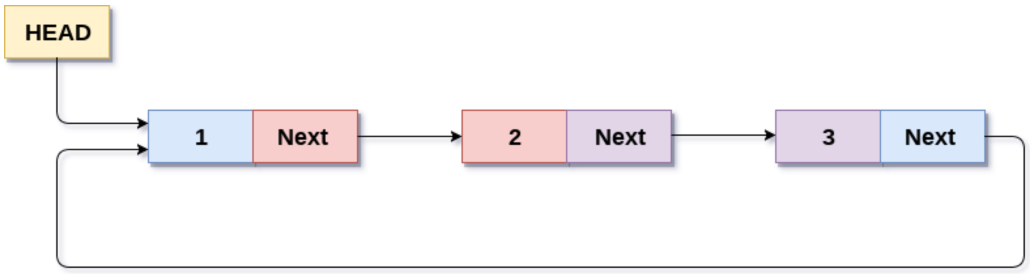
Step5-Keep moving the **temp** to its next node until it reaches to the last node in

The list (until **temp →next** is not equal to (**start**).

Step6-Set temp →next =new Node.

Step7-SetnewNode→next =**Start**

We can have circular singly linked list as well as circular doubly linked list.

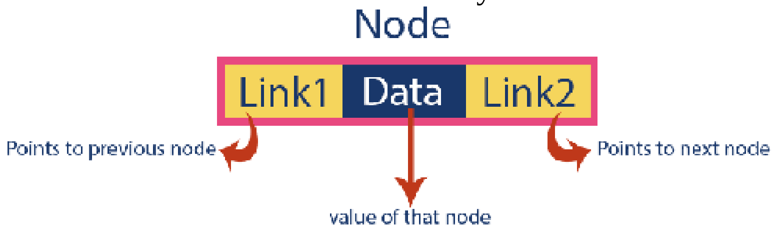


Circular Singly Linked List

Q3.A) Write a C program to implement doubly linked list
i) Insertion ii)Deletion iii)Display

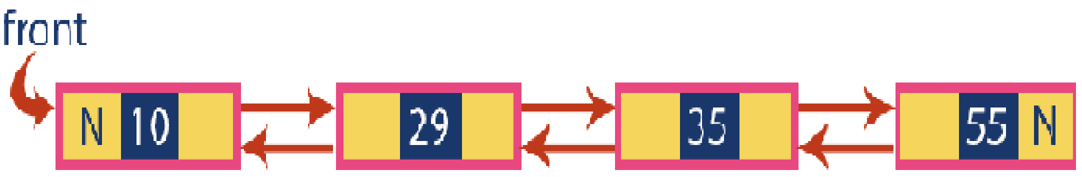
Double linked list is a sequence of elements in which every element has links to its previous element and next element in the sequence.

Everynodeinadoublelinkedlistcontainsthreefieldsandtheyareshowninthefollowingfigure...



Here,**'link1'**fieldisusedtostoretheaddressofthepreviousnodeinthesequence,**'link2'**fieldis used to store the address of the next node in the sequence and **'data'** field is used to store the actual value of that node.

Example:



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 per formed 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 anew node at beginning of the double linked list...

Step1-Create a **newNode** with given value and **newNode→previous** as **NULL**.

Step2-Check whether list is **Empty(start==NULL**)

Step3- If it is **Empty** then,assign **NULL** to **new Node→next** and **newNode** to **start**.

Step4- If it is **notEmpty** then, assign **start** to **newNode→next**and**newNode**to**start**

Inserting At End of the list

We can use the following steps to insert anew node at end of the double linked list...

Step1-Create a **new Node** with given value and **newNode→next**as**NULL**.

Step2-Check whether list is **Empty** (**start**==**NULL**)

Step3-If it is **Empty**, then assign **NULL** to **new Node→previous** and **new Node** to **head**.

Step4- If it is **not Empty**, then, define an ode pointer **temp** and initialize with **head**.

Step5-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 **new Node** to **temp** → **next** and **temp** to **new Node→previous**

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

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

Step1-Create a **newNode** with given value.

Step2-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**.

Step4- If it is **notEmpty** then, defines two node pointers **temp1**&**temp2** and initialize

temp1with**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 then ode 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 tothelastnodethendisplay'**Givennodeisnotfoundinthelist!!!Insertion not possible!!!**' and terminate the function. Otherwise move the **temp1**to next node.

Step 7 – Assign temp1→next to temp2,newNodeto temp→ next,

temp1tonewNode→previous,temp2 to newNode→ next

newNodetotemp2→previous.

In a Double 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 an ode from beginning of the single linked list...

Step1-Check whet her list is **Empty** (**start==NULL**)

Step2-If it is **Empty** then, display **'List is Empty!!!Deletion is not possible'** and terminate the function.

Step3-Ifit is **NotEmpty** then, define a Node pointer **'temp'** and initialize with **start**.

Step 4 - Check whether list is having only one node (temp → next ==NULL)

Step 5 - If it is TRUE then set start= NULL and delete temp (Setting Empty list conditions)

Step 6 - If it is FALSE then set start= temp → next, and delete temp (free (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 (start == 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 'temp' and 'temp1' and initialize 'temp' with start.

Step 4 - Check whether list has only one Node (temp → next == NULL)

Step 5 - If it is TRUE. Then, set start = NULL and delete temp. And terminate the function. (Setting Empty list condition)

Step 6 - If it is FALSE. Then, set 'temp1 = temp ' and move temp 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 temp1 → next = NULL and delete temp (free (temp)).

Deleting a Specific Node from the list:

Step 1 - Create a newNode with given value

Step 2 - Check whether list is Empty (start == NULL)

Step 3 - If it is Empty then, set newNode → next = NULL and start = newNode.

Step 4 - If it is Not Empty then, define a node pointer temp and initialize with start.

Step 5 - Keep moving the temp to its next node until it reaches to the node after which we want to delete the Node

Step 6-Finally, Set p->next=temp->next temp->next->prev=p

Step 7-delete temp (free (temp)).

Traversing:

- Assign the address of start pointer to a temp pointer.
- Display the information from the data field of each node.
- The function traverse () is used for traversing and displaying the information stored in the list from left to right.

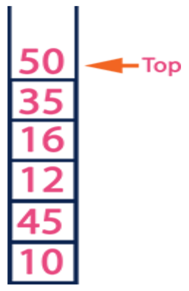
3B.i) Write C programs to implement Stack ADT using Array

Stack is a linear data structure.

"A Collection of similar data items in which both insertion and deletion operations are performed based on LIFO principle".

Example:

If we want to create a stack by inserting 10, 45,12,16,35 and 50. Then 10 becomes the bottommost element and 50 is the top most element. The last inserted element50 is at Top of the stack as shown in the image below...



The following operations are performed on the stack...

1. Push (To insert an element on to the stack)
2. Pop (To delete an element from the stack)
3. Display (To display elements of the stack)

Stack data structure can be implemented in two ways. They are as follows...

1. Using Array
2. Using Linked List

When stack is implemented using array, that stack can organize only limited number of elements. When stack is implemented using linked list, that stack can organize unlimited number of elements.

Stack Using Arrays:

A stack data structure can be implemented using one dimensional array. But stack implemented using array stores only fixed number of data values. This implementation is very simple. Just define a one dimensional array of specific size and insert or delete the values into that array by using **LIFO principle** with the help of a variable called **'top'**. Initially top is set to -1. Whenever we want to insert a value into the stack, increment the top value by one and then insert. Whenever we want to delete a value from the stack, then delete the top value and decrement the top value by one.

Stack Operations using Arrays:

A stack can be implemented using array as follows...

Before implementing actual operations, first follow thebe low steps to create an empty stack.

Step1-Include all the **header files** which are used in the program and define a constant **'SIZE'** with specific value.

Step2-Declare all the **functions** used in stack implementation.

Step3-Createaonedimensionalarraywithfixed size (**int stack[SIZE]**)

Step4-Definea integer variable **'top'** and initialize with **'-1'**.(**int top=-1**)

Step5-Inmain method, display menu with list of operations and make suitable function calls to perform operation selected by the user on the stack.

Push (value)-Inserting value into the stack:

Ina stack, push() is a function used to insert an element into the stack. Ina stack, the new element is always inserted at **top** position. Push function takes one integer value as parameter and inserts that value into the stack. We can use the following steps to push an element on to the stack...

- **Step 1-**Checkwhether**stack** is **FULL**.(**top==SIZE-1**)
- **Step2-**If it is **FULL**, then display **"Stack is FULL!!! Insertion is not possible!!!"** and terminate the function.
- **Step3-**IfitisNOTFULL, then increment **top** value by one(**top++**) and set stack[top] to value(**stack[top] =value**).

Pop()-Delete a value from the Stack:

In a stack, **pop()** is a function used to delete an element from the stack. In a stack, the element is always deleted from **top** position. Pop function does not take any value as parameter. We can use the following steps to pop an element from the stack...

Step1-Check whether **stack** is **EMPTY**. (**top==-1**)

Step 2 - If it is **EMPTY**, then display **"Stack is EMPTY!!! Deletion is not possible!!!"** and terminate the function.

Step3-Ifitis**NOTEMPTY**, then delete **stack[top]**and decrement **top** value by one(**top--**).

Display ()-Displays the elements of a Stack:

We can use the following steps to display the elements of a stack...

Step1-Check whether stack **is** EMPTY**. (**top==-1**)**

Step2-If it is EMPTY**, then display**" Stack is EMPTY!!!" **and terminate the function.**

Step3- If it is **NOT EMPTY**, then define variable '**I** 'and initialize with top. Display **stack[i]** value and decrement **i** value by one (**i--**).

Step3-Repeat above step until **I** value becomes '0'.

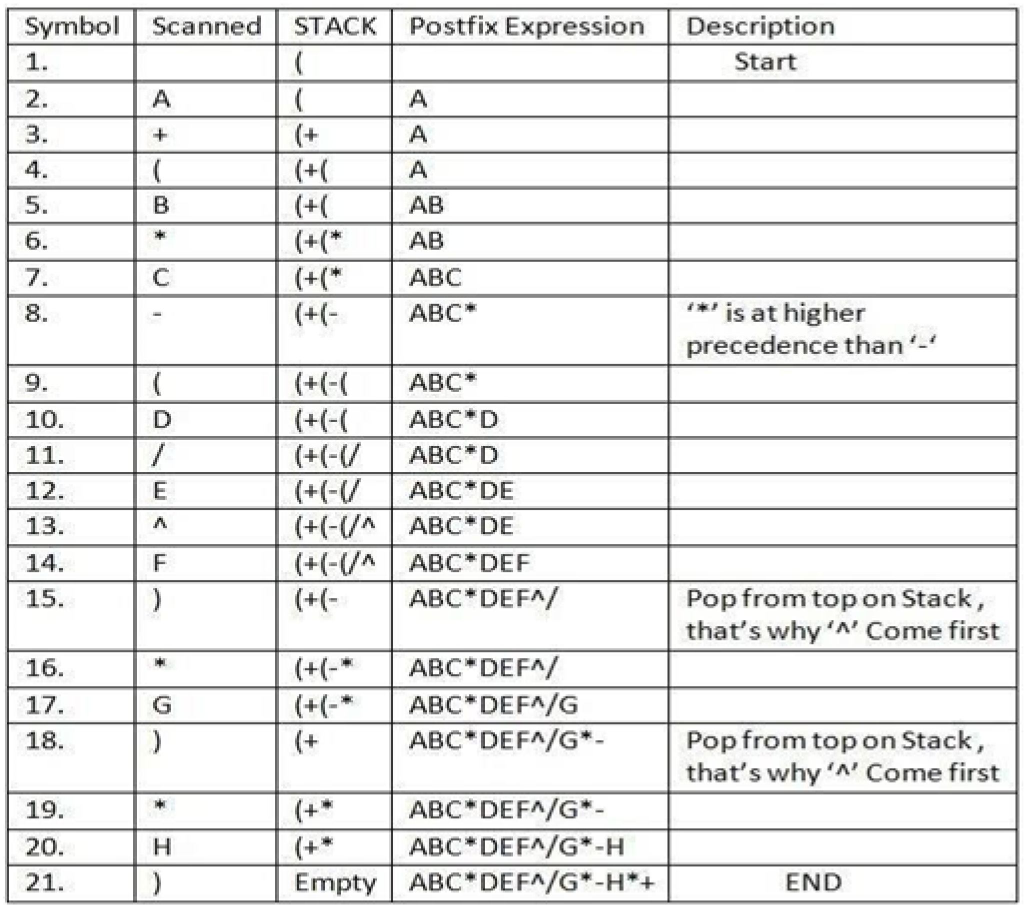
Lab4:
A.Write a C program that uses stack operations to convert a given infix expression in to its postfix equivalent. (Display the role of stack).

Infix to Postfix Conversion using Stack Data Structure
To convert Infix Expression into Postfix Expression using a stack data structure, we can use the following steps...

1. Read all the symbols one by one from left to right in the given Infix Expression.
2. If the reading symbol is operand, then directly print it to the result(Output).
3. If the reading symbol is left parenthesis'(',thenPushitontotheStack.
4. If the reading symbol is right parentheses)',then Pop all the contents of stack until respective left parenthesis is popped and print each popped symbol to the result.

If the reading symbol is operator (+,-,*,/etc.,),then Push it on to the Stack.However,first pop the operators which are already on the stack that have higher or equal precedence than current operator and print them to the result

Example: ConsiderthefollowingInfixExpression...
InfixExpression:**A+(B*C-(D/E^F)*G)*H**,where^isanexponentialoperator.



Symbol	Scanned	STACK	Postfix Expression	Description
1.		(		Start
2.	A	(	A	
3.	+	(+	A	
4.	(	(+(	A	
5.	B	(+(	AB	
6.	*	(+(*	AB	
7.	C	(+(*	ABC	
8.	-	(+(-	ABC*	'*' is at higher precedence than '-'
9.	(	(+(-(	ABC*	
10.	D	(+(-(	ABC*D	
11.	/	(+(-(/	ABC*D	
12.	E	(+(-(/	ABC*DE	
13.	^	(+(-(/^	ABC*DE	
14.	F	(+(-(/^	ABC*DEF	
15.	)	(+(-	ABC*DEF^/	Pop from top on Stack, that's why '^' Come first
16.	*	(+(-*	ABC*DEF^/	
17.	G	(+(-*	ABC*DEF^/G	
18.	)	(+	ABC*DEF^/G*-	Pop from top on Stack, that's why '^' Come first
19.	*	(+*	ABC*DEF^/G*-	
20.	H	(+*	ABC*DEF^/G*-H	
21.	)	Empty	ABC*DEF^/G*-H*+	END

4B.Write a C program for Evaluation of postfix expression.

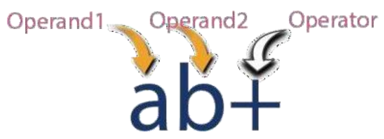
Postfix Expression Evaluation

A postfix expression is a collection of operators and operands in which the operator is placed after the operands. That means, in a postfix expression the operator follows the operands.

Postfix Expression has following general structure...

Operand1 Operand2 Operator

Example:

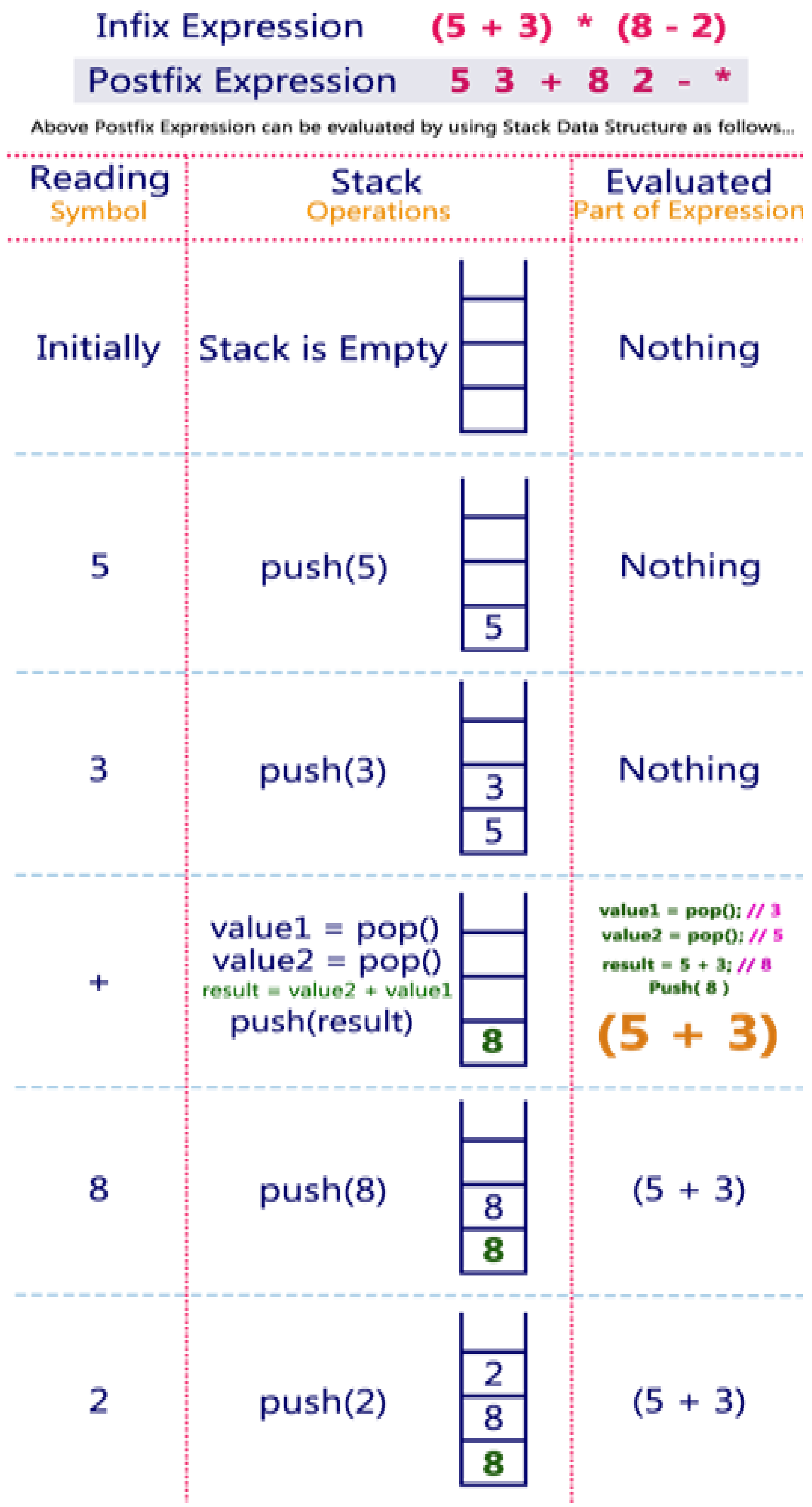


Postfix Expression Evaluation using Stack Data Structure

A postfix expression can be evaluated using the Stack data structure. To evaluate a postfix expression using Stack data structure we can use the following steps...

1. Read all the symbols one by one from left to right in the given Postfix Expression
2. If the reading symbol is operand, then push it on to the Stack.
3. If the reading symbol is operator (+ , - , * , / etc.,), then perform TWO pop operations and store the two popped operands in two different variables (operand1 and operand2). Then perform reading symbol operation using operand1 and operand2 and push result back on to the Stack.
4. Finally! Perform a pop operation and display the popped value as final result.

Example:



Infix Expression (5 + 3) * (8 - 2)

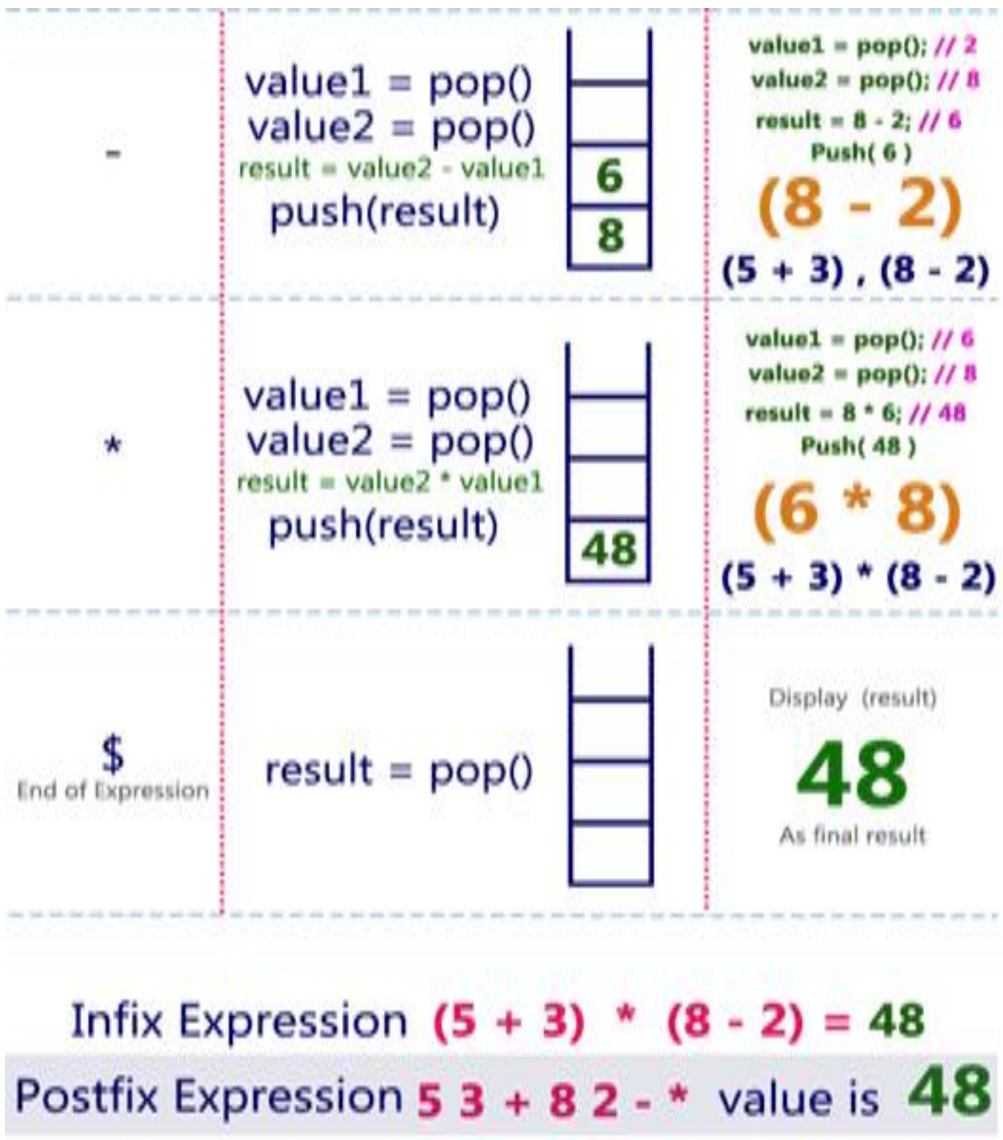
Postfix Expression 5 3 + 8 2 - *

Above Postfix Expression can be evaluated by using Stack Data Structure as follows...

Reading Symbol	Stack Operations		Evaluated Part of Expression
Initially	Stack is Empty		Nothing
5	push(5)	5	Nothing
3	push(3)	3 5	Nothing

+	value1 = pop()
	value2 = pop()
	result = value2 + value1
	push(result)
		8	value1 = pop(); // 3
			value2 = pop(); // 5
			result = 5 + 3; // 8
			Push(8)
			(5 + 3)

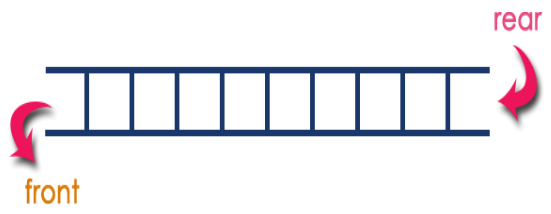
8	push(8)	8 8	(5 + 3)
2	push(2)	2 8 8	(5 + 3)


-
value1 = pop()
value2 = pop()
result = value2 - value1
push(result)
6
8
value1 = pop(); // 2
value2 = pop(); // 8
result = 8 - 2; // 6
Push(6)
(8 - 2)
(5 + 3) , (8 - 2)
*
value1 = pop()
value2 = pop()
result = value2 * value1
push(result)
48
value1 = pop(); // 6
value2 = pop(); // 8
result = 8 * 6; // 48
Push(48)
(6 * 8)
(5 + 3) * (8 - 2)
$
End of Expression
result = pop()
Display (result)
48
As final result
Infix Expression (5 + 3) * (8 - 2) = 48
Postfix Expression 5 3 + 8 2 - * value is 48

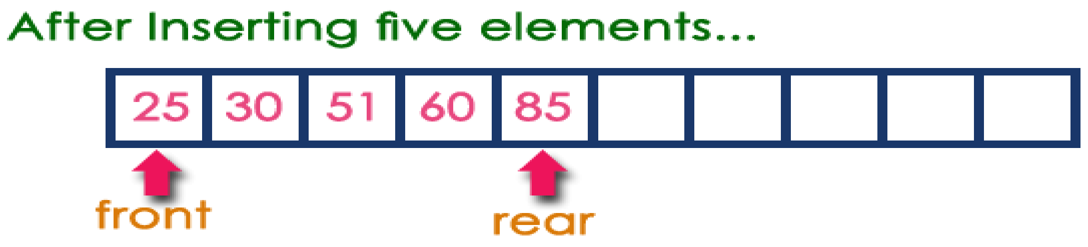
Lab5: Write C programs to implement Queue ADT using

i) Arrays ii)Linked Lists

Queue is a linear data structure in which the insertion and deletion operations are performed at two different ends. In a queue data structure, adding and removing of elements are performed at two different positions. The insertion is performed at one end and deletion is performed at other end. In a queue data structure, the insertion operation is performed at a position which is known as '**rear** 'and the deletion operation is performed at a position which is known as '**front**'. In Queue data structure, the insertion and deletion operations are performed based on **FIFO(First In First Out)** principle.



In a queue data structure, the insertion operation is performed using a function called "**enQueue()**" and deletion operation is performed using a function called" **deQueue()**".
Queue after inserting 25,30, 51, 60and 85.



Operations on a Queue:

The following operations are performed on a queue data structure...

1. enQueue (value)-(To insert an element in to the queue)
2. **deQueue()-(To delete an element from the queue)**
3. **display()-(To display the elements of the queue)**

Queue data structure can be implemented in two ways. They are as follows...

1. Using Arrays
2. Using Linked Lists

When a queue is implemented using array, that queue can organize only limited number of elements. When a queue is implemented using linked list, that queue can organize unlimited number of elements.

Queue Data structure Using Arrays:

A queue data structure can be implemented using one dimensional array. The queue implemented using array stores only fixed number of data values. The implementation of queue data structure using array is very simple. Just define a one dimensional array of specific size and insert or delete the values into that array by using **FIFO (First In First Out) principle** with the help of variables **'front'** and '**rear**'. Initially both '**front**' and '**rear**' are set to -1. Whenever, we want to insert a new value into the queue, increment '**rear**' value by one and then insert at that position. Whenever we want to delete a value from the queue, then delete the element which is at 'front' position and increment 'front' value by one.

Queue Operations using Arrays:

Queue data structure using array can be implemented as follows...

Before we implement actual operations, first follow the below steps to create an empty queue.

Step1- Include all the **header files** which are used in the program and define a constant **'SIZE'** with specific value.

Step2-Declareall the **user defined functions** which are used in queue implementation.

Step3- Create a one dimensional array with above defined SIZE (**int queue[SIZE]**)

Step4-Define two integer variables **'front'** and '**rear** 'and initialize both with **'-1'**. (**int front =-1, rear= -1**)

Step 5 - Then implement main method by displaying menu of operations list and make suitable function calls to perform operation selected by the user on queue.

Enqueue (value)-Inserting value into the queue:

In a queue data structure, enQueue() is a function used to insert a new element into the queue. In a 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 queue. We can use the following steps to insert an element into the queue...

Step1-Check whether **queue** is **FULL**.(**rear==SIZE-1**)
Step2-If it is **FULL**, then display **"Queue is FULL !!! Insertion is not possible!!!"** and terminate the function.
Step3-If it is **NOT FULL**, then increment **rear** value by one(**rear++**) and set **queue[rear]=value**.

deQueue()-Deleting a value from the Queue:

In a queue data structure, deQueue() is a function used to delete an element from the queue. In a queue, the element is always deleted from **front** position. The deQueue() function does not take any value as parameter. We can use the following steps to delete an element from the queue...

Step1-Checkwhether**queue**is**EMPTY**.(**front==rear**)

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 increment the **front** value by one(**front ++**). Then display **queue[front]** as deleted element. Then checkwhetherboth**front**and**rear**areequal(**front**==**rear**),ifit**TRUE**,thenset both **front** and **rear**to'**-1**'(**front**=**rear**=**-1**).

Display()-Displays the elements of a Queue:

We can use the following steps to display the elements of a queue...

Step1-Checkwhether**queue**is**EMPTY**.(**front==rear**)

Step2-Ifitis**EMPTY**,thendisplay**"QueueisEMPTY!!!"** and terminate the function.

Step3-If it is **NOTEMPTY**, then define an integer variable '**i**' and set '**i=front+1**'.

Step4-Display '**queue[i]**' value and increment '**i** 'value by one (**i++**). Repeat the same until '**i**' value reaches to **rear**(**i** <=**rear**)

6.Program to implement Binary search tree

i) Insertion **ii)deletion** **iii)Traversals**

Every Binary Search Tree is a binary tree but all the Binary Trees need not to be binary search trees.

The following operations are performed on a binary Search tree...

- Search
- Insertion
- Deletion

Search Operation in BST:

In a binary search tree, the search operation is performed with **O(logn)** time complexity. The search operation is performed as follows...

Step1: Read the search element from the user

Step2: Compare,the search element with the value of root node in the tree.

Step3: If both are matching, then display "Given node found!!!"and terminate the function

Step4: If both are not matching, then check whether search element is smaller or larger than that node value.

Step5: If search element is smaller, then continue the search process in left sub tree.

Step6: If search element is larger, then continue the search process in right sub tree.

Step 7: Repeat the same until we found exact element or we completed with a leaf node

Step8: If we reach to the node with search value, then display "Element is found" and terminate the function.

Step9: If we reach to a leaf node and it is also not matching, then display "Element not found" and terminate the function.

Insertion Operation in BST:

In a binary search tree, the insertion operation is performed with **O(logn)** time complexity. In binary search tree, new node is always inserted as a leaf node.

The insertion operation is performed as follows...

Step 1: Create a new Node with given value and set its **left** and **right** to **NULL**.

Step2: Check whether tree is Empty.

Step3: If the tree is **Empty**, then set **root** to **new Node**.

Step 4: If the tree is **Not Empty**, then check whether value of new Node is **smaller** or **larger** than the node (here it is root node).

Step 5: If new Node is **smaller** than **or equal** to the node, then move to its **left** child. If new Node is **larger** than the node, then move to its **right** child.

Step 6: Repeat the above step until we reach to a **leaf** node (e.i., reach to NULL).

Step7: After reaching a leaf node, then insert the new Node as **left child** if new Node is **smaller or equal** to that leaf else insert it as **right child**.

Deletion Operation in BST:

In a binary search tree, the deletion operation is performed with **O(log n)** time complexity. Deleting a node from Binary search tree has following three cases...

Case1:Deletinga Leaf node(A node with no children)
Case2:Deleting a node with one child
Case3:Deleting a node with two children

Case1: Deleting a leaf node:

We use the following steps to delete a leaf node from BST...

Step1:Find the node to be deleted using **search operation**

Step2: Delete the node using **free** function (If it is a leaf) and terminate the function.

Case2:Deleting a node with one child:

We use the following steps to delete a node with one child from BST...

Step1:Find the node to be deleted using **search operation**
Step2: If it has only one child, then create a link between its parent and child nodes.
Step 3: Delete the node using **free** function and terminate the function.

Case3: Deleting a node with two children:

We use the following steps to delete a node with two children from BST...

Step1:Find the node to be deleted using **search operation**

Step2: Ifithastwochildren,thenfindthe**largest**nodeinits**leftsubtree** (OR)

The **smallest** node in its **right sub tree**.

Step3:Swap both **deleting node** and node which found in above step.

Step4: Then,check whether deleting node came to **case 1** or **case 2** else go to **Step2**

Step5: If it comes to **case1**, then delete using case1logic.

Step6: If it comes to **case2**, then delete using case2logic.

Step7: Repeatthesame process until node is deleted from the tree.

Tree Traversals:

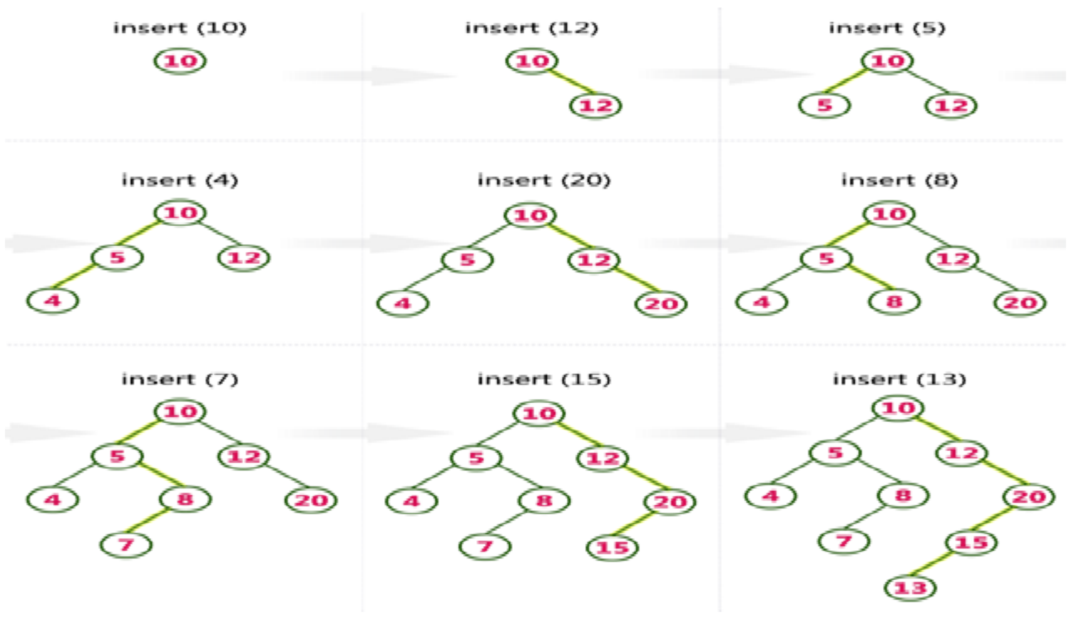
In **Pre-order** the parent node visited first then the left child node and at last the right child node.

In**In-order**theleftchildnodevisitedfirstthenparentnodeandatlasttherightchildnode.

In **Post-order** the left child node visited first then the right child node and in last parent node.

Example:

ConstructaBinarySearchTreebyinsertingthefollowingsequenceofnumbers...
10,12,5,4,20,8,7,15and13



7. Write a C program to implement binary search tree Non-recursively traversals

(i) Pre-Order
(ii) Post–Order(iii)In-Order

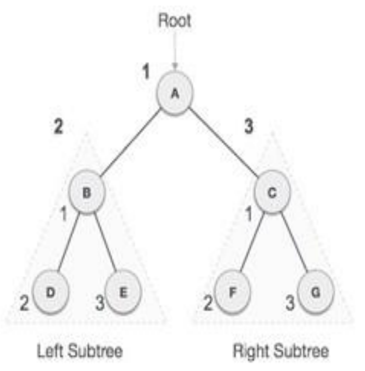
Tree traversal is the process of visiting each node in the tree exactly once. Visiting each node in a graph should be done in a systematic manner. If searcher sultana visit to all the vertices, it is called a traversal. There are basically three traversal techniques for a binary tree that are,

- ❖ Preorder traversal
- ❖ Inorder traversal
- ❖ Postorder traversal

Preorder traversal:

To **traverse a binary tree in preorder**, following operations are carried out:

1. Visit the root.
2. Traverse the left sub tree of root.
3. Traverse the right sub tree of root



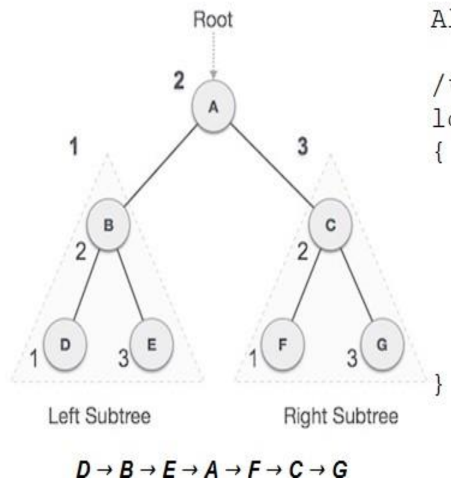
A → B → D → E → C → F → G

```
Algorithm preorder(t)
/*t is a binary tree. Each node of t has three fields:
lchild, data, and rchild.*/
{
If t! =0 then
{
        Visit(t);
        Preorder(t->lchild);
        Preorder(t->rchild);
 }
```

In-order traversal:

To traverse a binary tree in in-ordertraversal,following operations are carried out:

- ○ Traverse the left most sub tree.
- ○ Visit the root.
- ○ Traverse the right most sub tree.



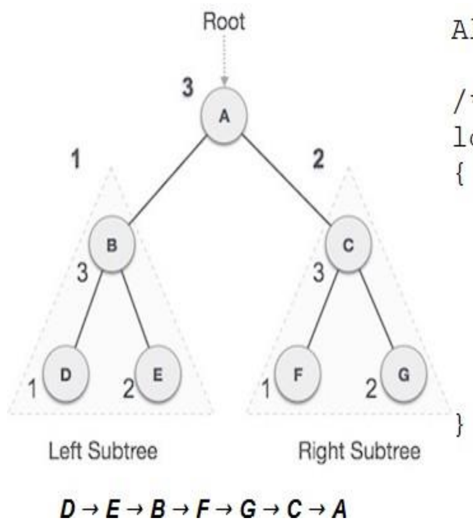
```
Algorithm inorder(t)

/*t is a binary tree. Each node of t has three fields:
lchild, data, and rchild.*/
{
     If t! =0 then
     {
          Inorder(t->lchild);
          Visit(t);
          Inorder(t->rchild);
     }
}
```

Post-order traversal:

Totraverseabinarytreeinpost-ordertraversal,followingoperationsarecarriedout:

1. Traverse the left sub tree of root.
2. Traverse the right sub tree of root.
3. Visit the root.

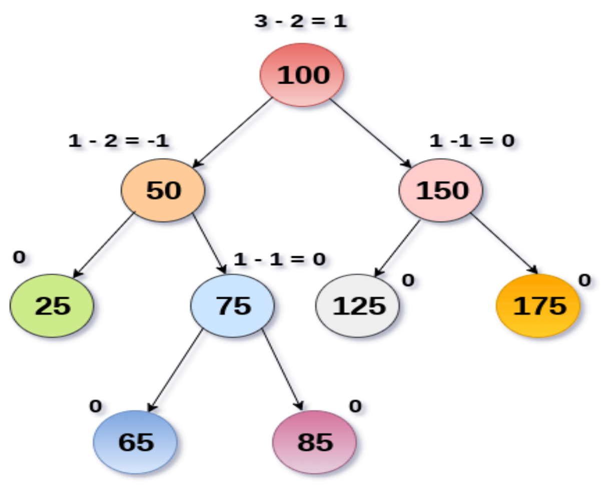


```
Algorithm postorder(t)

/*t is a binary tree .Each node of t has three fields:
lchild, data, and rchild.*/
{
     If t! =0 then
     {
          Postorder(t->lchild);
          Postorder(t->rchild);
          Visit(t);
     }
}
```

8(A) Write a C Program to Check if a Given Binary Tree is an AVL Tree or Not

If a binary search tree has a balance factor, then it is an **AVL (Adelso-Velskii and Landis)** tree. This means that in an AVL tree the difference between left sub tree and



AVL Tree

right sub tree height is at most one.

To check if a Binary tree is balanced we need to check three conditions:

1. The absolute difference between heights of left and right sub trees at any node should be less than 1.
2. For each node, its left sub tree should be a balanced binary tree.
3. For each node, its right sub tree should be a balanced binary tree.

We will need a function that can calculate the height of the tree. One way to do this is to write a separate function for calculating the height and call it every time height is needed. This is going to be computationally inefficient.
The better way to implement this will be returning height in the same function. For each node, we will return -1if it is not balanced and the height to ift hat node/sub tree if it is balanced.

The algorithm is as follows:

- If node==null→return0
- Check left sub tree. If not balanced→return-1
- Check right sub tree. If not balanced→return-1
- The absolute between heights of left and right subtrees.Ifit isgreaterthan1→return-1.
- If the tree is balanced→ return height

8(B) Write a C program to find height to binary tree

There are two conventions to define the height of a Binary Tree

- Number of nodes on the longest path from the root to the deepestnode.
- Number of edges on the longest path from the root to the deepest node.

Start the algorithm by taking the root node as an input. Next, we calculate the height of the left and right child nodes of the root. In case the root doesn't have any child node, we return the height ofthetreeas0.

Recursively call all the nodes from the left and right sub tree of the root node to calculate the height of a binary tree. Finally, when calculate the height of the left and right sub tree, take the maximumheightamongbothandaddonetoit.Thenumberreturnbythealgorithmwouldbetheheightofthebinary tree.

Procedure Binary tree Height:

```
If root==NUL Lthen
        Return0;
Else
Left Tree height=Binary tree height(Root →left) ;
Right Tree height=Binary tree height(Root→Right);
Return Max(Left Tree height, Right Tree height)+1;
```

8(C Write a C program to count the number of leaf nodes in a tree

Procedure for count the number of leaf nodes in a tree:

1) If the node is null return0,this is also the base case of our recursive algorithm
2) If a leaf node is encountered the nreturn1
3) Repeat the process with left and right sub tree
4) Return the sum of leaf nodes from both left and right sub tree

An iterative algorithm to get the total number of leaf nodes of binary tree:

The recursive algorithm for counting leaf nodes was pretty easy, so is the iterative algorithm aswell.Similar to iterative In Order traversal example, we have used a Stack to traverse the binary tree.

Steps of the iterative algorithm to get a total number of leaf nodes of a binary tree:

1) If the root is null then return zero.
2) Start the count with zero
3) Push the root into Stack
4) Loop until Stack is not empty
5) Pop the last node and push left and right children of the last node if they are not null.
6) Increase the count

At the end of the loop, the count contains the total number of leaf nodes.

9A.WriteaC Program to implement Breadth First Search (BFS)

Algorithm for BFS:

Step1: Define a Queue of size total number of vertices in the graph.

Step2: Select any vertex as **starting point** for traversal. Visit that vertex and insert it into the Queue.

Step3: Visitallthe**adjacent**verticesofthevertexwhichisatfrontoftheQueuewhichisnotvisited and insert them into the Queue.

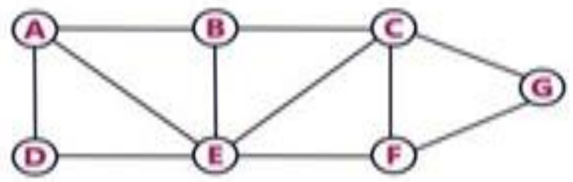
Step4: When there is no new vertex to be visit from the vertex at front of the Queue then delete that vertex from the Queue.

Step5: Repeat step 3and4until queue becomes empty.

Step6: WhenqueuebecomesEmpty,thenproducefinalspanningtreebyremoving unused edges from the graph

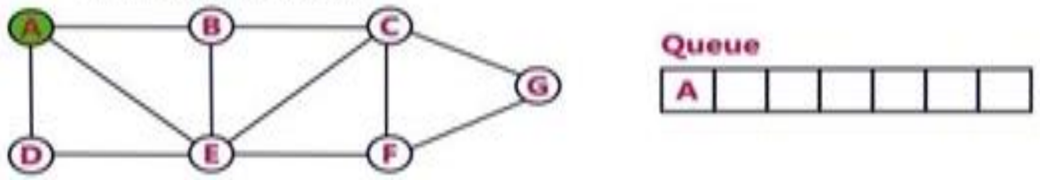
Example:

Consider the following example graph to perform BFS traversal



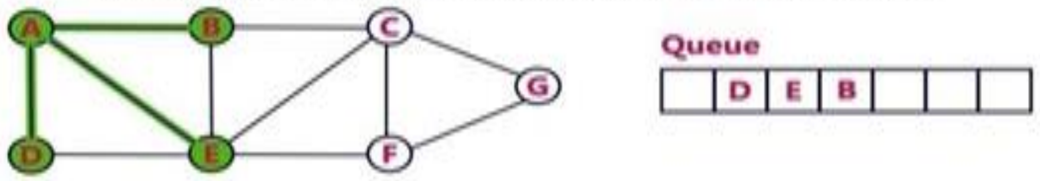
Step 1:

- Select the vertex **A** as starting point (visit **A**).
- Insert **A** into the Queue.



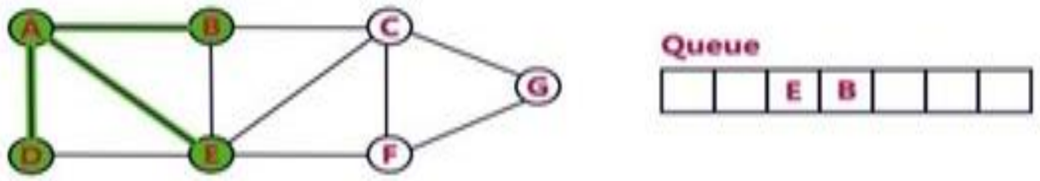
Step 2:

- Visit all adjacent vertices of **A** which are not visited (**D**, **E**, **B**).
- Insert newly visited vertices into the Queue and delete A from the Queue..



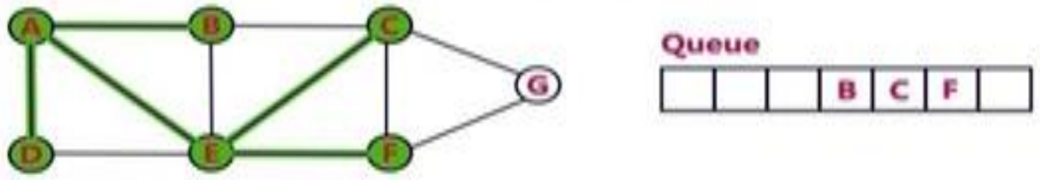
Step 3:

- Visit all adjacent vertices of **D** which are not visited (there is no vertex).
- Delete D from the Queue.



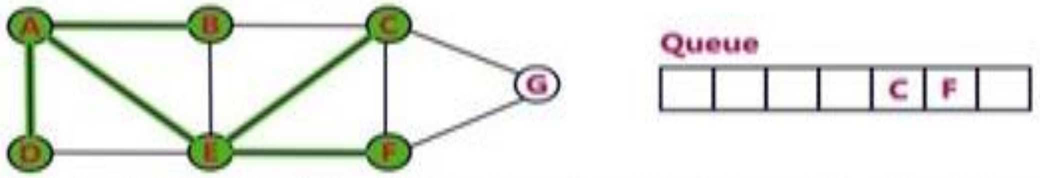
Step 4:

- Visit all adjacent vertices of **E** which are not visited (**C**, **F**).
- Insert newly visited vertices into the Queue and delete E from the Queue.



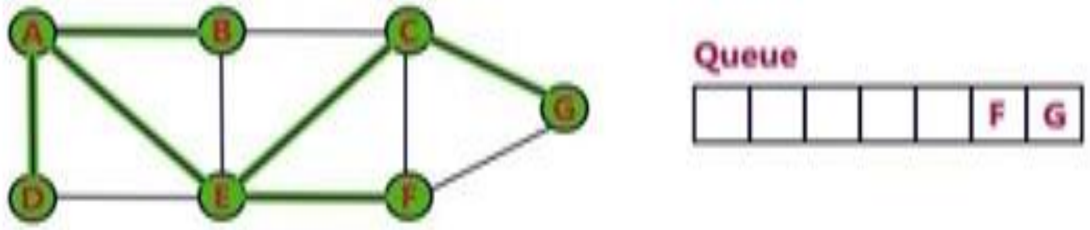
Step 5:

- Visit all adjacent vertices of **B** which are not visited (**there is no vertex**).
- Delete **B** from the Queue.



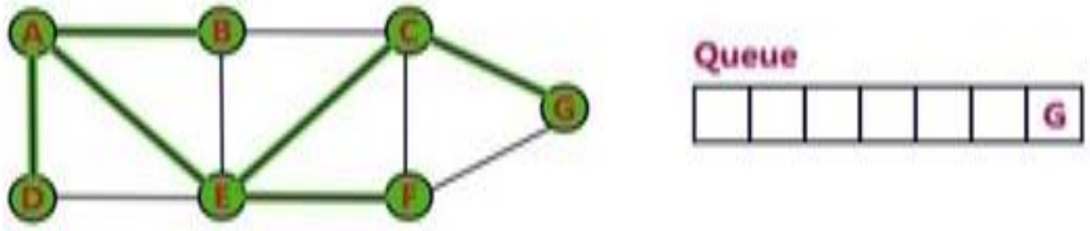
Step 6:

- Visit all adjacent vertices of **C** which are not visited (**G**).
- Insert newly visited vertex into the Queue and delete **C** from the Queue.



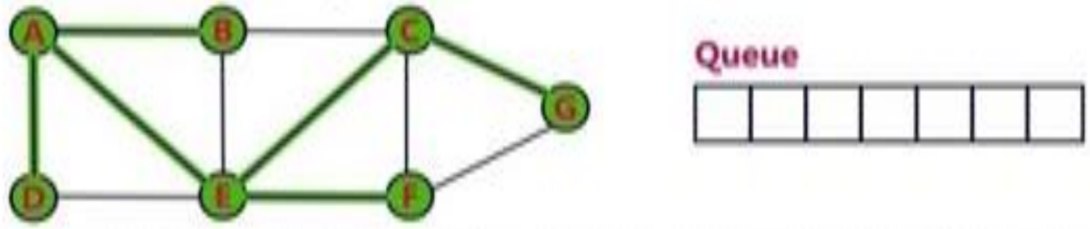
Step 7:

- Visit all adjacent vertices of **F** which are not visited (**there is no vertex**).
- Delete **F** from the Queue.

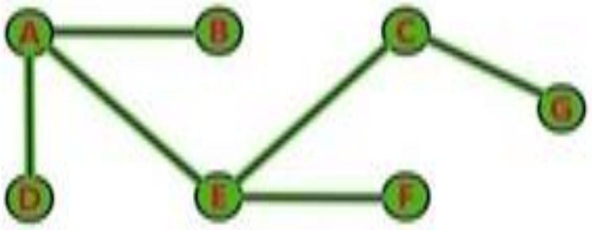


Step 8:

- Visit all adjacent vertices of **G** which are not visited (**there is no vertex**).
- Delete **G** from the Queue.



- Queue became Empty. So, stop the BFS process.
- Final result of BFS is a Spanning Tree as shown below...



9B.Writea C Program to implement Depth FirstSearch (DFS)

Algorithm:

Step1: DefineaStackofsizetotalnumberofvertices Intergraph.

Step2: Selectanyvertexas**startingpoint**fortraversal.Visitthatvertexand push it on to the Stack.

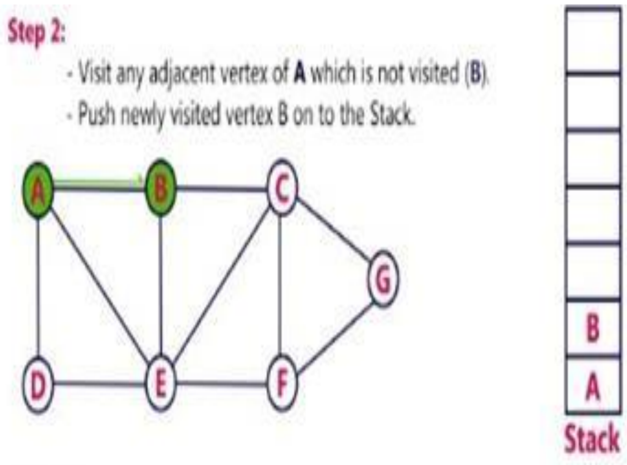
Step3: Visitanyoneofthe**adjacent**vertexofthevertexwhichisattopof the stack which is not visited and pushes it onto the stack.

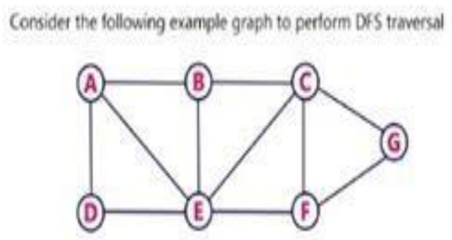
Step4: Repeat step3 until there is no new vertex to be visit from the vertex on top of the stack.

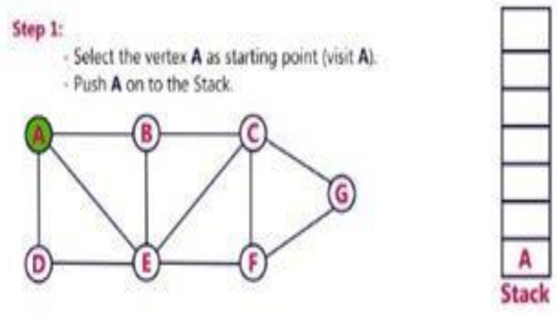
Step5: When there is no new vertex to be visit then use **backtracking** and pop one vertex from the stack.

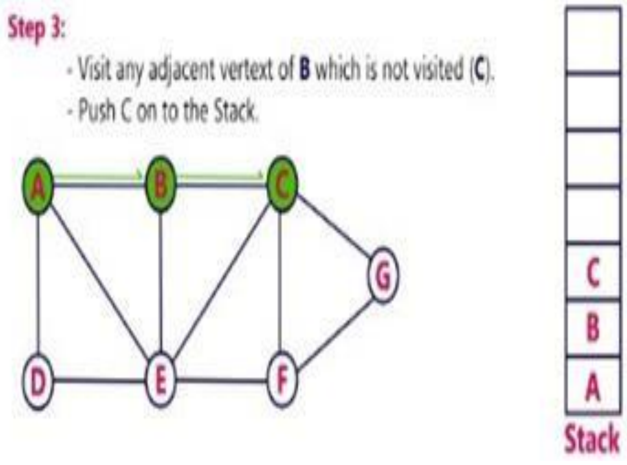
Step6: Repeats teps3,4and 5 until stack becomes Empty.

Step7: When stack becomes Empty, then produce finals panning tree by removing unused edges from the graph


Step 2:
- Visit any adjacent vertex of **A** which is not visited (**B**).
- Push newly visited vertex B on to the Stack.

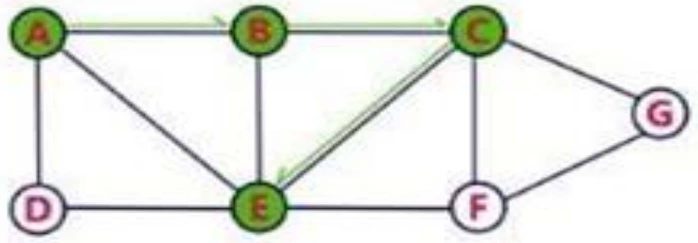

Consider the following example graph to perform DFS traversal

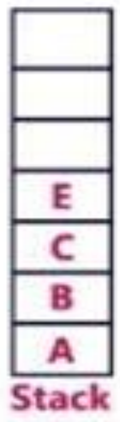

Step 1:
- Select the vertex **A** as starting point (visit **A**).
- Push **A** on to the Stack.


Step 3:
- Visit any adjacent vertext of **B** which is not visited (**C**).
- Push C on to the Stack.

Step 4:

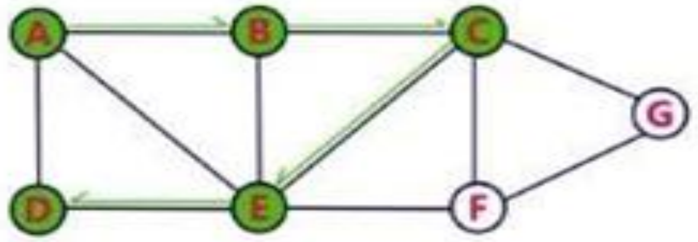
- Visit any adjacent vertext of **C** which is not visited (**E**).
- Push E on to the Stack

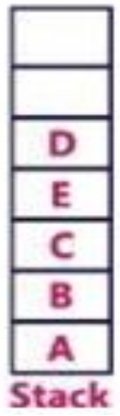




Step 5:

- Visit any adjacent vertext of **E** which is not visited (**D**).
- Push D on to the Stack

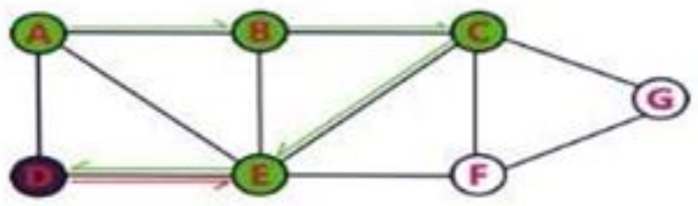


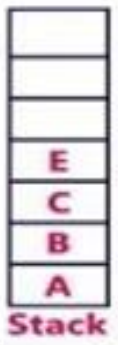


Stack

Step 6:

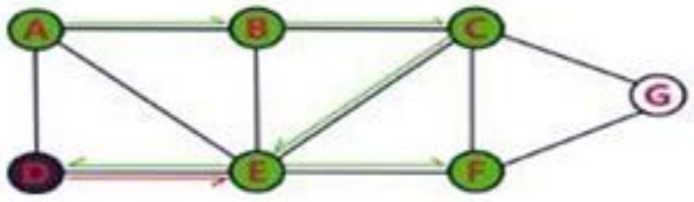
- There is no new vertiex to be visited from D. So use back track.
- Pop D from the Stack.

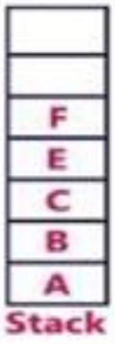




Step 7:

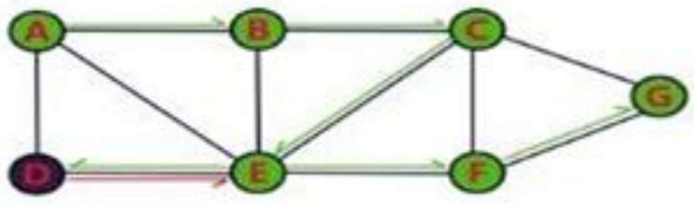
- Visit any adjacent vertex of **E** which is not visited (**F**).
- Push **F** on to the Stack.

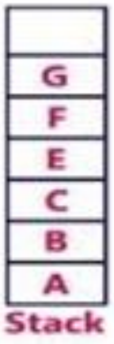




Step 8:

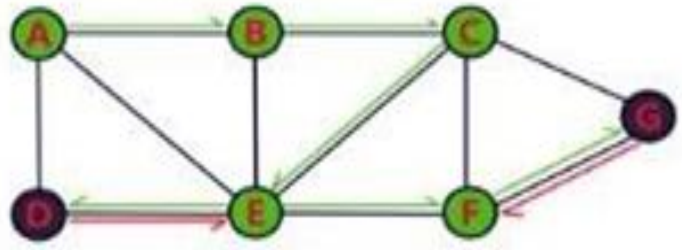
- Visit any adjacent vertex of **F** which is not visited (**G**).
- Push **G** on to the Stack.

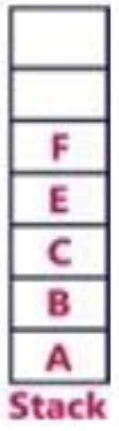




Step 9:

- There is no new vertiex to be visited from G. So use back track.
- Pop G from the Stack.





10. A) Write a c program to implement different hash methods in c

Procedure for to implement different hash methods:

- Hashing is the process of mapping large amount of data item to smaller table with the help of hashing function.
- A fixed process converts a key to a hash key is known a as **Hash Function.**
- This function takes a key and maps it to a value of a certain length which is called a **Hash value** or **Hash.**

What is Hash Table?

- Hash table or hash map is a data structure used to store key-value pairs.
- It is a collection of items stored to make it easy to find them later.
- It uses a hash function to compute an index into an array of buckets or slots from which the desired value can be found.
- It is an array of list where each list is known as bucket.
- It contains value based on the key.
- Hash table is used to implement the map interface and extends Dictionaryc lass.

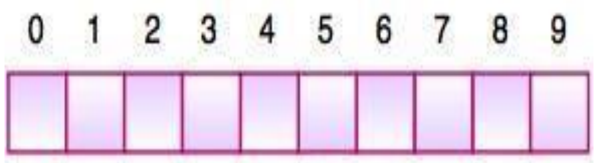


Fig. Hash Table

- Suppose we have integer items {26, 70, 18,31, 54, 93}.One common method of determining a hash key is the division method of hashing and the formulais :
    - **Hash Key =Key Value% Number of Slots in the Table**
- Division method or reminder method takes an item and divides it by the table size and returns the remainder as its hash value.

Data Item	Value% No. of Slots	Hash Value
26	26 %10 =6	6
70	70 %10 =0	0
18	18 %10 =8	8
31	31 %10 =1	1
54	54 %10 =4	4
93	93 %10 =3	3

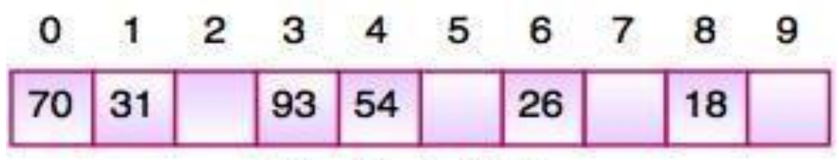


Fig. Hash Table

a. Linear Probing

- Take the above example, if we insert next item40inour collection, it would have a hash value of0 (40 % 10 = 0). But 70 also had a hash value of 0, it becomes a problem. This problem is called as **Collision** or **Clash**. Collision creates a problem for hashing technique.
- **Linear probing is used for resolving the collisions in hash table**, data structures for maintaining a collection of key-value pairs. The simplest method is called Linear Probing.
- Formula to compute linear probing is:**P=(1 +P)% (MOD)Table_size**

  For example,

0
1
2
3
4
5
6
7
8
9
70
31
93
54
26
18

Fig. Hash Table

Linear probing solves this problem:

P = H(40)44%10=**0**

Position0 is occupied by 70. so we look elsewhere for a position to store40.

Using Linear Probing:

P= (P + 1) % table-size0 +1 %10 =**1**

But, position 1is occupied by31, so we look elsewhere for a position to store40.

Using linear probing, we

trynextposition:1+1%10=2Position2 is empty, so 40 is

inserted there

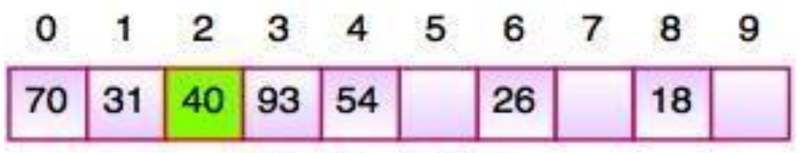


Fig. Hash Table

b. Quadratic Probing:

It is also used for resolving the collisions in hash table. We look for i^2‘thslotini’thiteration.

Let hash(x)be the slot index computed using hash function.

If slot hash(x)%S is full, then we try(hash(x)+1*1)%S

If (hash(x)+1*1)%S is also full, then we try(hash(x)+2*2)%S

If(hash(x)+2*2)%S is also full, then we try (hash(x)+3*3)%S

c. **Doublehashing**isacollisionresolvingtechnique.Doublehashingusestheideaofapplyingasecondhashfunctiontokeywhenacollisionoccurs.

Double hashing can be done using:

(hash1(key)+i*hash2(key))%TABLE_SIZE

Herehash1()andhash2()arehashfunctionsandTABL

E_SIZEissizeofhashtable.

(We repeat by increasing collision occurs)

First hash function is typically hash1(key)=key%TABLE_SIZE

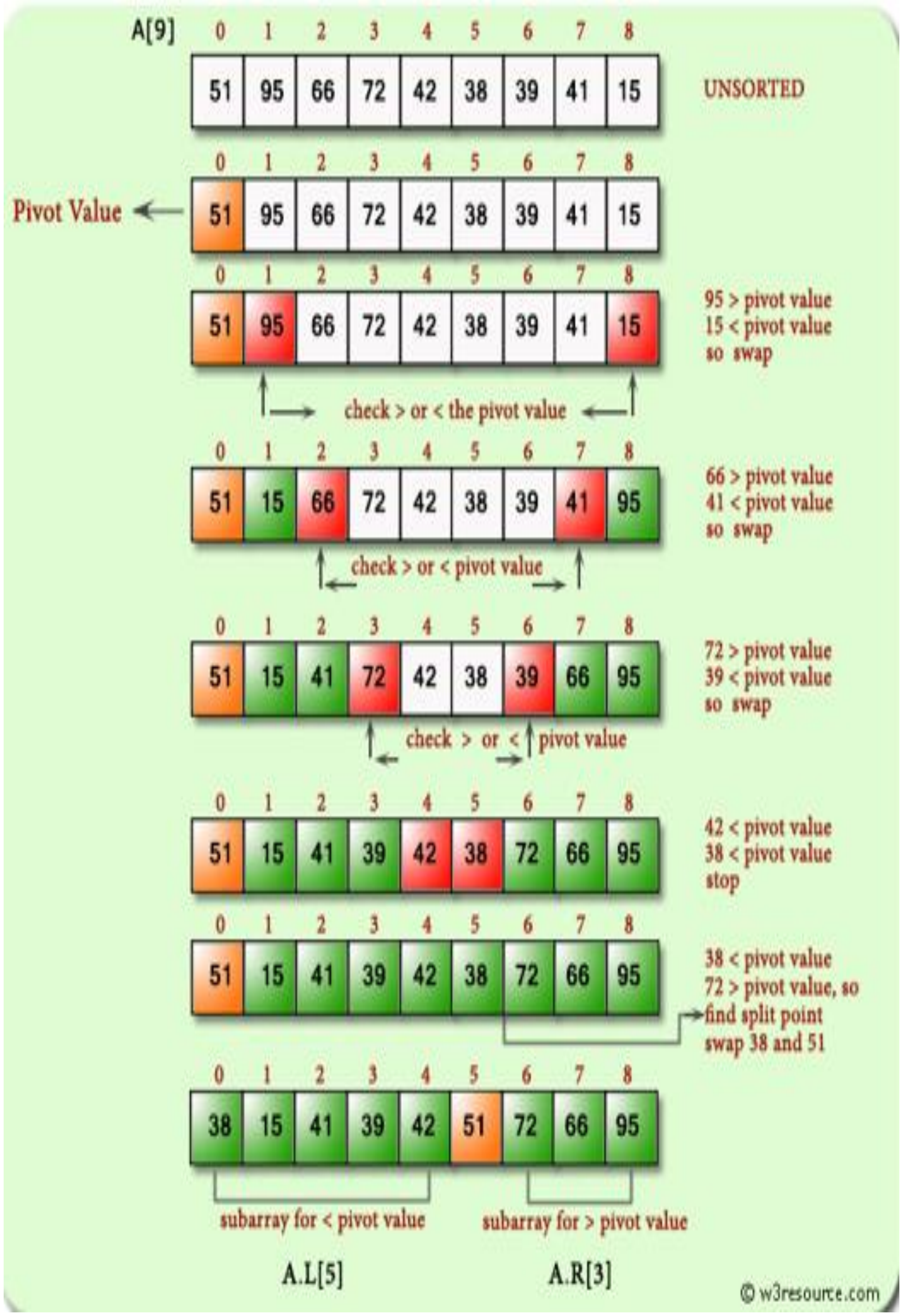
A popular second hash functionis:**hash2(key)=PRIME–(key% PRIME)**

Where PRIME is a prime smaller than the TABLE_SIZE.

11A.Implementation of Quick Sort in C

Quick sort works in the following procedure:

1. Taking the a logical view in perspective, consider situation where one had to sort the papers bearing the names of the students, by name. One might use the approaches follows:
2. Select a splitting value, say L.The splitting value is also known as **Pivot**.
3. Divide the stack of papers into two. A-Land M-Z. It is not necessary that the piles should be equal.
4. Repeat the above two steps with the A-Lpile,splitting it into its significant two halves.And M-Zpile,splitin to its halves. The process is repeated until the piles are smaller nough to be sorted easily.
5. Ultimately, the smaller piles can be placed one onto pof the other to produce a fully sorted and ordered set of papers.
6. The approach used here is **recursion** at each split to get to the single-element array.
7. At every split, the pile was divided and then the same approach was used for the smaller piles.
8. Due to these features, quick sort is also called as ***partition exchange sort.***
    a. An example eight come in handy to understand the concept.


A[9]
0 1 2 3 4 5 6 7 8
51 95 66 72 42 38 39 41 15
UNSORTED
Pivot Value
51 95 66 72 42 38 39 41 15
51 95 66 72 42 38 39 41 15
95 > pivot value
15 < pivot value
so swap
check > or < the pivot value
51 15 66 72 42 38 39 41 95
66 > pivot value
41 < pivot value
so swap
check > or < pivot value
51 15 41 72 42 38 39 66 95
72 > pivot value
39 < pivot value
so swap
check > or < pivot value
51 15 41 39 42 38 72 66 95
42 < pivot value
38 < pivot value
stop
51 15 41 39 42 38 72 66 95
38 < pivot value
72 > pivot value, so
find split point
swap 38 and 51
38 15 41 39 42 51 72 66 95
subarray for < pivot value
subarray for > pivot value
A.L[5]
A.R[3]
© w3resource.com

11B.Implementation Heap Sort in C

The Heap sort algorithm to arrange a list of elements in ascending order Is performed using the following steps...

Step1-Construct a **Binary Tree** with given list of Elements.

Step2-Transform the Binary Tree in to **Min Heap.**

Step3-Delete the root element from Min Heap using **Heap if** method.

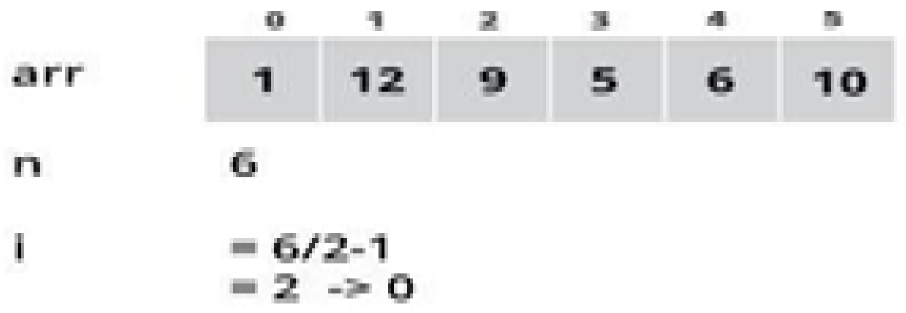
Step4-Put the deleted element into the Sorted list.

Step5-Repeatthesame until Min Heap becomes empty.

Step6 -Display the sorted list.

Note: Heap it is the process of adjusting the tree, to maintain the heap property.

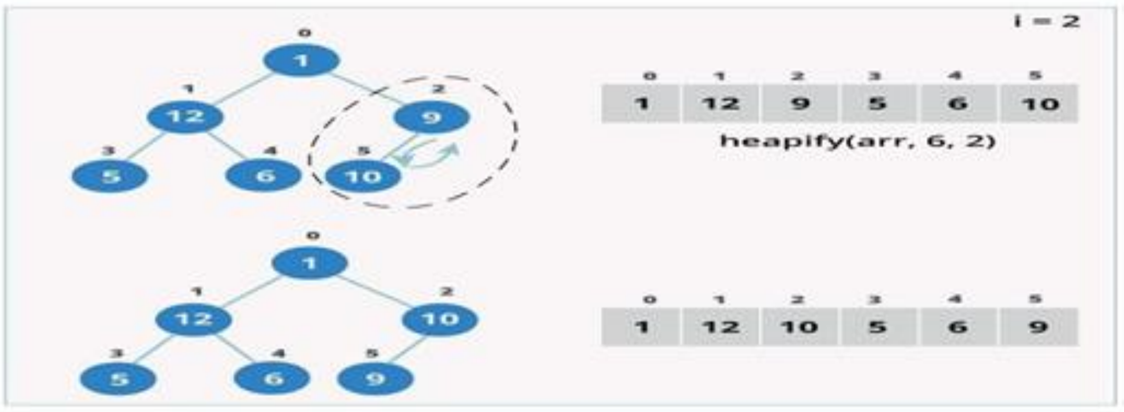
Example:



	0	1	2	3	4	5
arr	1	12	9	5	6	10

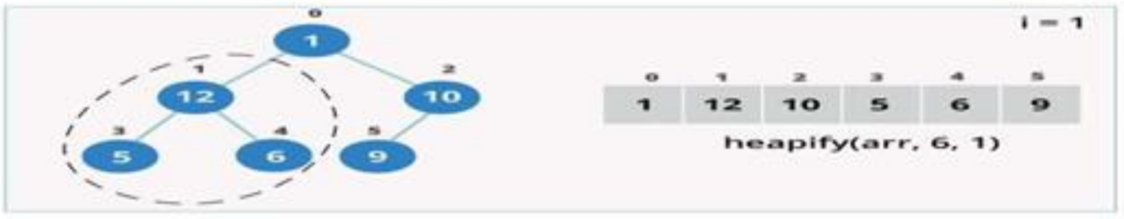
n 6

i = 6/2-1
= 2 -> 0

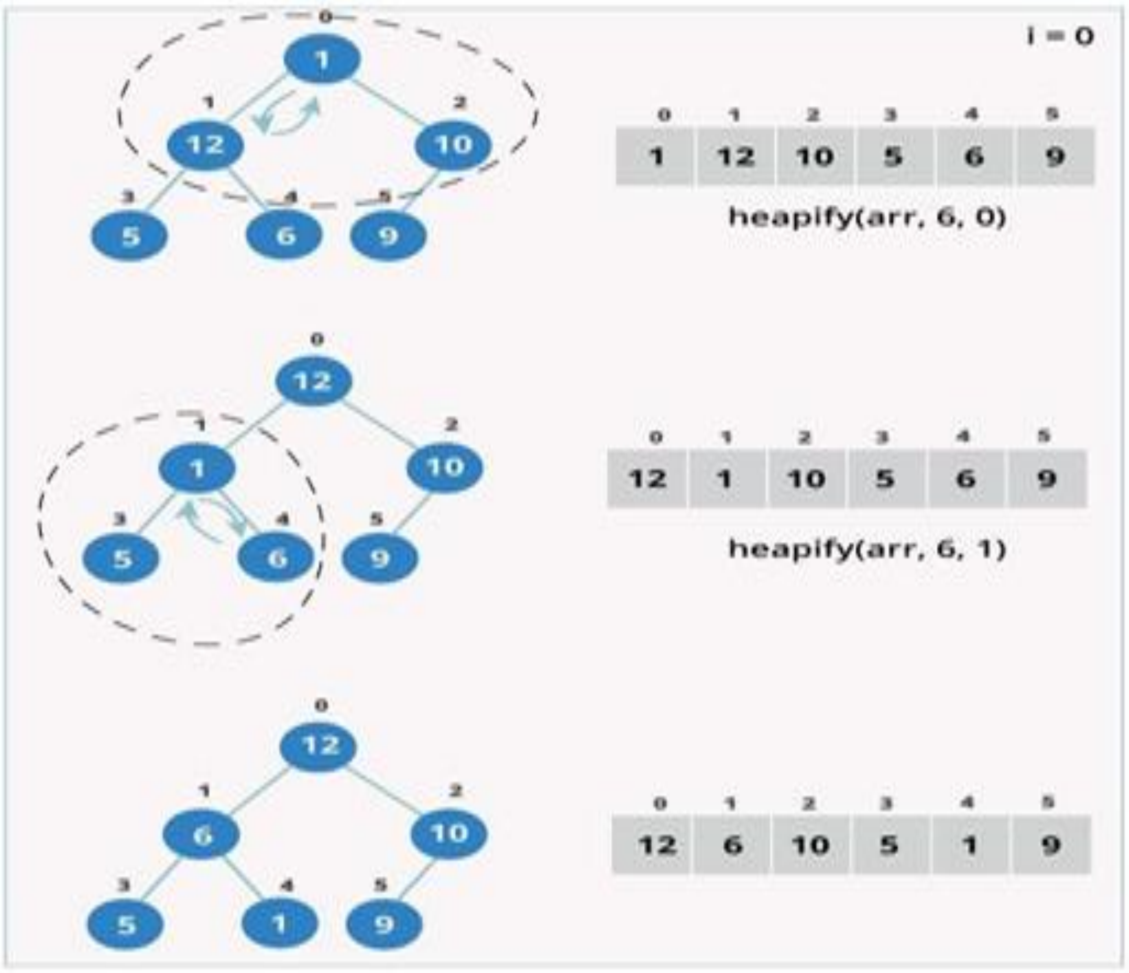


0	1	2	3	4	5
1	12	9	5	6	10

0	1	2	3	4	5
1	12	10	5	6	9



0	1	2	3	4	5
1	12	10	5	6	9



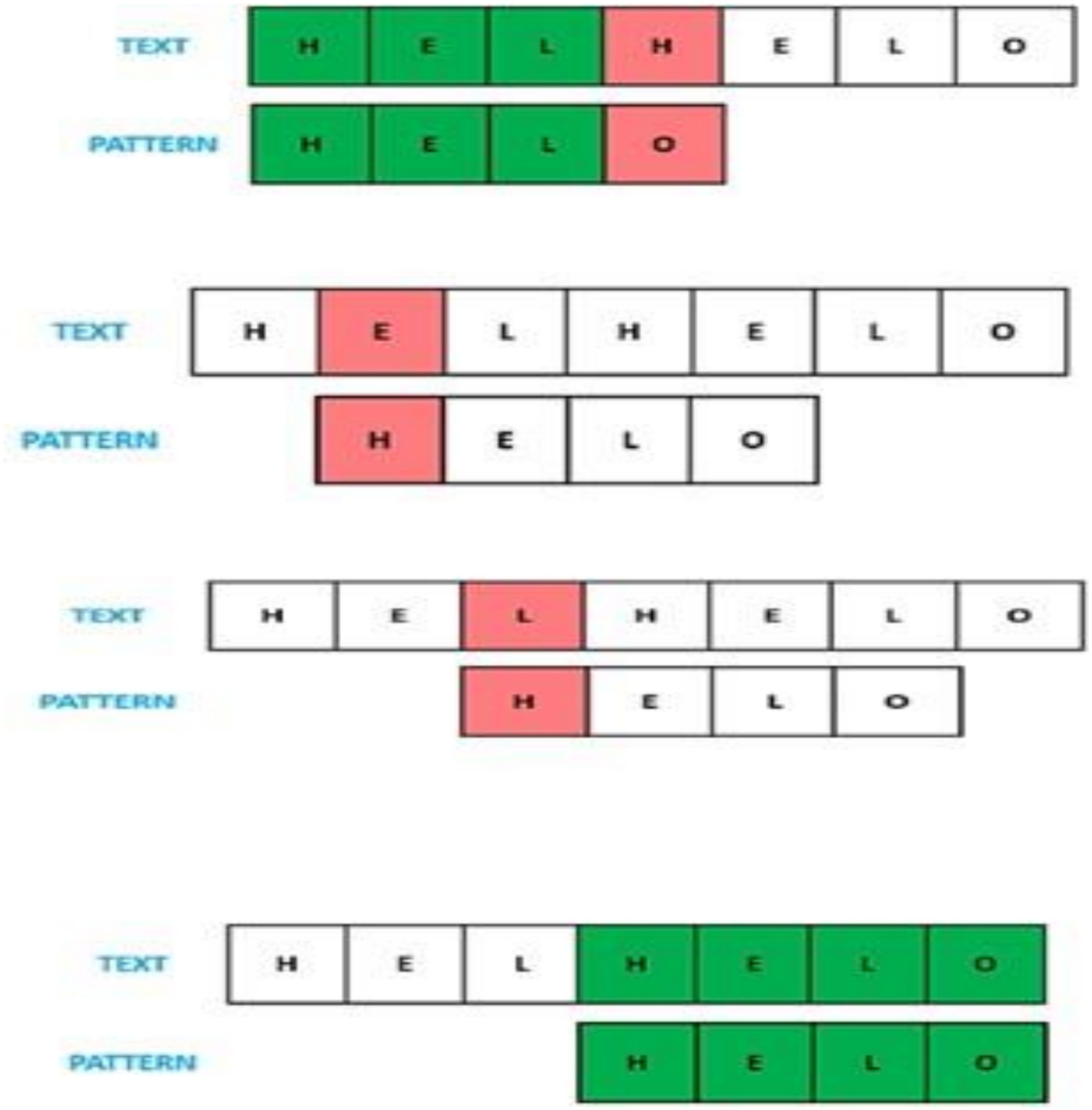
	0	1	2	3	4	5
heapify(arr, 6, 0)	1	12	10	5	6	9
heapify(arr, 6, 1)	12	1	10	5	6	9
	12	6	10	5	1	9

12A.Implement brute-force method of string matching in C

The principle of brute-force search is quite simple .Comparing the characters from left to right continuously (it is crucial because faster approaches work differently). The algorithm checks whether the actual character in the text matches the give character in the pattern.

The Brute Force algorithm compares the pattern to the text, one character at a time, until un-matching characters are found.

Brute-force substrings earch works is as follows.



Brute-force pseudo-code:

```
do
    if (text letter == pattern letter)
            compare next letter of pattern to next letter of text
    else
            move pattern down text by one letter
while (entire pattern found or end of text)
```

12. B)Knuth Morris Pratt Pattern Matching Algorithm in C

The Knuth-Morris-Pratt (KMP) string searching algorithm differs from the brute-force algorithm by keeping track of information gained from previous comparisons.

KMP algorithm is used to find a **"Pattern"** in a **"Text"**. This algorithm compares character by character from left to right.But whenever am match occurs, it uses a preprocessed table called **"Prefix Table"** to skip characters comparison while matching. Sometimes prefix table is also known as **LPS Table**. Here LPS stands for **"Longest proper Prefix which is also Suffix"**.

Steps for Creating LPS Table (Prefix Table)

Step1: Define a one dimensional array with the size equal to the length of the Pattern.(LPS[size])

Step2: Define variables **i &j**. Set i =0, j= 1andLPS[0]=0.

Step3: Comparethecharactersat **Pattern[i]**and **Pattern[j].**

Step 4: If both are matched then set **LPS[j] = i+1** and increment both i & j values by one.Goto to Step 3.

Step5: If both are not matched then check the value of variable 'i'. If it is '0' then set **LPS[j] = 0** and increment 'j' value by one, if it is not '0' then set **i = LPS[i-1]**. Goto Step3.

Step6: Repeat above steps until all the values of LPS[]arefilled.

Example:

Example for creating KMP Algorithm's LPS Table (Prefix Table)

Consider the following Pattern

	0	1	2	3	4	5	6
Pattern :	A	B	C	D	A	B	D

Let us define LPS[] table with size 7 which is equal to length of the Pattern

	0	1	2	3	4	5	6
LPS							

Step 1 - Define variables i & j. Set i = 0, j = 1 and LPS[0] = 0.

	0	1	2	3	4	5	6
LPS	0						

i = 0 and j = 1

Step 2 - Campare Pattern[i] with Pattern[j] ===> A with B.
Since both are not matching and also "i = 0", we need to set LPS[j] = 0 and increment 'j' value by one.

	0	1	2	3	4	5	6
LPS	0	0					

i = 0 and j = 2

Step 3 - Campare Pattern[i] with Pattern[j] ===> A with C.
Since both are not matching and also "i = 0", we need to set LPS[j] = 0 and increment 'j' value by one.

	0	1	2	3	4	5	6
LPS	0	0	0				

i = 0 and j = 3

Step 4 - Campare Pattern[i] with Pattern[j] ===> A with D.
Since both are not matching and also "i = 0", we need to set LPS[j] = 0 and increment 'j' value by one.

	0	1	2	3	4	5	6
LPS	0	0	0	0			

i = 0 and j = 4

Step 5 - Campare Pattern[i] with Pattern[j] ===> A with A.
Since both are matching set LPS[j] = i+1 and increment both i & j value by one.

	0	1	2	3	4	5	6
LPS	0	0	0	0	1		

i = 1 and j = 5

Step 6 - Campare Pattern[i] with Pattern[j] ===> B with B.
Since both are matching set LPS[j] = i+1 and increment both i & j value by one.

	0	1	2	3	4	5	6
LPS	0	0	0	0	1	2	

i = 2 and j = 6

Step 7 - Campare Pattern[i] with Pattern[j] ===> C with D.
Since both are not matching and i !=0, we need to set i = LPS[i-1]
===> i = LPS[2-1] = LPS[1] = 0.

	0	1	2	3	4	5	6
LPS	0	0	0	0	1	2	

i = 0 and j = 6

Step 8 - Campare Pattern[i] with Pattern[j] ===> A with D.
Since both are not matching and also "i = 0", we need to set LPS[j] = 0 and increment 'j' value by one.

	0	1	2	3	4	5	6
LPS	0	0	0	0	1	2	0

Here LPS[] is filled with all values so we stop the process. The final LPS[] table is as follows...

	0	1	2	3	4	5	6
LPS	**0**	**0**	**0**	**0**	**1**	**2**	**0**

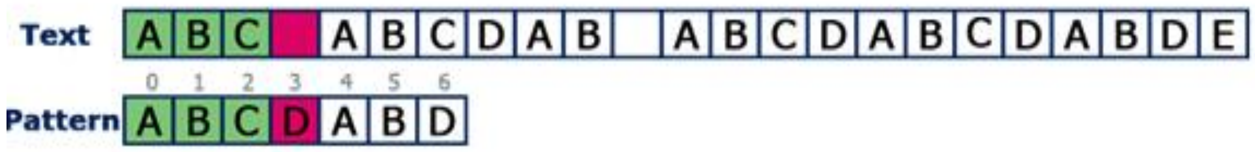
Consider the following Text and Pattern

Text : ABC ABCDAB ABCDABCDABDE
Pattern : ABCDABD

LPS[] table for the above pattern is as follows...

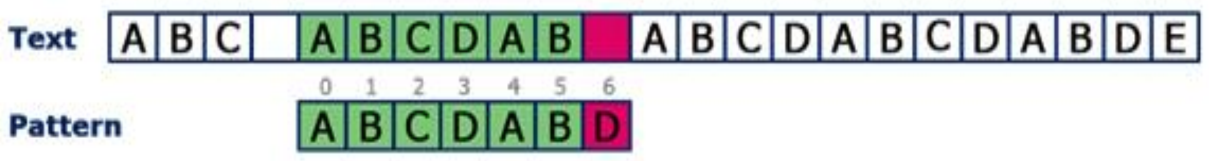
	0	1	2	3	4	5	6
LPS	0	0	0	0	1	2	0

Step 1 - Start comparing first character of Pattern with first character of Text from left to right



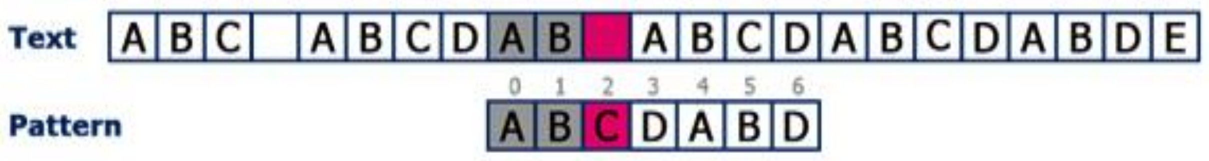
Here mismatch occured at Pattern[3], so we need to consider LPS[2] value. Since LPS[2] value is '0' we must compare first character in Pattern with next character in Text.

Step 2 - Start comparing first character of Pattern with next character of Text.



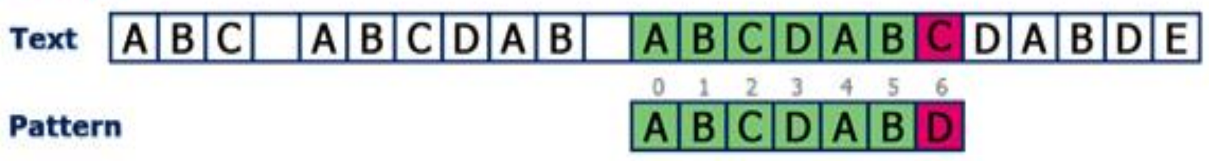
Here mismatch occured at Pattern[6], so we need to consider LPS[5] value. Since LPS[5] value is '2' we compare Pattern[2] character with mismatched character in Text.

Step 3 - Since LPS value is '2' no need to compare Pattern[0] & Pattern[1] values



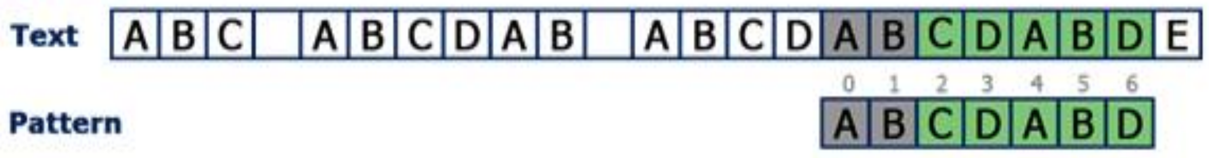
Here mismatch occured at Pattern[2], so we need to consider LPS[1] value. Since LPS[1] value is '0' we must compare first character in Pattern with next character in Text.

Step 4 - Compare Pattern[0] with next character in Text.



Here mismatch occured at Pattern[6], so we need to consider LPS[5] value. Since LPS[5] value is '2' we compare Pattern[2] character with mismatched character in Text.

Step 5 - Compare Pattern[2] with mismatched character in Text.



Here all the characters of Pattern matched with a substring in Text which is starting from index value 15. So we conclude that given Pattern found at index 15 in Text.