Module 3- Linked List and Trees	
Module 3 Syllabus	**LINKED LISTS**: Additional List Operations, Sparse Matrices, Doubly Linked List. **TREES:** Introduction, Binary Trees, Binary Tree Traversals, Threaded Binary Trees.

LINKED LISTS

3.1. Additional List Operations

3.1.1.ReverseOperations For chains: It is often necessary to build a variety of functions for manipulating singly linked list. **Inverting (or reversing) a chain** is one of the useful operation.
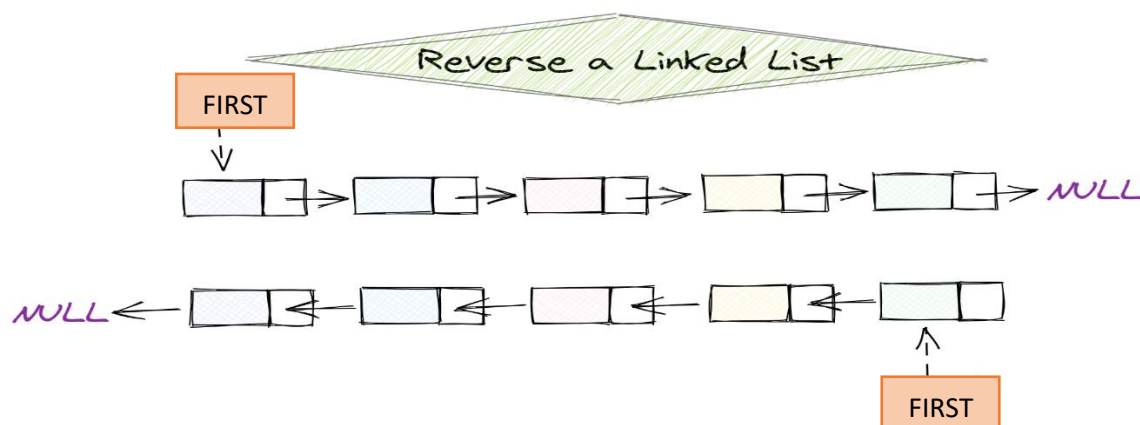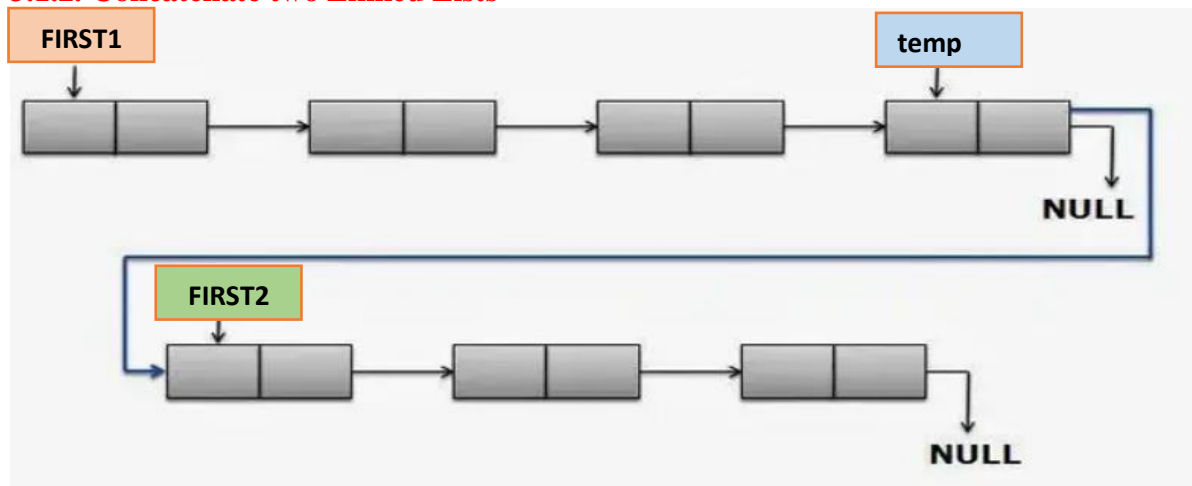

Program code for reverse operation

```
struct node
{
        int data;
        struct node *link;
};
void reverse()
{
struct node*trail, mid=NULL,*lead;
lead= first;
while(lead)
{
        trail=mid;
mid=lead;
lead=lead->link;
mid->link=trail;
 }
first=mid;

}
```

3.1.2. Concatenate two Linked Lists

Algorithm for concatenation

Let us assume that the two linked lists are referenced by **first1** and **first2** respectively.

1. If the first linked list is empty then return **first2**.

2. If the second linked list is empty then return **first1**.

3. Store the address of the starting node of the first linked list in a pointer variable, say **temp**.

4. Move the **temp** tothe last node of the linked list through simple linked list traversal technique.

5. Store the address of the first node of the second linkedlist in the link field of the node pointed by **temp**. Return **first1.**

Program code to concatenate two Linked Lists

```
struct node
{
        int data;
        struct node *link;
};
typedefstruct node * NODE

NODE concatenate()
{
    NODE cur=first1; //place temp on the first node of the first linked list
 if (first1==NULL)  //if the first linked list is empty
      return(first2);
if(first2==NULL)  //if second linked list is empty
      return(first1)
```

```
        while (cur->link!=NULL)              //move p to the last node
            {
             cur=cur->link;
            }
        cur->link=first2;                    //address of the first node of the second linked
list stored in the last node of the first linked list
            first2=NULL;
        return(first1);
}
```

3.1.3.Search the linked list

Searching can be done for two cases:

1. List is unsorted
2. List is sorted

List is unsorted

To do the search operation in the unsorted list, we need to follow the following steps:

1. Set first as cur
2. Repeat step 3 while cur≠NULL
3. If key = cur→data; search is successful and return

   Else set cur=cur→link
4. Search failed
5. Exit

Program code for Searching a key in an Unsorted Linked List

```
void search()
{   struct node *temp;
    int key;
    cur=first;
    printfcv("Enter key");
    scanf("%d", &key);
    while (cur!=NULL)
    {       if(key==cur→data)
            {       printf("Search successful");
                    return;
            }
            else    cur=cur→link;
```

```
        }
        printf("Search failed");
    }
```

List is sorted

If we want to search any key element then we need to follow the step to do the search operation for the sorted list, we need to follow the following steps:

1. Set cur=first
2. Repeat step 3 while cur≠NULL
3. If key>cur→data then set cur=cur→link

   Else if key= cur→data then display search is success

   Else display search is failed
4. Exit

Program code for Searching a key in aSorted Linked List

```
void search()
{       struct node *cur;
        int key;
        cur=first;
printf("Enter key");
        scanf("%d", &key);
        while (cur!=NULL)
        {       if(key>cur→data)
                        cur=cur→link;
                else if(key==cur→data)
                {       printf("Search successful");
                        return;
                }
                else
                        printf("Search unsuccessful");
        }
}
```

3.1.4.Inserting a Node After a specific Node(key) value in a Linked List

```c
void insertafterspecific()
{
int key;
struct node *cur;
temp= (struct node*)malloc(sizeof(struct node));
printf("\n enter the element");
scanf("%d",&temp->data);
temp->link=NULL;


printf("enter the key value \n");
scanf("%d",&key);
if(first == NULL)
  {
      Printf("list is empty");
  }
else
  {
  cur=first;
  while(cur->data != key)
        {
        cur = cur->link;
        }
  temp->link = cur->link;
  cur->link = temp;
  }
printf("\nOne node inserted!!!\n");
}
```

3.1.5.Removing a specific in a Linked List

```c
void removeSpecific()
{
struct Node *cur = head, *prev;
int key;
printf("\n enter the node to be deleted");
```

```
        scanf("%d",&key);


while(cur->data != key)
  {
      if(cur -> link == NULL){
       printf("\nGiven node not found in the list!!!");
       }
      prev = cur;
      cur = cur ->link;
  }
  prev-> link= cur -> link;
  free(cur);
   printf("\nOne node deleted!!!\n\n");
}
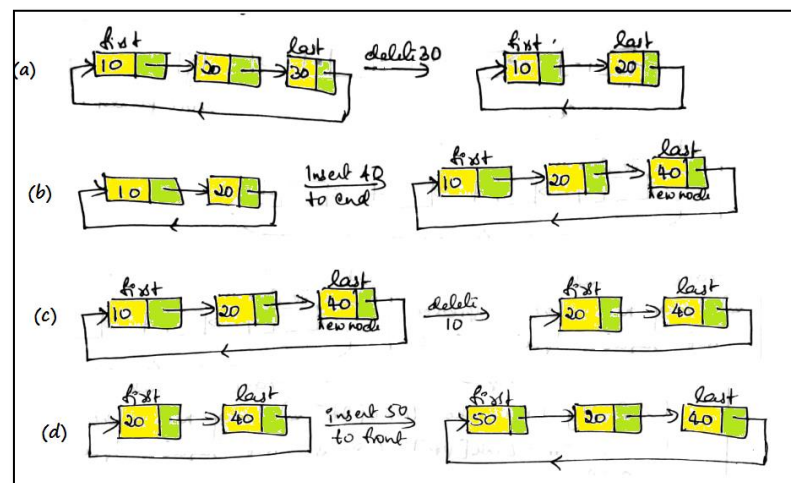```

### 3.1.6. Operations for Circular Linked List



Fig.3.1.6: Operations of Circular Linked Lists.

1. When we delete from end, delete the node which lies in the last position and make its previous node link points to first node as in Fig. 3.1.6(a).

2. When we insert to end, make last node link connect to new node and new node link connect to first node as in Fig. 3. 1.6(b).

3. When we delete from front, delete the first node and make the last node link to point to next node of first node as in Fig. 3.1.6 (c).

4. When we insert to front, make last node link connect to new node and new node link connect to first node and then make new node itself as first as in Fig. 3.1.6(d).

### 3.1.6.1 Create a Node

```
struct node
{
int data;
struct node *next;
};
struct node *head = NULL,*temp;
```

3.1.6.2 Insert into the front of the list

```
void insertAtBeginning()
{
NODE cur, temp;
newNode = (struct node*)malloc(sizeof(struct node));
printf("\n enter the data:");
scanf("%d", &temp -> data);
if(first == NULL)
    {
        first = temp;
        temp-> link= first;
    }
 else
    {
        cur = first;
        while(cur -> link != first)
        {
                cur = cur -> link;
         }
        cur -> link = temp;
        temp -> link = first;
        first = temp;
    }
printf("\nInsertion success!!!");
}
```

3.1.6.3 Insert into the end

```
void insertAtEnd()
{
NODE cur, temp;
newNode = (struct node*)malloc(sizeof(struct node));
printf("\n enter the data:");
scanf("%d", &temp -> data);
if(first == NULL)
    {
        first = temp;
        temp-> link= first;
    }
 else
    {
        cur = first;
        while(cur -> link != first)
        {
                cur = cur -> link;
         }
        cur -> link = temp;
        temp -> link = first;
    }
printf("\nInsertion success!!!");
}
```

3.1.6.4. Delete from front

```
void deletefront()
        {
            if(first== NULL)
                {
                printf("List is Empty!!! Deletion not possible!!!");
                }
```

```
else
{
NODE cur=first, temp=first;
while(cur->link!=first)
        {
            cur=cur->link;
        }
Cur->link = first -> link;
First=first->link;
temp->link=NULL;
free(temp);
}
printf("\nDeletion success!!!");
}
```

3.1.6.5. Delete from end

```
Void deleteend()
{
NODE cur=first;
    if(first == NULL)
    {
            printf("List is Empty!!! Deletion not possible!!!");
    }
    else
    {
        NODE prev=NULL;
        while(cur ->link!=first)
            {
                prev = cur;
                cur = cur -> link;
            }
         prev -> link = first;
        free(cur);
        cur=NULL;
```

```
        }
    printf("\nDeletion success!!!");
        }
```

3.1.7: Finding length of a circular linked list

```
    int count(struct Node *first)//function to count number of nodes
    {
    int cnt = 0;
    struct Node *cur = first;
        //Iterating till end of list
    do
        {
                cur = cur->link;
                cnt++;
        } while (cur != first);
        return cnt;
    }
```

3.2. Sparse matrices

- In linked representation, we use linked list data structure to represent a sparse matrix. In this linked list, we use two different nodes namely header node and element node.

- Header node consists of three fields and element node consists of five fields as shown in the Fig. 3.2.

- Consider the sparse matrix used in the Triplet representation of Fig. 3.2(a). This sparse matrix can be represented using linked representation as shown in the below image.

- In this representation, H0, H1..., H5 indicates the header nodes which are used to represent indexes.

- Remaining nodes are used to represent non-zero elements in the matrix, except the very first node which is used to represent abstract information of the sparse matrix (i.e., It is a matrix of 5 X 6 with 6 non-zero elements).

- In this representation, in each row and column, the last node right field points to its respective header node (Fig 3.2 (b)).
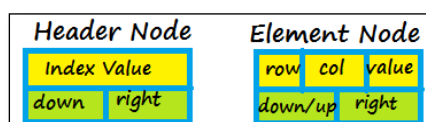
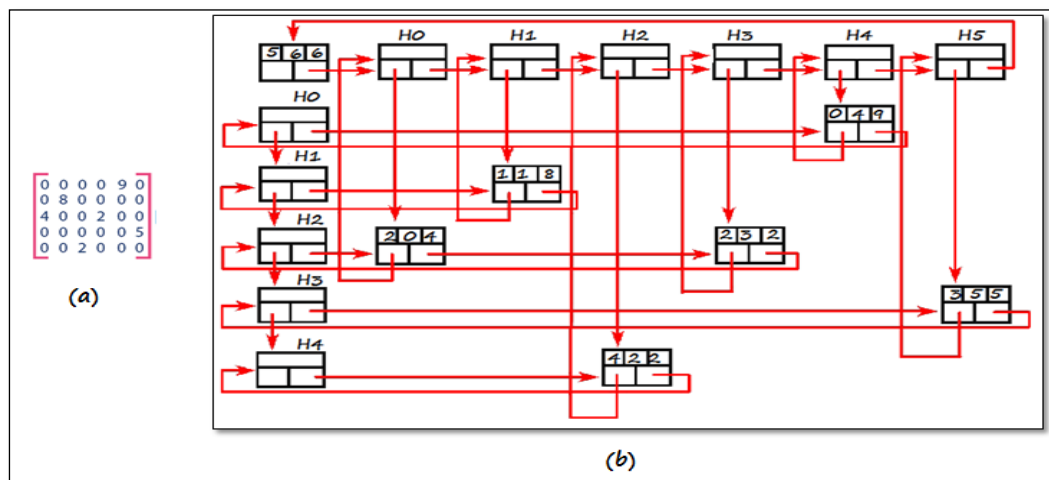Fig 3.2: Header and element node representation.

Fig 3.2.1. Linked list representation of sparse matrix

Sparse matrix using Linked List C program

```
#include<stdio.h>

#include<stdlib.h>

struct list

{

int row, column, value;

struct list *next;

};

struct list *HEAD=NULL;

void insert(int, int , int );

void print ();

int main()

{

intSparse_Matrix[4][4] ={ {9 , 0 , 0 , 0 },{0 , 0 , 0 , 0 },{0 , 5 , 0 , 8 },{3 , 0 , 0 , 0 } };

for(inti=0;i<4;i++)

   {

for(int j=0;j<4;j++)

     {

if(Sparse_Matrix[i][j] != 0)

        {

insert(i, j, Sparse_Matrix[i][j]);
```
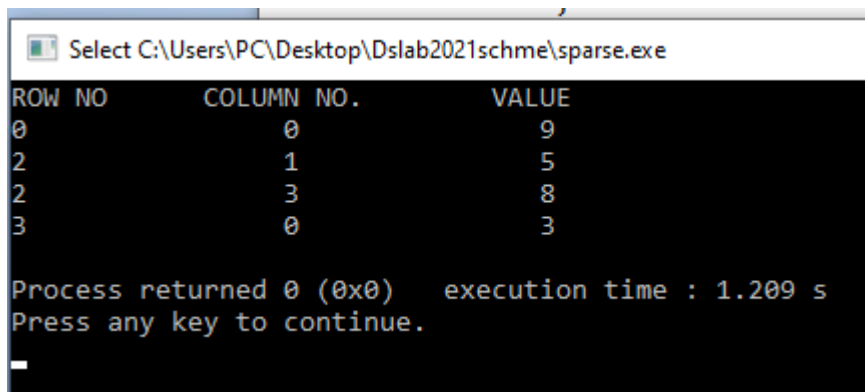
```
            }
         }
      }
   // print the linked list.
print();
}
void insert( int r, int c, int v)
{
struct list *ptr,*temp;
int item;
ptr = (struct list *)malloc(sizeof(struct list));
if(ptr == NULL)
   {
printf("\n OVERFLOW");
   }
else
   {
ptr->row = r;
ptr->column = c;
ptr->value = v;
if(HEAD == NULL)
      {
ptr->next = NULL;
      HEAD = ptr;
      }
else
      {
temp = HEAD;
while (temp -> next != NULL)
         {
temp = temp -> next;
         }
temp->next = ptr;
ptr->next = NULL;
      }
```

```
    }
}
void print()
{
struct list *tmp = HEAD;
printf("ROW NO     COLUMN NO.      VALUE \n");
while (tmp != NULL)
    {
printf("%d \t\t %d \t\t %d \n", tmp->row, tmp->column, tmp->value);
tmp = tmp->next;
    }
}
```

3.3.Doubly Linked List

Doubly linked list is a linear collection of data elements, called nodes, where each node N is divided into three parts:

1. An information field INFO which contains the data of N.
2. A pointer field LLINK which contains the pointer to previous node.
3. A pointer field RLINK which contains the pointer to next node.

This list also contains pointer field 'first' which points to first node and 'last' which points to last node.

3.3.1 Representation

Fig. 3.3.1(a) shows the representation of doubly linked list. Using first and RLINK we can traverse in forward direction. Using last and LLINK we can traverse in backward direction. Structure declaration of doubly linked list is shown in Fig.3.3.1(b)
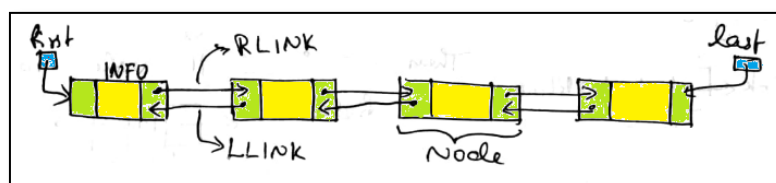

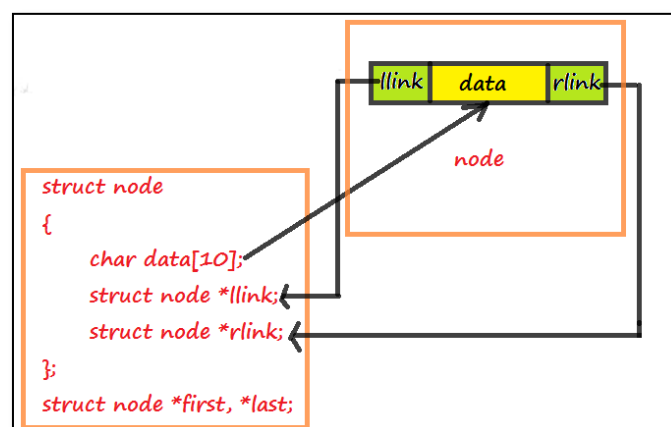Fig.3.3.1(a): Representation of Doubly Linked List.

Fig.3.3.1(b): Structure Declaration of Doubly Linked List.

3.3.2 Representation of Doubly Linked List in Memory

Let us view how a doubly linked list is maintained in the memory. It can be represented in memory as Fig.3.3.2.
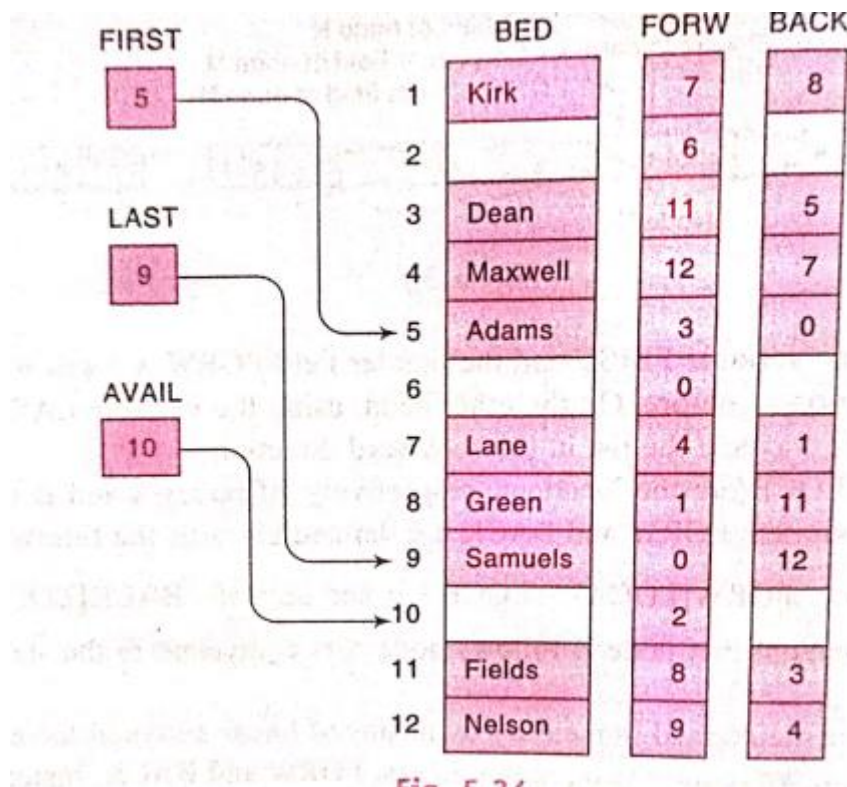
Fig. 3.3.2: Representation of Doubly Linked List in Memory.

3.4.Operations of Doubly Linked List

3.4.1.Create

Initially we make 'first' as 'NULL' as in Fig.1. When we create a new node, name it as 'temp' and store the value in its data field. For example, enter 10 to linked list as in Fig. Then make temp→prev and temp→next as NULL as in Fig.1. If we have only one node which is created just now, then first=NULL; then make new node itself as first and end. We can create 'n' number of nodes together. When you create onenew node, then first is not NULL now. So now we have to connect new nodes right link to old nodes left as in Fig.1and then make first to point to temp as in Fig.1.
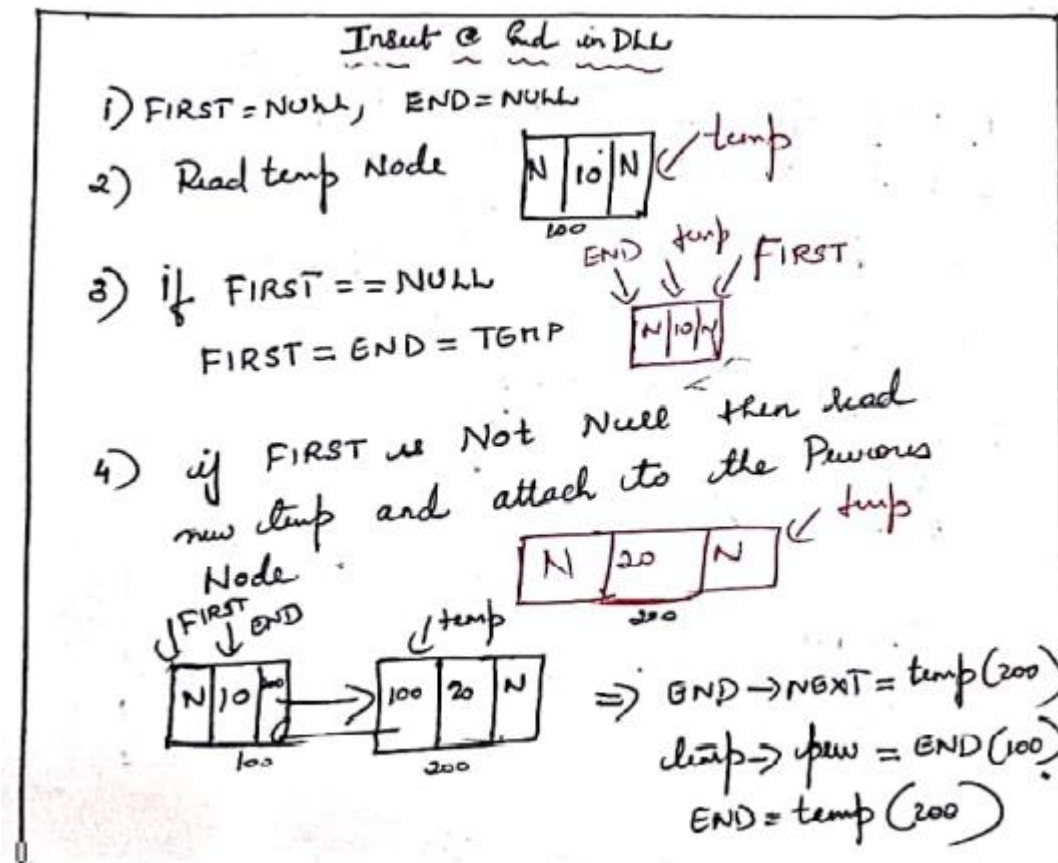
Fig.1: Creation of Doubly Linked List

Program code for create function

struct node

{

      int data;

      struct node *prev;

      struct node *next;

};

typedefstruct node * NODE;

NODE temp,FIRST=NULL,END=NULL;

void create()

{

      intn,i=1;

      printf("\n enter the no of elements to be inserted into the list\n");

      scanf("%d",&n);

      while(i<=n)

```
        {
        printf("enter the details of the node  %d",i++);
        temp = (NODE)malloc(sizeof (struct node));
                printf("Enter the data to be inserted:\n");
                scanf("%d",temp->data);
                temp->prev = temp->next = NULL;
                if (FIRST==NULL)
                        FIRST = END = temp;
                else
                {       END->next=temp;
                        temp->prev=END;
                        END=temp;
                }
        }
}
```

3.4.2.Insert to end

It works same as create function, by inserting new node to front. Here only one node can be inserted at a time.

Program code for insert front operation

```
voidinsertend()
{       struct node * temp;

temp = (NODE)malloc(sizeof (struct node));
printf("Enter the data to be inserted:\n");
scanf("%d",temp->data);
temp->prev = temp->next = NULL;
        if (FIRST==NULL)
                FIRST = LAST = temp;
        else
                {       END->next=temp;
                        temp->prev=END;
                        END=temp;
                }
```

}

3.4.3.Insert to front

Here we start from first position. If nothing is there in the list, then 'first' is pointing to NULL. So if first is NULL then the new node created itself will be pointing to last and first. Then we create a new node temp usingmalloc and insert a new value=30 to it and make its left and right link as NULL as in Fig. 3. If first is not equal to NULL, then connect this temp to the left link of last node as in Fig.3.
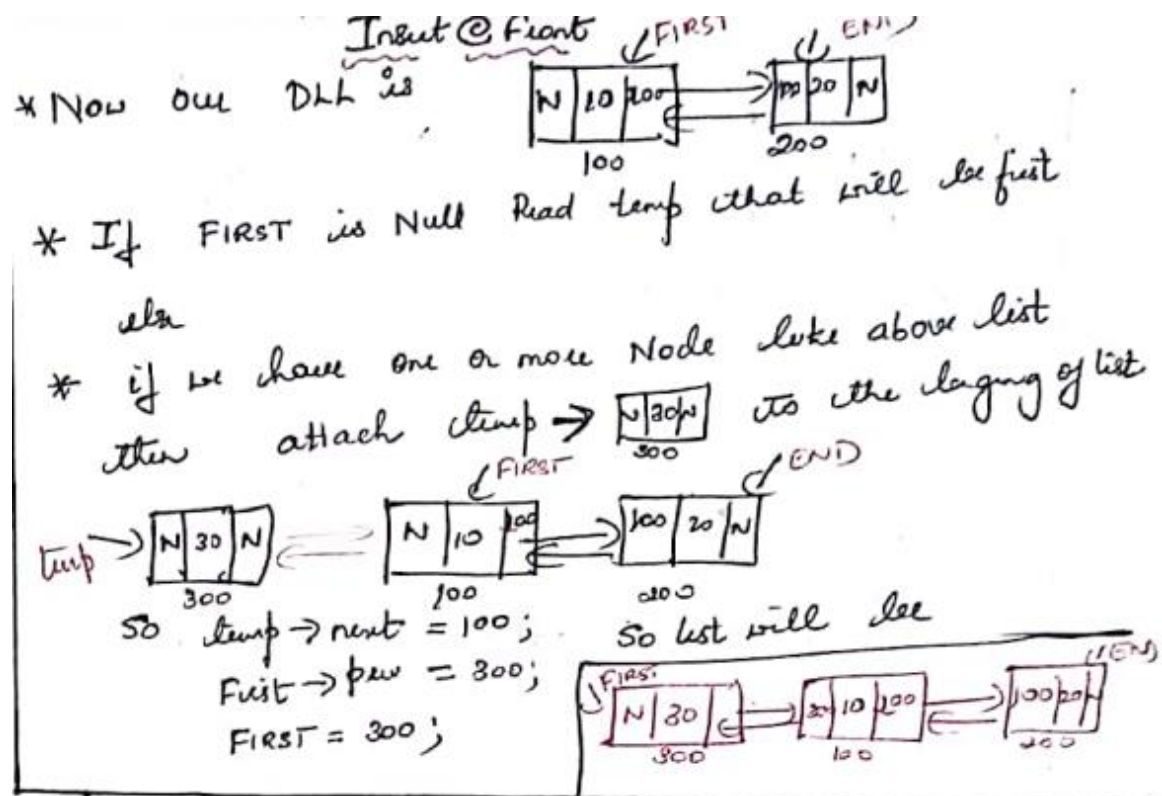

Fig.3: Steps to Inserting a new Node to the front of Doubly Linked List

Program code for insert frontoperation

voidinsert_end()

```
{
printf("enter the details of the node \n");
temp = (NODE)malloc(sizeof (struct node));
        printf("Enter the data to be inserted:\n");
        scanf("%d",temp->data);
        temp->prev = temp->next = NULL;
        if (FIRST==NULL)
```

```
                    FIRST = END = temp;
            else
            {        temp->next=FIRST;
                     FIRST->prev=temp;
                     FIRST=temp;
            }
    }
```

3.4.4.Delete from Front

If first is pointing to NULL, then there is no element in the list. Otherwise we make first as 'temp' and delete the data in 'temp', make its right link node as 'first'. Then make first llink as NULL as in Fig.4. If we have only one node in the list and if we perform delete front, then the same above steps are followed. But now first = NULL. This means list is empty. So make 'last' also as NULL.
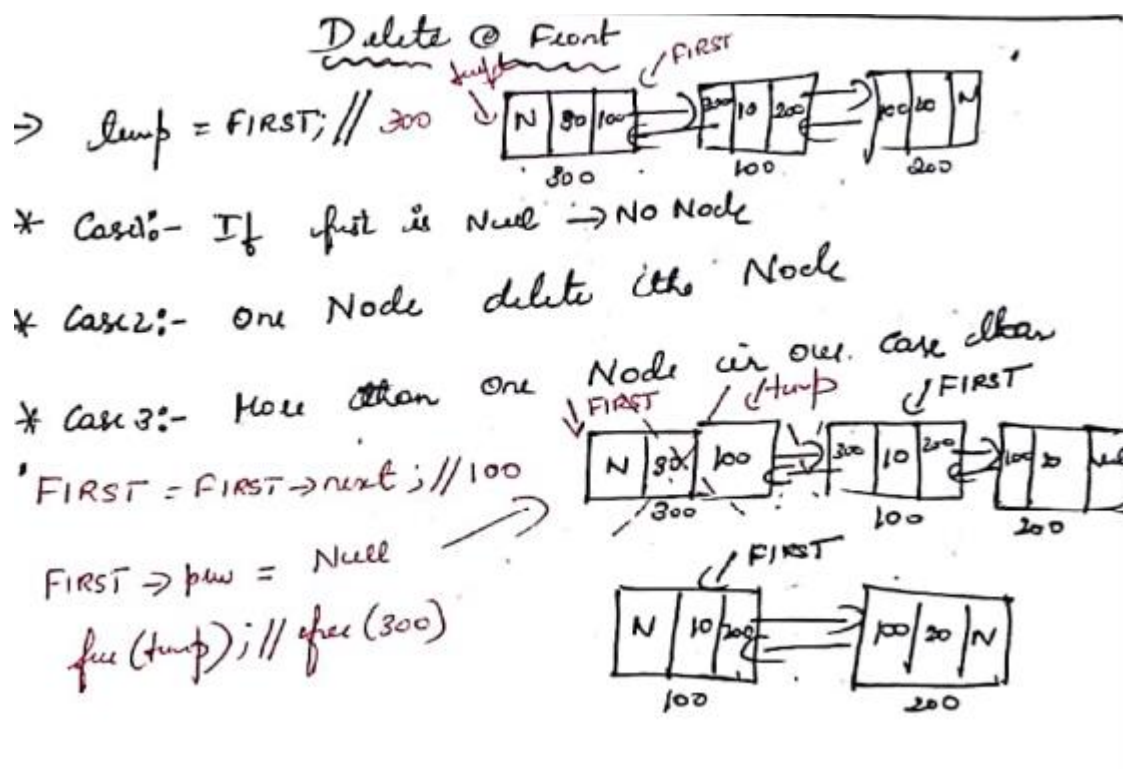
Fig.4: Steps to Deleting a Node fromthe front of a Doubly Linked List

Program code for delete front operation

voidDeletionfront() //Delete node from front of DLL

```
{
        temp=FIRST;
        if(FIRST==NULL) // check for empty list
                        printf("List is empty\n");
        else if(FIRST==END) // otherwise check for single node in list
        {
                printf("deleted element is %d\n", temp->data);
                FIRST=NULL;
                END=NULL;
                free(temp);
        }
else // otherwise delete node from front of DLL
        {
                printf("deleted element is %d\n", temp->data);
                FIRST =FIRST->next;
                FIRST->prev=NULL;
                free(temp);
        }
        return;
```

3.4.5.Delete from End

Here we delete nodes from last pointer. If last is pointing to NULL, then list is empty. Otherwise make last node as'temp' and delete temp→data.Then make previous node as last and its right link to point to NULL as in Fig.5.Now if we have only one node after deletion, and if we perform delete end, last becomes NULL. This means the list is empty. Hence if last is pointing to NULL, and then make first also pointing to NULL.
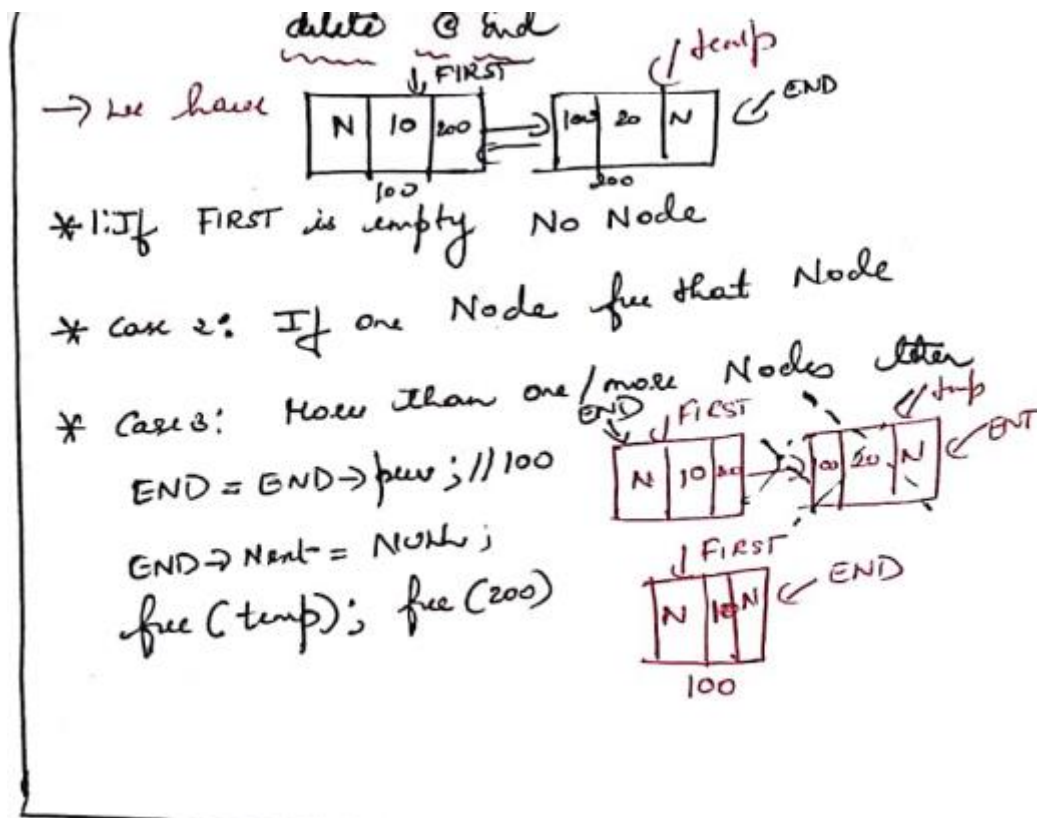
Fig.5: Steps to Deleting a Node from the end of a Doubly Linked List

Program code for delete end operation

voidDeletionend() // delete node at end of DLL

{

       temp = END;

       if(FIRST==NULL) // check for empty list

printf("List is empty\n");

       else if(FIRST==END) // otherwise check for single node in list

       {

              printf("deleted element is %s\n", temp->ssn);

              FIRST=NULL;

              END=NULL;

              free(temp);

       }

       else // otherwise delete end node from DLL

       {

              printf("deleted element is %s\n", temp->ssn);

              END=END->prev;

```
            END->next=NULL;

            free(temp);

      }

      return ;

} // end of deletionend
```

3.4.6.Traverse (Display)

If first is pointing to NULL, then print that the list is empty. Else, make temp to point to first and display temp→data. Then make temp to point to next node and display its data and so on until temp points to NULL. So we traverse from first node to last node.
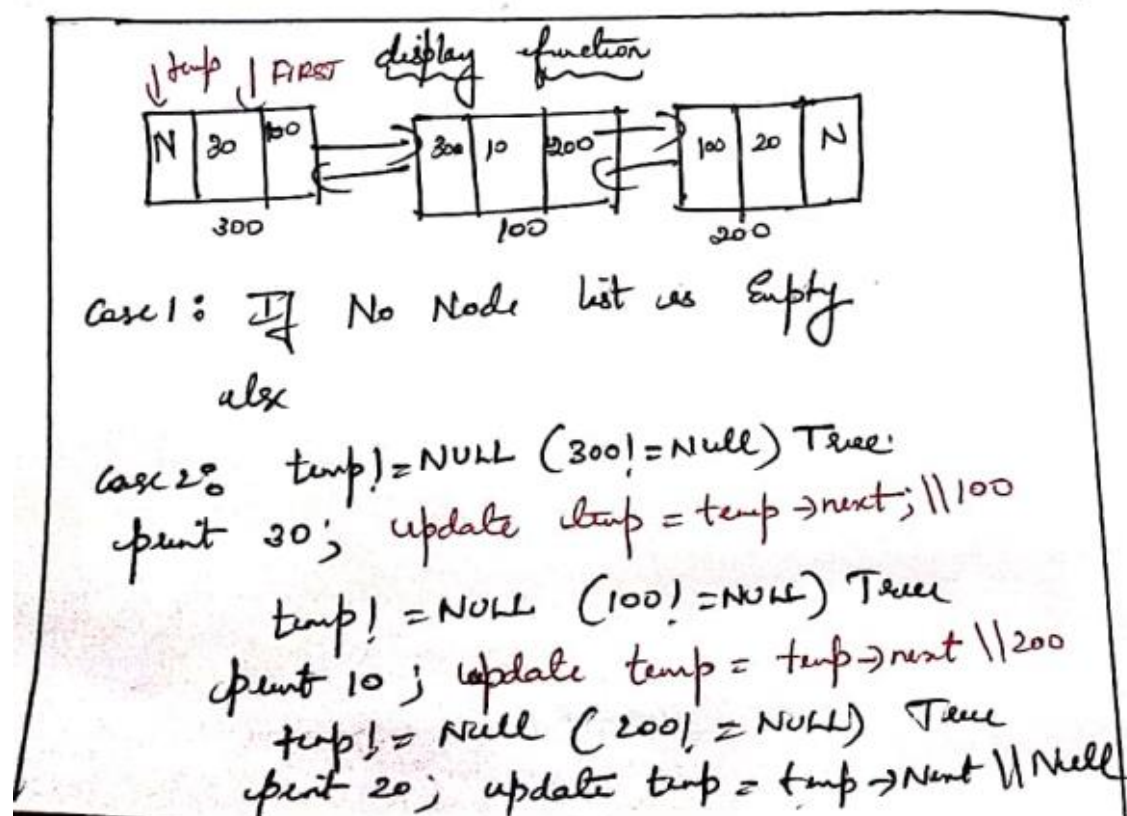
voiddisplay_count() //Display the status of DLL and count the number of nodes in it
{

      temp=FIRST;

      int count=0;

      if(FIRST==NULL) // check for empty list

            printf("the list is NULL and count is %d\n", count);

else

      {

            printf("the list details:\n");

            while(temp!=NULL) // display all nodes in the list

            {

count++;

printf("%d",temp->data);

temp=temp->next;

}

printf("\n node count is %d\n",count);

} // end of else

return;

} // end of display()

3.5 Header Linked List

A header linked list is a special type of linked list which contains a header node at the beginningof the list. The following are the two variants of a header linked list:

- *Grounded header linked list* which storesNULL in the next field of the last node as in Fig. 3.5 (a).
- *Circular header linked list* which stores the address of the header node in the next field of the last node. Here, the header node will denote the end of the list as in Fig. 3.5(b).
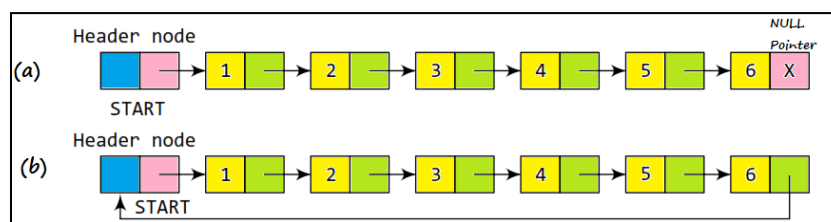
Fig 3.5: Types of Header Linked Lists.

Properties of circular header lists:

- NULL pointer is not used and hence all pointers contains valid address
- Every node has a predecessor. So the 1st node may not require special case.

Algorithm for circular header lists

1. Set ptr = Link [start]

2. Repeat step 3 and 4 while ptr ≠start

3. Apply process to INFO[ptr]

4. Set ptr = Link[ptr] (pointer points to next node)

5. Exit

3.6. Doubly Circular Linked List

- A doubly linked list points to not only the next node but to the previous node as well. A circular doubly linked list contains 2 NULL pointers. The 'Next' of the last node points to the first node in a doubly-linked list. The 'Prev' of the first node points to the last node.

- A doubly circular linked list looks as follows:
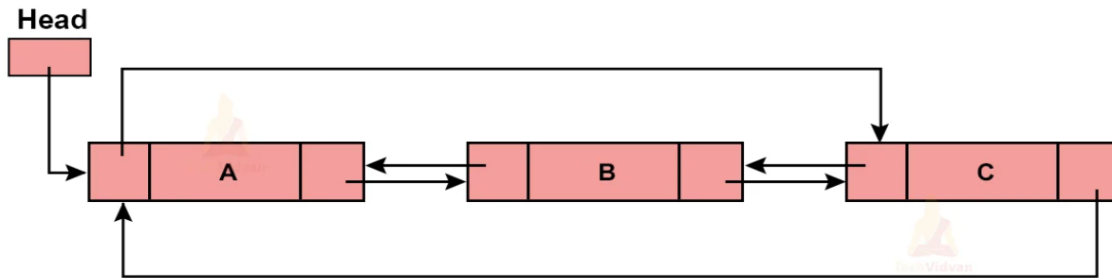
In C, the structure of a Circular doubly linked list can be given as,

```
struct node
  {
        struct node *prev;
        int data;
        struct node *next;
  };
```
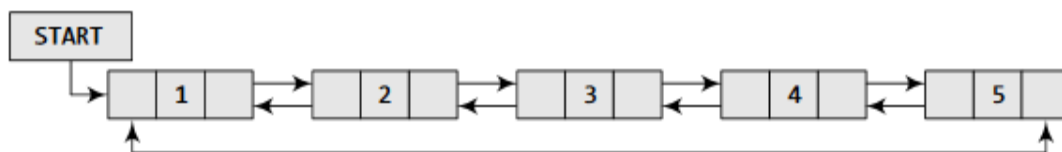

3.6.1.Inserting a New Node in a Circular Doubly Linked List :-

In this section, we will see how a new node is added into an already existing circular doubly linked list. We will take two cases and then see how insertion is done in each case. Rest of the cases are similar to that given for doubly linked lists.
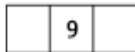
Case 1: The new node is inserted at the beginning.
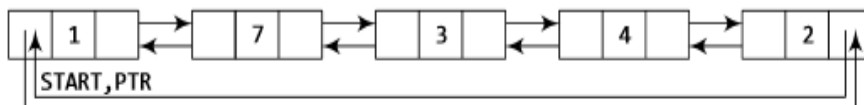**Case 2: The new node is inserted at the end.**

3.6.1.1. Inserting a Node at the Beginning of a Circular Doubly Linked List:-
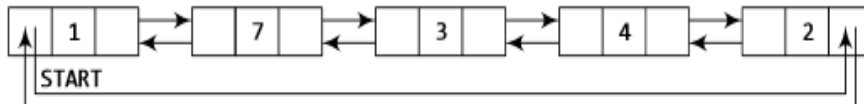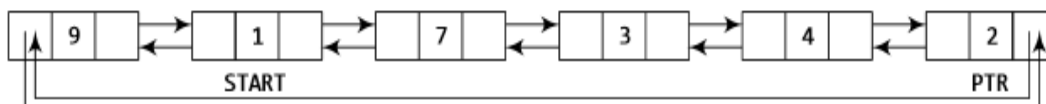

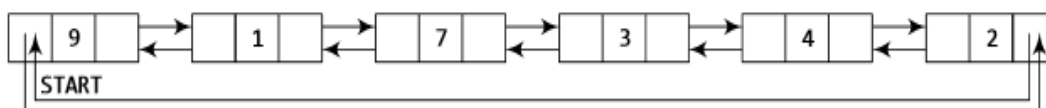Allocate memory for the new node and initialize its DATA part to 9.

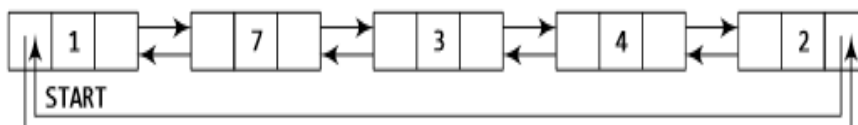Take a pointer variable PTR that points to the first node of the list.

Move PTR so that it now points to the last node of the list. Insert the new node in between PTR and the START node.
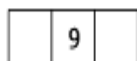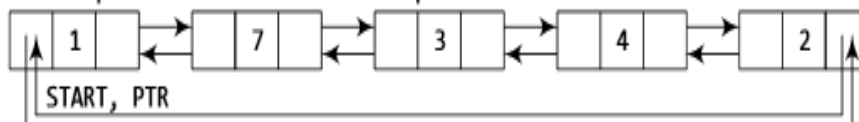
START will now point to the new node.

3.6.1.2. Inserting a Node at the End of a Circular Doubly Linked List :-
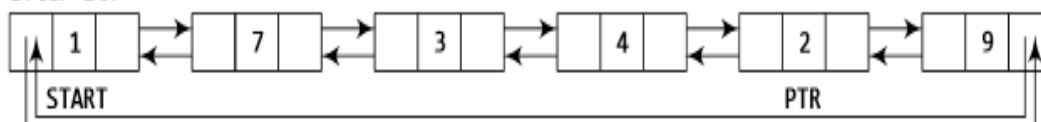

Allocate memory for the new node and initialize its DATA part to 9.

Take a pointer variable PTR that points to the first node of the list.

Move PTR to point to the last node of the list so that the new node can be inserted after it.
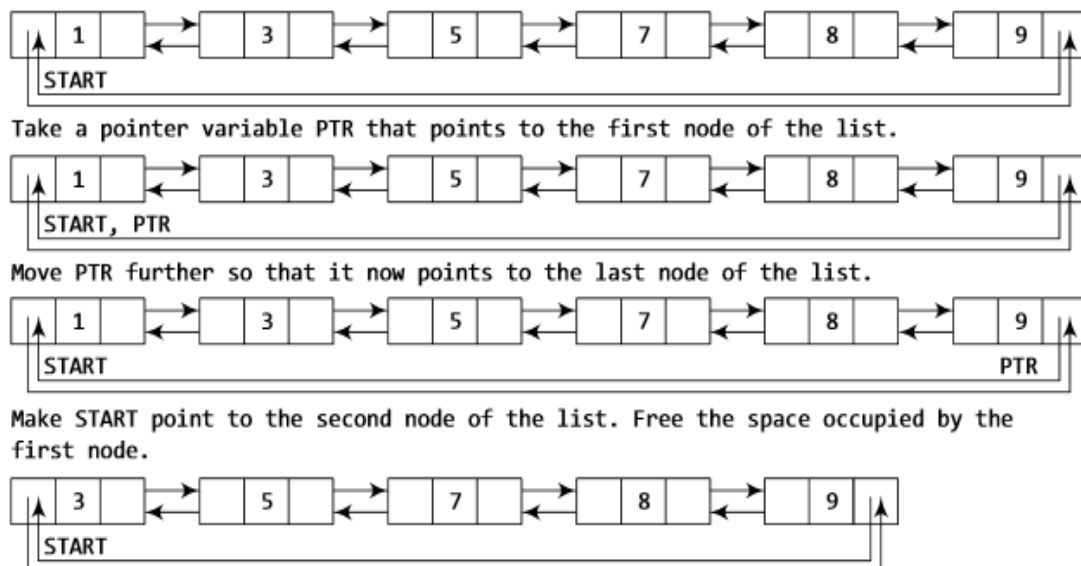
3.6.2. Deleting a Node from a Circular Doubly Linked List:-

In this section, we will see how a node is deleted from an already existing circular doubly linked list. We will take two cases and then see how deletion is done in each case. Rest of the cases are same as that given for doubly linked lists.
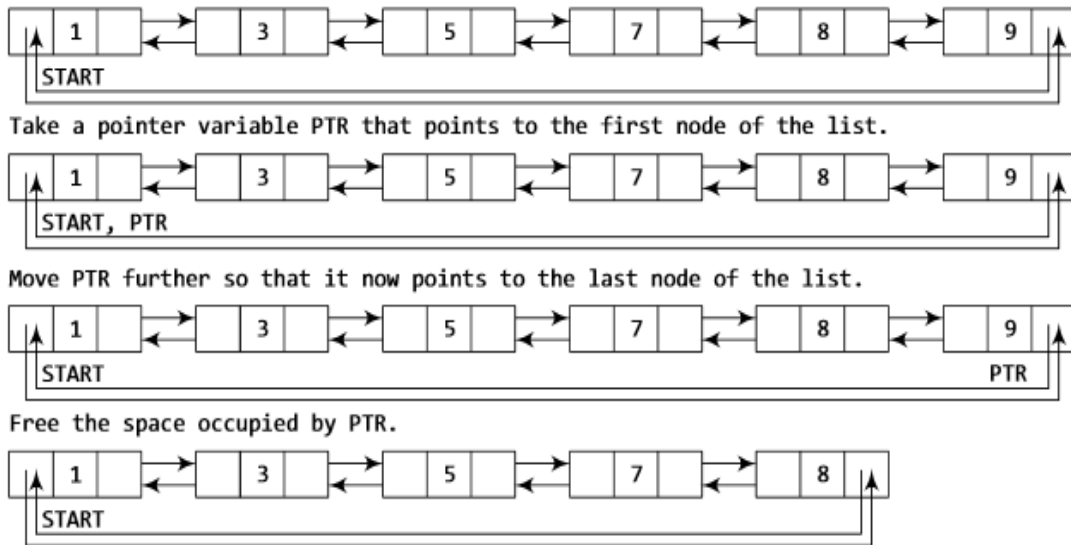
Case 1: The first node is deleted.
Case 2: The last node is deleted.

3.6.2.1. Deleting the First Node from a Circular Doubly Linked List:-

3.6.2.2. Deleting the Last Node from a Circular Doubly Linked List :-

Take a pointer variable PTR that points to the first node of the list.

Move PTR further so that it now points to the last node of the list.

Free the space occupied by PTR.


```
1                  *******************
2                  Programming Example
3                  *******************
   1. Write a program to create a circular doubly linked list and perform
4  insertions and deletions at the beginning and end of the list.
5
6  #include <stdio.h>
7  #include <conio.h>
8  #include <malloc.h>
9
   structnode
10 {
11      structnode *next;
12      intdata;
13      structnode *prev;
14 };
15
   structnode *start = NULL;
16 structnode *create_ll(structnode *);
17 structnode *display(structnode *);
18 structnode *insert_beg(structnode *);
19 structnode *insert_end(structnode *);
20 structnode *delete_beg(structnode *);
   structnode *delete_end(structnode *);
21 structnode *delete_node(structnode *);
22 structnode *delete_list(structnode *);
23
24 intmain()
25 {
26      intoption;
        clrscr();
27      do
28      {
            printf("\n\n *****MAIN MENU *****");
29          printf("\n 1: Create a list");
30          printf("\n 2: Display the list");
            printf("\n 3: Add a node at the beginning");
31          printf("\n 4: Add a node at the end");
32          printf("\n 5: Delete a node from the beginning");
33
```

```
34        printf("\n 6: Delete a node from the end");
35        printf("\n 7: Delete a given node");
36        printf("\n 8: Delete the entire list");
37        printf("\n 9: EXIT");
38        printf("\n\n Enter your option : ");
39        scanf("%d", &option);
40        switch(option)
41        {
42            case1: start = create_ll(start);
43                printf("\n CIRCULAR DOUBLY LINKED LIST CREATED");
44                break;
45            case2: start = display(start);
46                break;
47            case3: start = insert_beg(start);
48                break;
49            case4: start = insert_end(start);
50                break;
51            case5: start = delete_beg(start);
52                break;
53            case6: start = delete_end(start);
54                break;
55            case7: start = delete_node(start);
56                break;
57            case8: start = delete_list(start);
58                printf("\n CIRCULAR DOUBLY LINKED LIST DELETED");
59                break;
60        }
    }while(option != 9);
    getch();
    return0;
}

structnode *create_ll(structnode *start)
{
    structnode *new_node, *ptr;
    intnum;
    printf("\n Enter -1 to end");
    printf("\n Enter the data : ");
    scanf("%d", &num);
    while(num != -1)
    {
        if(start == NULL)
        {
            new_node = (structnode*)malloc(sizeof(structnode));
            new_node ->prev = NULL;
            new_node -> data = num;
            new_node -> next = start;
            start = new_node;
        }
        else
        {
            new_node = (structnode*)malloc(sizeof(structnode));
            new_node -> data = num;
            ptr = start;
            while(ptr -> next != start)
            ptr = ptr -> next;
            new_node ->prev = ptr;
            ptr -> next = new_node;
            new_node -> next = start;
            start ->prev = new_node;
        }
```

```
84          printf("\n Enter the data : ");
85          scanf("%d", &num);
86      }
87      returnstart;
88  }
89  structnode *display(structnode *start)
90  {
91      structnode *ptr;
92      ptr = start;
93      while(ptr -> next != start)
94      {
95          printf("\t %d", ptr -> data);
96          ptr = ptr -> next;
97      }
98      printf("\t %d", ptr -> data);
99      returnstart;
100 }
101 structnode *insert_beg(structnode *start)
102 {
103     structnode *new_node, *ptr;
104     intnum;
105     printf("\n Enter the data : ");
106     scanf("%d", &num);
107     new_node = (structnode *)malloc(sizeof(structnode));
108     new_node-> data = num;
109     ptr = start;
110     while(ptr -> next != start)
111     ptr = ptr -> next;
112     new_node ->prev = ptr;
113     ptr -> next = new_node;
114     new_node -> next = start;
115     start ->prev = new_node;
116     start = new_node;
117     returnstart;
118 }
119 structnode *insert_end(structnode *start)
120 {
121     structnode *ptr, *new_node;
122     intnum;
123     printf("\n Enter the data : ");
124     scanf("%d", &num);
125     new_node = (structnode *)malloc(sizeof(structnode));
126     new_node -> data = num;
127     ptr = start;
128     while(ptr -> next != start)
129     ptr = ptr -> next;
130     ptr -> next = new_node;
131     new_node ->prev = ptr;
132     new_node -> next = start;
133     start->prev = new_node;
134     returnstart;
135 }
136 structnode *delete_beg(structnode *start)
137 {
138     structnode *ptr;
139     ptr = start;
```

```
134    while(ptr -> next != start)
135    ptr = ptr -> next;
136    ptr -> next = start -> next;
137    temp = start;
138    start=start->next;
139    start->prev=ptr;
140    free(temp);
141    returnstart;
142}

143 structnode *delete_end(structnode *start)
144 {
145    structnode *ptr;
146    ptr=start;
147    while(ptr -> next != start)
148    ptr = ptr -> next;
149    ptr ->prev -> next = start;
150    start ->prev = ptr ->prev;
151    free(ptr);
152    returnstart;
153}

154 structnode *delete_node(structnode *start)
155 {
156    structnode *ptr;
157    intval;
158    printf("\n Enter the value of the node which has to be deleted : ");
159    scanf("%d", &val);
160    ptr = start;
161    if(ptr -> data == val)
162    {
163        start = delete_beg(start);
164        returnstart;
165    }
166    else
167    {
168        while(ptr -> data != val)
169        ptr = ptr -> next;
170        ptr ->prev -> next = ptr -> next;
171        ptr -> next ->prev = ptr ->prev;
172        free(ptr);
173        returnstart;
174    }
175 }

176 structnode *delete_list(structnode *start)
177  {
178    structnode *ptr;
    ptr = start;
    while(ptr -> next != start)
    start = delete_end(start);
    free(start);
    returnstart;
  }
```

TREES:
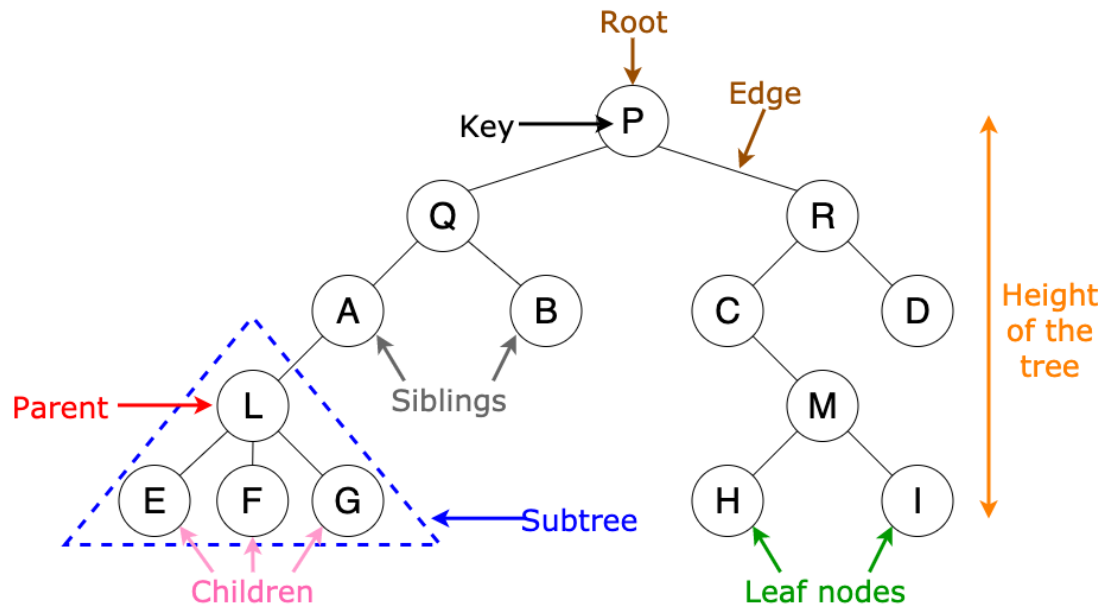
In linear data structure data is organized in sequential order and in non-linear data structure data is organized in random order. A tree is a very popular non-linear data structure used in a wide range of applications.In tree data structure, every individual element is called as **Node**. Node in a tree data structure stores the actual data of that particular element and link to next element in hierarchical structure.

3.7.Tree Definition

Definition: A tree is a finite set of one or more nodes such that: (i) there is a specially designated node called the root; (ii) the remaining nodes are partitioned into n>= 0 disjoint sets T1, ...,Tn where each of these sets is a tree. T1, ...,Tn are called the sub-trees of the root.

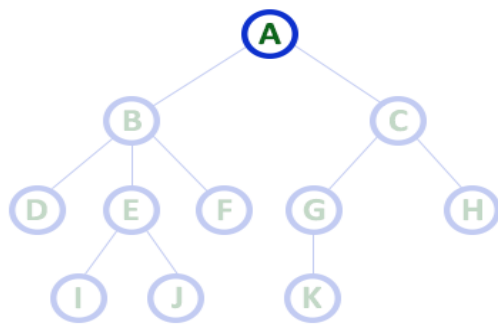In a tree data structure, if we have **N** number of nodes then we can have a maximum of **N-1** number of links.
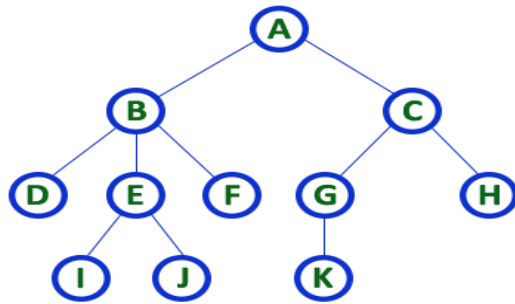
TREE with 11 nodes and 10 edges

- In any tree with 'N' nodes there will be maximum of 'N-1' edges

- In a tree every individual element is called as 'NODE'

3.8.Basic Terminology

- **Root node:** The root node R is the topmost node in the tree. If R = NULL, then it means the tree is empty. Ex: A is the root node.
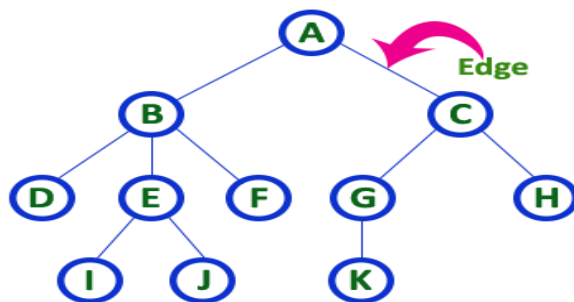
  In a tree data structure, the first node is called as **Root Node**. Every tree must have a root node. We can say that the root node is the origin of the tree data structure. In any tree, there must be only one root node. We never have multiple root nodes in a tree.

Here 'A' is the 'root' node
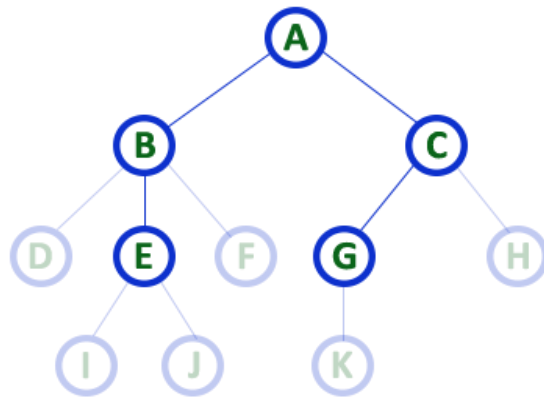
- In any tree the first node is called as ROOT node

- **Edge:**In a tree data structure, the connecting link between any two nodes is called as **EDGE**. In a tree with 'N' number of nodes there will be a maximum of 'N-1' number of edges.

- In any tree, 'Edge' is a connecting link between two nodes.

- **Parent**

In a tree data structure, the node which is a predecessor of any node is called as **PARENT NODE**. In simple words, the node which has a branch from it to any other node is called a parent node. Parent node can also be defined as "**The node which has child / children**".

Here A, B, C, E & G are **Parent** nodes

- In any tree the node which has child / children is called 'Parent'
- A node which is predecessor of any other node is called 'Parent'

- **Child:** In a tree data structure, the node which is descendant of any node is called as **CHILD Node**. In simple words, the node which has a link from its parent node is called as child node. In a tree, any parent node can have any number of child nodes. In a tree, all the nodes except root are child nodes.
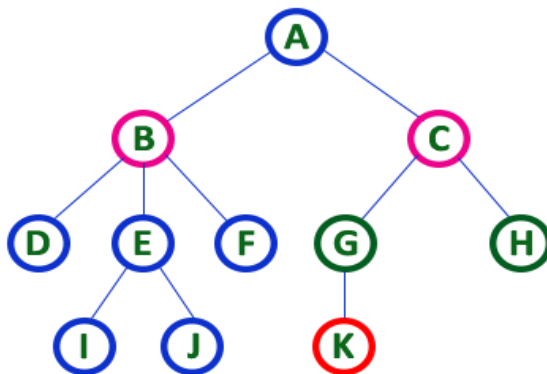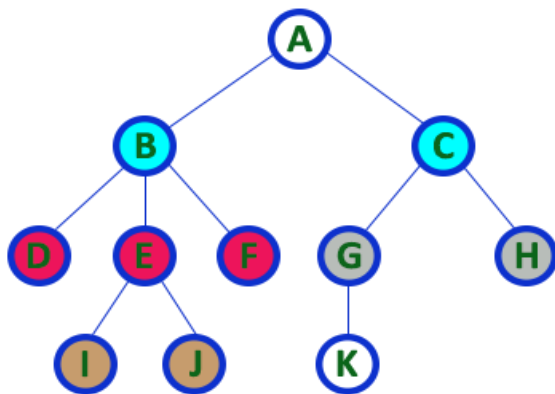

Here B & C are **Children** of A
Here G & H are **Children** of C
Here K is **Child** of G

- descendant of any node is called as **CHILD Node**

- **Siblings:** children of same parent is called siblings.

Here **B & C** are **Siblings**
Here **D E & F** are **Siblings**
Here **G & H** are **Siblings**
Here **I & J** are **Siblings**

- In any tree the nodes which has same Parent are called 'Siblings'

- The children of a Parent are called 'Siblings'

- **Sub-trees:** In a tree data structure, each child from a node forms a subtree recursively. Every child node will form a subtree on its parent node.
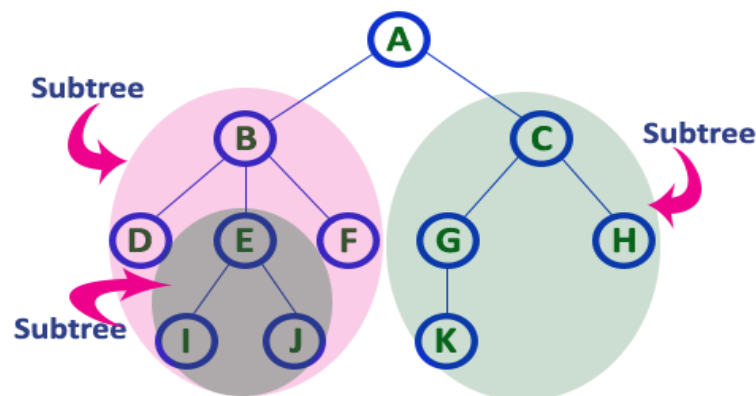
- **Internal Nodes:**

In a tree data structure, nodes other than leaf nodes are called as Internal Nodes. The root node is also said to be Internal Node if the tree has more than one node. Internal nodes are also called as **'Non-Terminal'** nodes.

Non-terminals: The nodes other than leaf nodes are called non terminals.
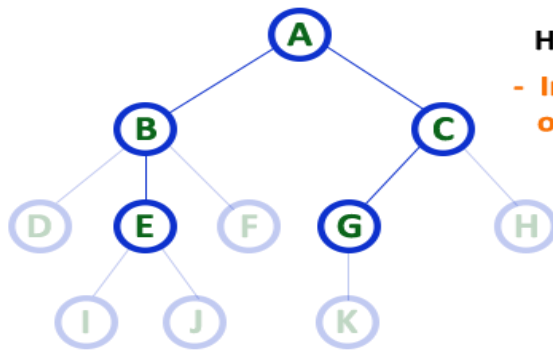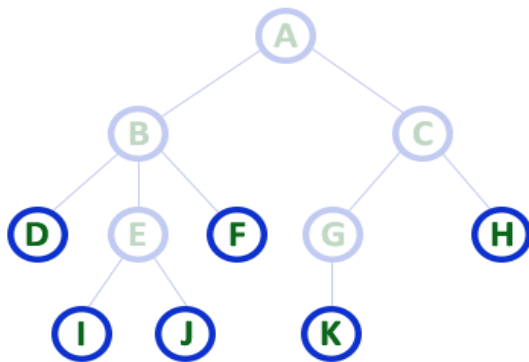
Here A, B, C, E & G are Internal nodes

- In any tree the node which has atleast one child is called 'Internal' node

- Every non-leaf node is called as 'Internal' node

- **Leaf node:** A node that has no children is called the leaf node or the terminal node.

Here D, I, J, F, K & H are Leaf nodes

- In any tree the node which does not have children is called 'Leaf'

- A node without successors is called a 'leaf' node

- **Path:** A sequence of consecutive edges is called a path. In a tree data structure, the sequence of Nodes and Edges from one node to another node is called as **PATH** between that two Nodes. **Length of a Path** is total number of nodes in that path. In below example **the path A - B - E - J has length 4**.
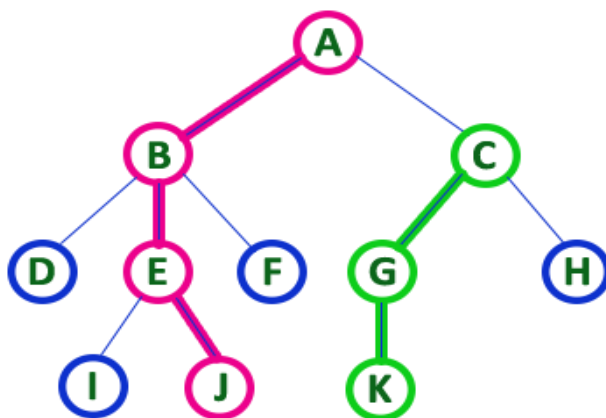
- In any tree, 'Path' is a sequence of nodes and edges between two nodes.

Here, 'Path' between A & J is
A - B - E - J

Here, 'Path' between C & K is
C - G - K

- **Degree**

In a tree data structure, the total number of children of a node is called as **DEGREE** of that Node. In simple words, the Degree of a node is total number of children it has. The highest degree of a node among all the nodes in a tree is called as '**Degree of Tree**'

- **Degree of a node:** Degree of a node is equal to the number of children that a node has. The degree of a leaf node is zero.

- **The degree of a tree :**The degree of a tree is the maximum degree of the nodes in the tree.
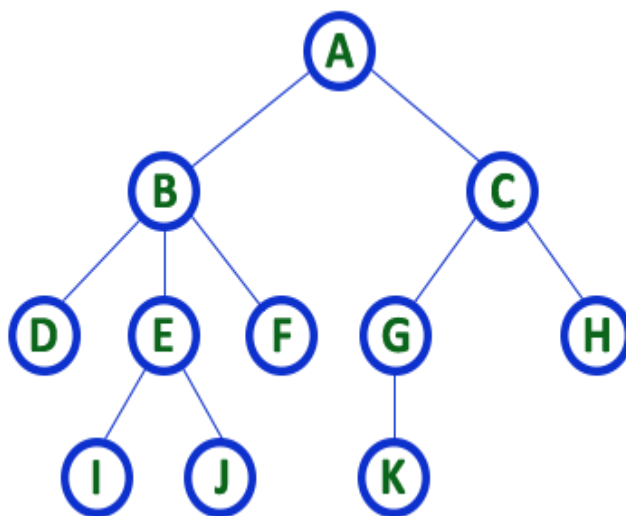
Here **Degree** of B is **3**

Here **Degree** of A is **2**

Here **Degree** of F is **0**

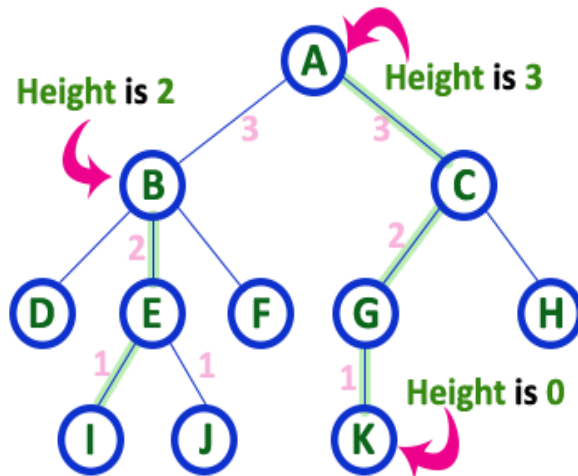- In any tree, 'Degree' of a node is total number of children it has.

- **Level:**Every node in the tree is assigned a level number in such a way that the root node is at level 1, children of the root node are at level number 2. Thus, every node is at one level higher than its parent. So, all child nodes have a level number given by parent's level number + 1.

Level 0

Level 1

Level 2

Level 3
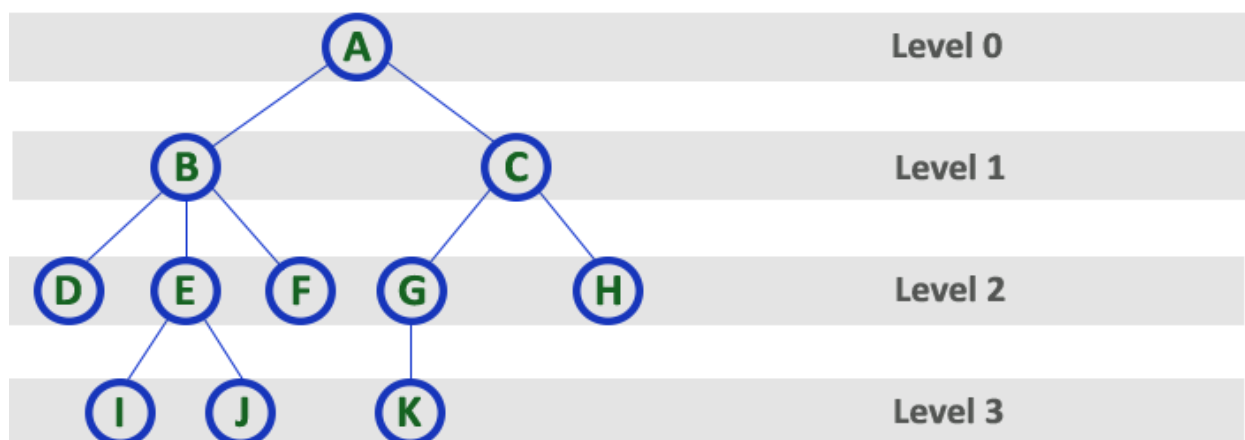
- **Height of the tree:** In a tree data structure, the total number of edges from leaf node to a particular node in the longest path is called as **HEIGHT** of that Node. In a tree, height of the root node is said to be **height of the tree**. In a tree, **height of all leaf nodes is '0'.**

Height of the tree: is the maximum distance between the root node of the tree and the leaf node of the tree.

- **Depth of the tree:** In a tree data structure, the total number of egdes from root node to a particular node is called as **DEPTH** of that Node. In a tree, the total number of edges from root node to a leaf node in the longest path is said to be **Depth of the tree**. In simple words, the highest depth of any leaf node in a tree is said to be depth of that tree. In a tree, **depth of the root node is '0'.**
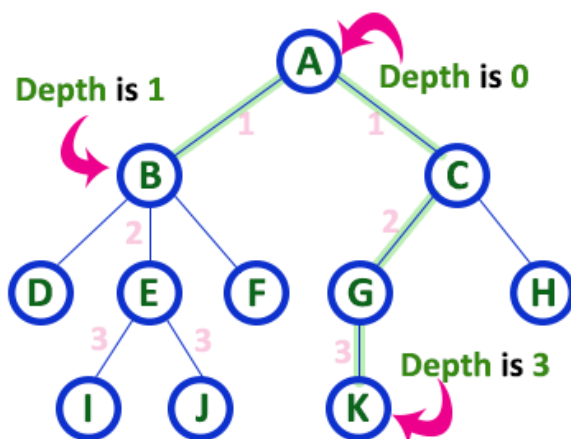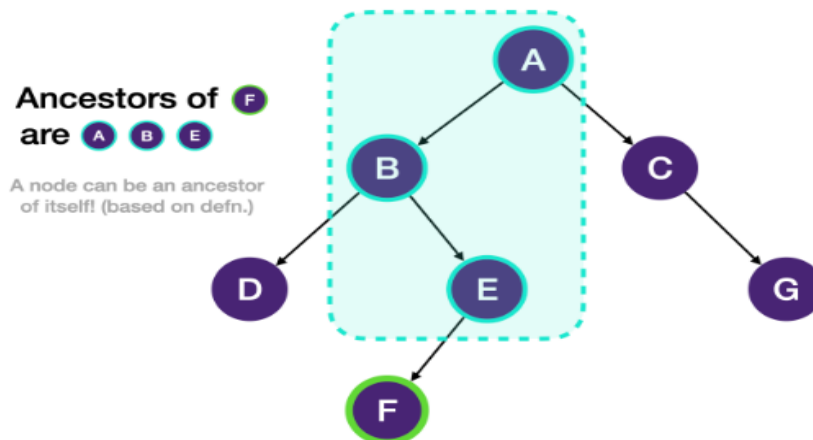
- **Ancestor node:** An ancestor of a node is any predecessor node on the path from root to that node. The root node does not have any ancestors.

Ancestor: nodes that are between the path from the root to the current node

Ancestors of F **are** A B E

A node can be an ancestor of itself! (based on defn.)

- **Descendant node:** A descendant node is any successor node on any path from the node to a leaf node. Leaf nodes do not have any descendants.

Descendent: nodes that are reachable from the current node when moving down the tree

Descendent of B **are** D E F

A node can be a descendent of itself! (based on defn.)
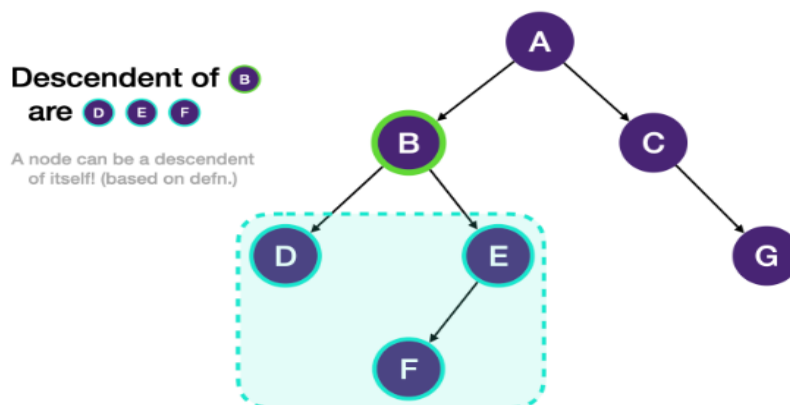
- **In-degree:** In-degree of a node is the number of edges arriving at that node.
- **Out-degree:** Out-degree of a node is the number of edges leaving that node.

3.9.Binary Trees

- A binary tree is an important type of tree structure which occurs very often. It is characterized by the fact that any node can have at most *two branches*, i.e., there is no node with degree greater than two. For binary trees we distinguish between the sub-tree on the left and on the right, whereas for trees the order of the sub-trees was irrelevant. Also a binary tree may have zero nodes. Thus a binary tree is really a different object than a tree.

- **Definition:** A binary tree is a finite set of nodes which is either empty or consists of a root and two disjoint binary trees called the left subtree and the right subtree.
- The distinctions between a binary tree and a tree should be analyzed. First of all there is no tree having zero nodes, but there is an empty binary tree. The two binary trees in Fig. 3.9 are different. The first one has an empty right subtree while the second has an empty left subtree. If the above are regarded as trees, then they are the same despite the fact that they are drawn slightly differently.
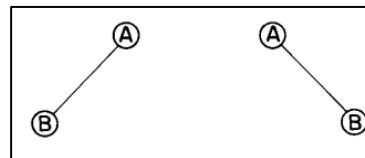

Fig. 3.9: binary trees example

3.9.1. Difference between General tree and Binary tree

General tree	Binary tree
General tree is a tree in which each node can have many children or nodes.	Whereas in binary tree, each node can have at most two nodes.
The subtree of a general tree do not hold the ordered property.	While the subtree of binary tree hold the ordered property.
In data structure, a general tree can not be empty.	While it can be empty.
In general tree, a node can have at most **n(number of child nodes)** nodes.	While in binary tree, a node can have at most **2(number of child nodes)** nodes.
In general tree, there is no limitation on the degree of a node.	While in binary tree, there is limitation on the degree of a node because the nodes in a binary tree can't have more than two child node.
In general tree, there is either zero subtree or many subtree.	While in binary tree, there are mainly two subtree: **Left-subtree** and **Right-subtree**.

3.10. Different kind Binary Trees:

1. **Skewed Tree** :A skewed tree is a tree, skewed to the left or skews to the right. or It is a tree consisting of only left sub-tree or only right sub-tree. Figure 3.10(a) shows the sample of skewed Tree.

   - A tree with only left sub-trees is called Left Skewed Binary Tree.
   - A tree with only right sub-trees is called Right Skewed Binary Tree.

2. **A binary tree T is said to complete** if all its levels, except possibly the last level, have the maximum number node 2^i , i ≥ 0 and if all the nodes at the last level appears as far left as possible.Tree 4.3b is called a *complete binary tree*. Notice that all terminal nodes are on adjacent levels. The terms that we introduced for trees such as: degree, level, height, leaf. parent, and child all apply to binary trees in the natural way.
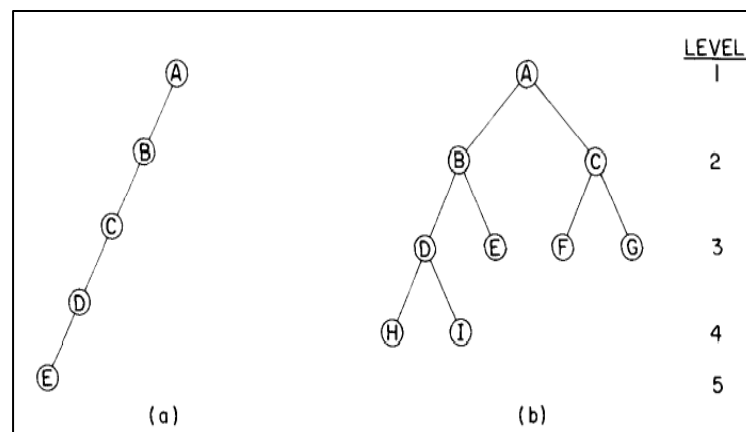

Fig. 3.10: Two sample binary trees

3. **Full Binary Tree**: A full binary tree of depth 'k' is a binary tree of depth k having   2k – 1 nodes, k ≥ 1.
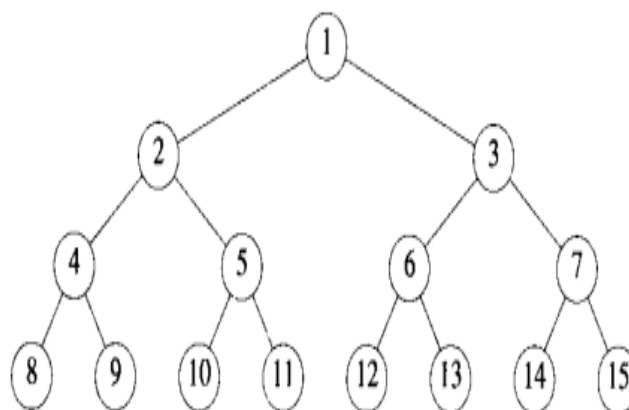

Figure: Full binary tree of level 4 with sequential node number

3.11.Properties of Binary Tree

Lemma 1& 2: [Maximum number of nodes]:

(1) The maximum number of nodes on level i of a binary tree is 2^{i-1}, i ≥1.

(2) The maximum number of nodes in a binary tree of depth k is $2^k - 1$, k ≥ 1

Proof:

Step 1: The proof is by induction on i. **Induction Base:** The root is the only node on level i = 1. Hence, the maximum number of nodes on level i =1 is $2^{i-1} = 2^0 = 1$.

Step 2: Induction Hypothesis: Let i be an arbitrary positive integer greater than 1. Assume that the maximum number of nodes on level i -1 is 2^{i-2}

Step 3: Induction Step: The maximum number of nodes on level i-1 is 2^{i-2} by the induction hypothesis. Since each node in a binary tree has a maximum degree of 2, the maximum number of nodes on level i is two times the maximum number of nodes on level i-1, i.e or $2*2^{i-2} = 2^{i-1}$

(2) The maximum number of nodes in a binary tree of depth k is $\sum_{i=1}^{k} = \sum_{i=1}^{k} 2^{i-1} = 2^{k-1}$

Lemma– 3:

For any nonempty binary tree, T, if no is the number of terminal nodes and n2 the number of nodes of degree 2, then $n_0 = n_2 + 1$.

Proof:

Consider the binary tree in the Fig. 3.11. each node in the tree should have a maximum of 2 children. A node may not have any child or it can have single child or it can have 2 children. But a node in a binary tree can not have more that 2 children.
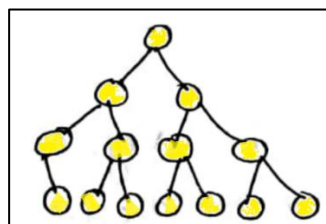

Fig. 3.11: binary tree

Let   No of nodes of degree 0 = n_0

        No of nodes of degree 1 = n_1

        No of nodes of degree 2 = n_2

So total no. of nodes in the tree is = $n_0 + n_1 + n_2$ ………………………………..1

Observe from the tree that the total no of nodes is equal to the total no. of branches (B) plus 1

n = B+1 …………………………………………………………………..2

if there is a node with degree 1, no. of branches $= 1$

so for n_1, no. of nodes with degree 1, no. of branches $= 1n_1$………………………3

if there is a node with degree 2, no. of branches $= 2$

so for n_2, no. of nodes with degree 2, no. of branches $= 2n_2$………………………4

add 3 and 4, we get

$B = 1n_1 + 2n_2$ …………………………………………………………………………5

Substitute 5 in 2 we get

$n = 1n_1 + 2n_2 + 1$ …………………………………………………………………….6

from 1 and 6 we get

$n_0 + n_1 + n_2 = 1n_1 + 2n_2 + 1$

$\Rightarrow n_0 + n_1 + n_2 - 1n_1 - 2n_2 - 1 = 0 \Rightarrow n_0 - n_2 - 1 = 0 \Rightarrow n_0 = n_2 + 1$

Hence it is proved that for any nonempty binary tree, T, if no is the number of terminal nodes and n2 the number of nodes of degree 2, then $n_0 = n_2 + 1$.

3.12.Binary Tree Representation

There are 2 ways of representations

1. **Array representation**
   - If T is a binary tree, then it can be maintained in memory using sequential representation as follows:
     a. Root R of tree T is stored in TREE[1]
     b. If a node n occupies TREE[k], then its left child is stored in TREE[2*k] and right child is stored in TREE[2*k+1]
     c. NULL is used to represent empty sub-tree
     d. If TREE[1] is NULL, then it is an empty tree.
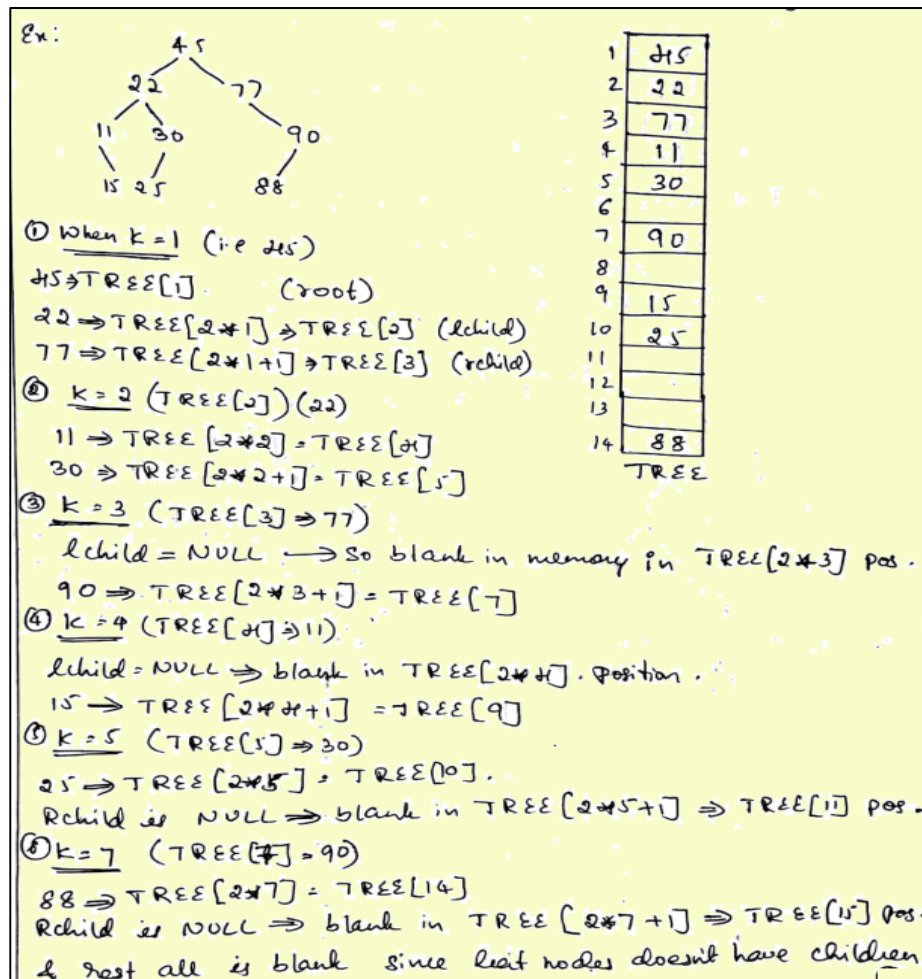   - Example is given in the Fig.3.12.1.

Fig.3.12.1: Example for array representation of binary tree

2. Linked representation:

- While the above representation appears to be good for complete binary trees it is wasteful for many other binary trees. In addition, there presentation suffers from the general inadequacies of sequential representations. Insertion or deletion of nodes from the middle of a tree requires the movement of potentially many nodes to reflect the change in level number of these nodes.

- These problems can be easily overcome through the use of a linked representation. Each node will have three fields LCHILD, DATA and RCHILD as in Fig.3.12.2.
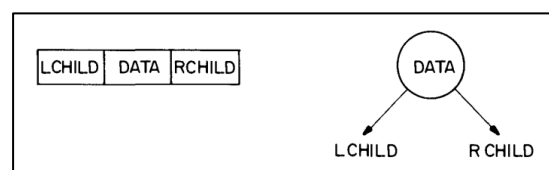

Fig.3.12.2: Linked representation of binary tree

- Ex: Consider a binary tree in the Fig. 3.12.3: The linked representation of this tree is given in the Fig. 3.12.3.(b).it can be represented in the memory as shown
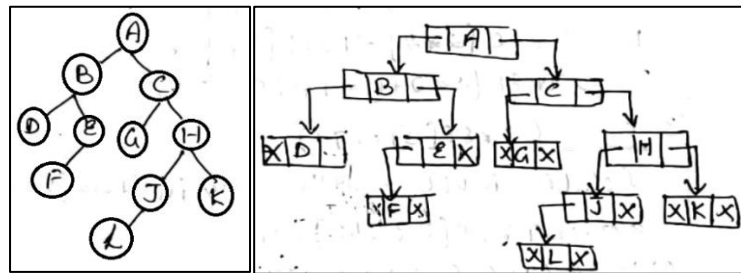

Fig. 3.12.3(a): Example for Linked representation of binary tree

Fig. 3.12.3(b): memory representation of binary tree

3.13. Binary Tree Traversal

There are 3 types of binary tree traversal

3.13.1.Preorder traversal

To traverse a non-empty binary tree in pre-order, the following operations are performed recursively at each node.

The algorithm works by:

1. Visiting the root node,

2. Traversing the left sub-tree, and finally

3. Traversing the right sub-tree.

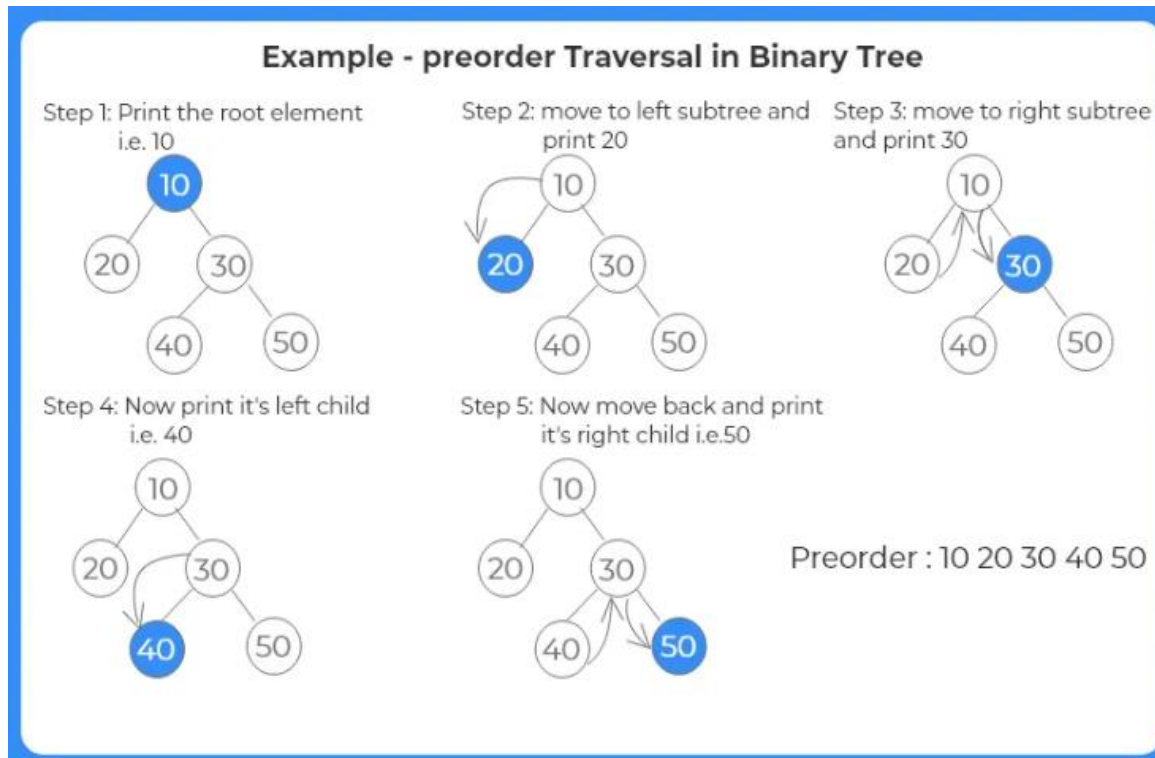**Function code:**

```
void preorder(struct treeNode *node)
{
        if (node != NULL)
        {
                printf("%d", node->data);
                preorder(node->left);
                preorder(node->right);
```

```
        }
        return;
}
```


Example - preorder Traversal in Binary Tree

Preorder : 10 20 30 40 50

3.13.2. Postorder traversal

To traverse a non-empty binary tree in post-order, the following operations are performed recursively at each node. The algorithm works by:

    1. Traversing the left sub-tree,

    2. Traversing the right sub-tree.

    3. Visiting the root node

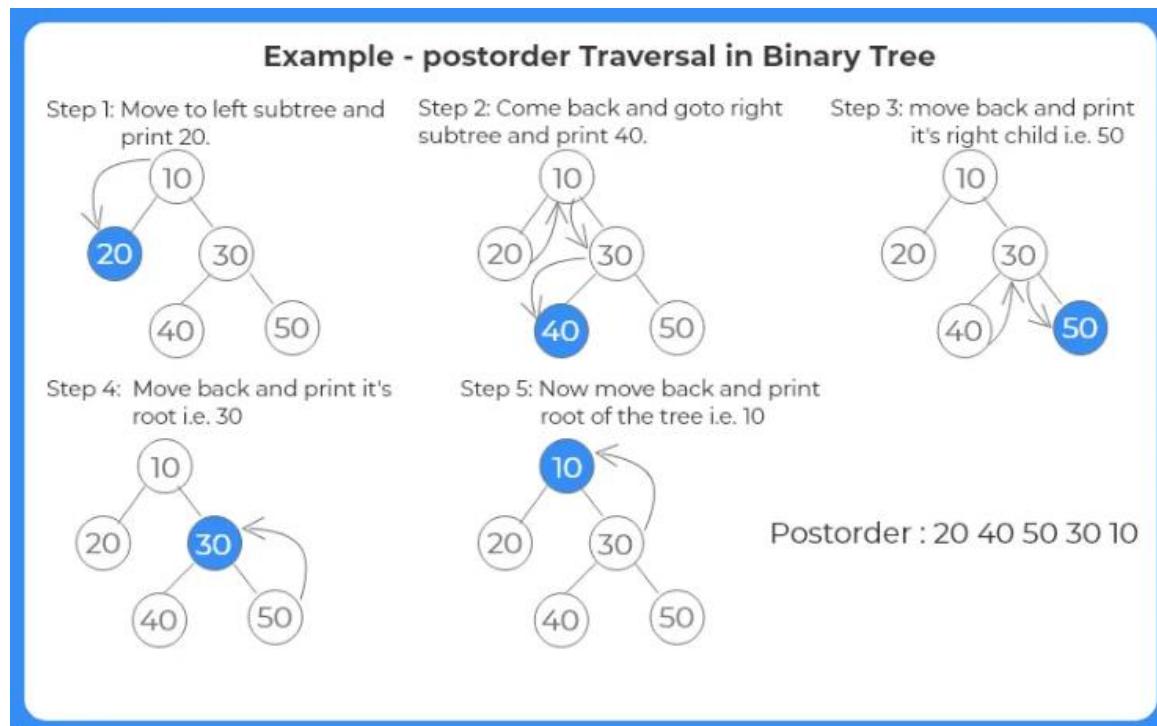**Function code:**

```
void postorder(struct treeNode *node)
{
        if (node != NULL)
        {
                postorder(node->left);
                postorder(node->right);
                printf("%d", node->data);
        }
        return;
```

}

3.13.3.Inorder traversal

To traverse a non-empty binary tree in inorder, the following operations are performed recursively at each node. The algorithm works by:

    1. Traversing the left sub-tree,

    2. Visiting the root node

    3. Traversing the right sub-tree.

Function code:

```
void inorder(struct treeNode *node)
{
        if (node != NULL)
        {
                inorder(node->left);
                printf("%d", node->data);
                inorder(node->right);
        }
        return;
}
```

Example - Inorder Traversal in Binary Tree
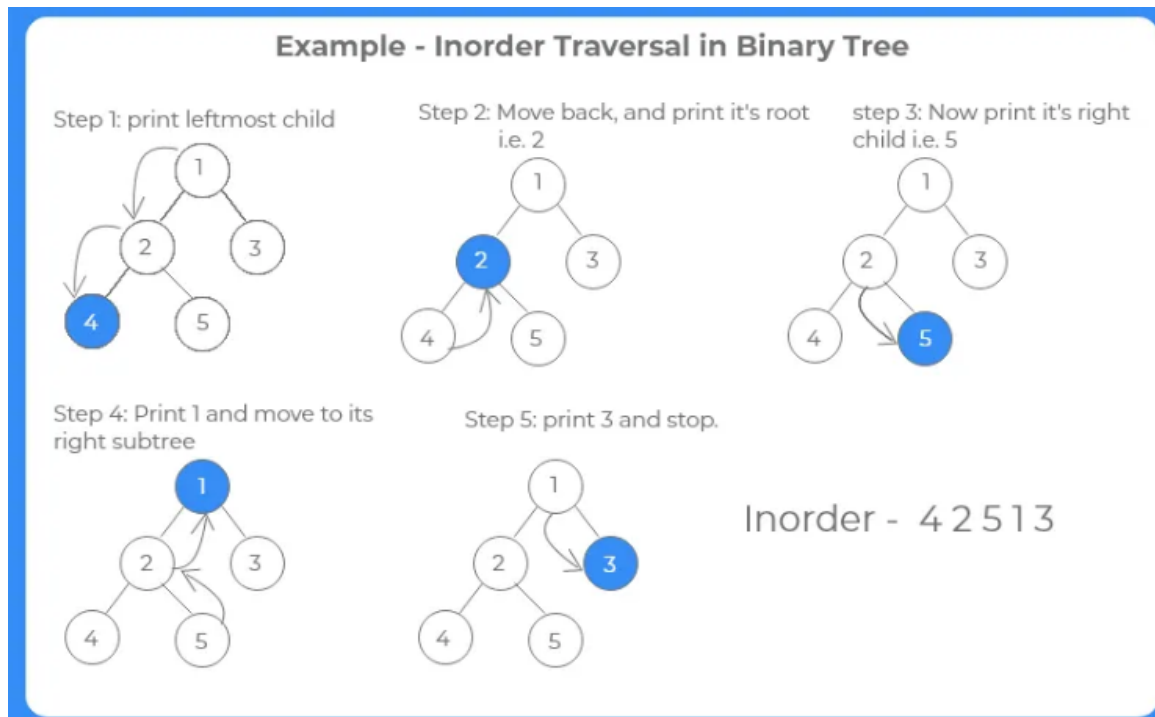
Constructing a Binary Tree from Traversal Results

- We can construct a binary tree if we are given at least two traversal results.

- The first traversal must be the in-order traversal and the second can be either pre-order or post-order traversal.

- The in-order traversal result will be used to determine the left and the right child nodes, and the pre-order/post-order can be used to determine the root node.

Example1: In–order Traversal: DJHBEAFICG   Post–order Traversal: JHDEBIFGCA

Construct the tree:

> **Step 1**: Use the post-order sequence to determine the root node of the tree. The first element would be the root node.
>
> **Step 2:** Elements on the left side of the root node in the in-order traversal sequence form the left sub-tree of the root node. Similarly, elements on the right side of the root node in the in-order traversal sequence form the right sub-tree of the root node
>
> **Step 3:** Recursively select each element from post-order traversal sequence and create its left and right sub-trees from the in-order traversal sequence. Look at Fig.3.13.3 be which constructs the tree from its traversal results. Now consider the in-order traversal and post-order traversal sequences of a given binary tree.

Post order **JHDEBIFGCA** and In order: **DJHBEAFICG**

Action	Parent	Left part in Inorder	Right Part in Inorder	Partial Tree
Find Root in Post-order	A	DJHBE	FICG	
Consider Left subtree of A i.,e DJHBE in Postorder and identify the parent	B	DJH	E	
Consider left subtree of B I,e JHD and identify the parent in postorder	D	NULL	JH	
Consider Right subtree of D, i.e JH. identify the parent in postorder	H	J	NULL	
Consider Right subtree of A, FICG Identify the parent node in postorder, C	C	FI	G	
Consider left subtree of C, FI. Identify parent in postorder	F	I	NULL	

Fig 3.13.3:Construction of Binary Tree

Example 2:

In order: **1 2 3 5 9 19 21 25** Pre order: **5 2 1 3 19 9 21 25**

| Action	Parent	Left part in Inorder	Right Part in Inorder	Partial Tree
Find Root in Pre-order	5	1 2 3	59 19 21 25	
Consider Left subtree of 5 i.,e 123 in Preorder and identify the root	2	1	3	
Consider right subtree of 5 I,e 9,19,21,25 and identify the parent in preorder	19	9	21 25	
Consider Right subtree of 19,i.e 21 25. identify the parent in preorder	21	NULL	25	

Fig 3.13.3: Construction of Binary Tree

3.14.Iterative Inorder traversal (or) Inorder Tree Traversal without Recursion.

Here we can use a stack to perform inorder traversal of a Binary Tree. Below is the algorithm for traversing a binary tree using stack.

- Create an empty stack (say S).
- Initialize the current node as root.
- Push the current node to S and set current = current->left until current is NULL
- If current is NULL and the stack is not empty then:
    - Pop the top item from the stack.
    - Print the popped item and set current = popped_item->right
    - Go to step 3.
- If current is NULL and the stack is empty then we are done.

Example1:
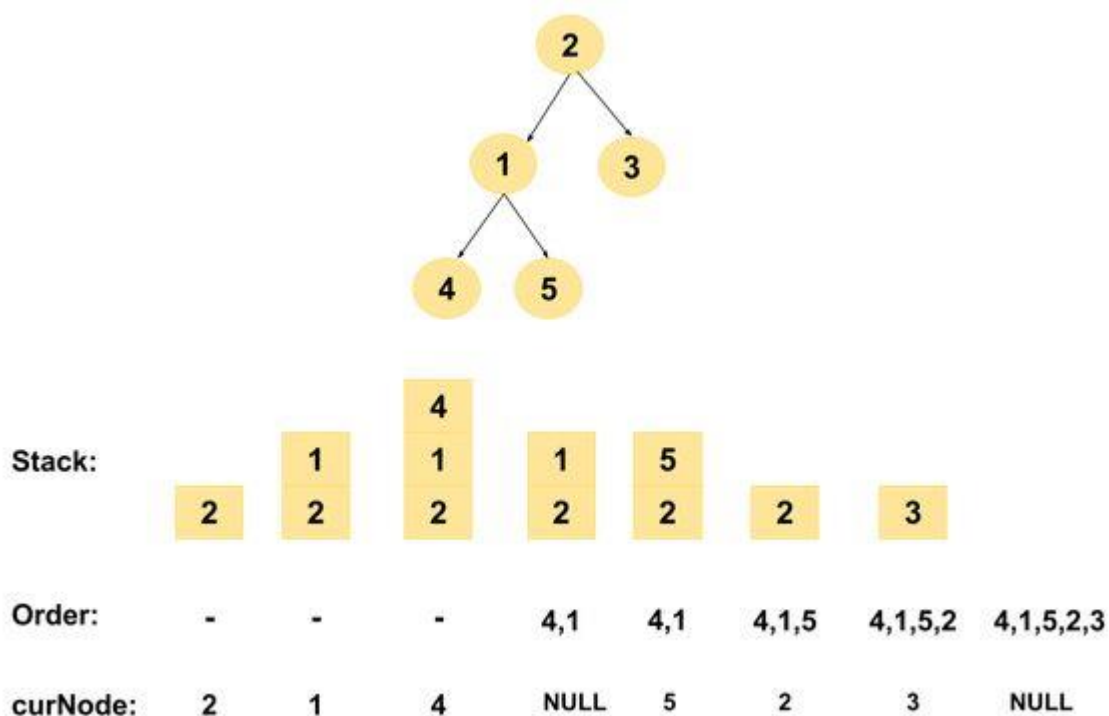
Example2:

Let us consider the below tree for example

```
        1
      /   \
     2     3
    / \
   4   5
```

Initially Creates an empty stack: S = NULL and set current as address of root: **current -> 1**

current = 1: Pushes the current node and set current = current->left until current is NULL:

- current -> 1, push 1: Stack S -> 1
- current -> 2, push 2: Stack S -> 2, 1
- current -> 4, push 4: Stack S -> 4, 2, 1
- current = NULL

Now Pop from S

- Pop 4: Stack S -> 2, 1. **Print "4"**.
- current = NULL /*right of 4 */ and go to step 3. Since current is NULL step 3 doesn't do anything.

Step 4 is repeated and pop again.

- Pop 2: Stack S -> 1. **Print "2"**.
- current -> 5 /*right of 2 */ and go to step 3

current = 5: Push 5 to stack and make current = current->left which is NULL

- Stack S -> 5, 1. current = NULL

Now pop from S

- Pop 5: Stack S -> 1. **Print "5"**.
- current = NULL /*right of 5 */ and go to step 3. Since current is NULL step 3 doesn't do anything

Step 4 repeated again:

- Pop 1: Stack S -> NULL. **Print "1"**.
- current -> 3 /*right of 1 */

current = 3: Pushes 3 to stack and make current NULL

- Stack S -> 3
- current = NULL

Step 4 pops from S:

- Pop 3: Stack S -> NULL. **Print "3"**.
- current = NULL /*right of 3 */

Now the traversal is complete as the stack has become empty.

Example2:
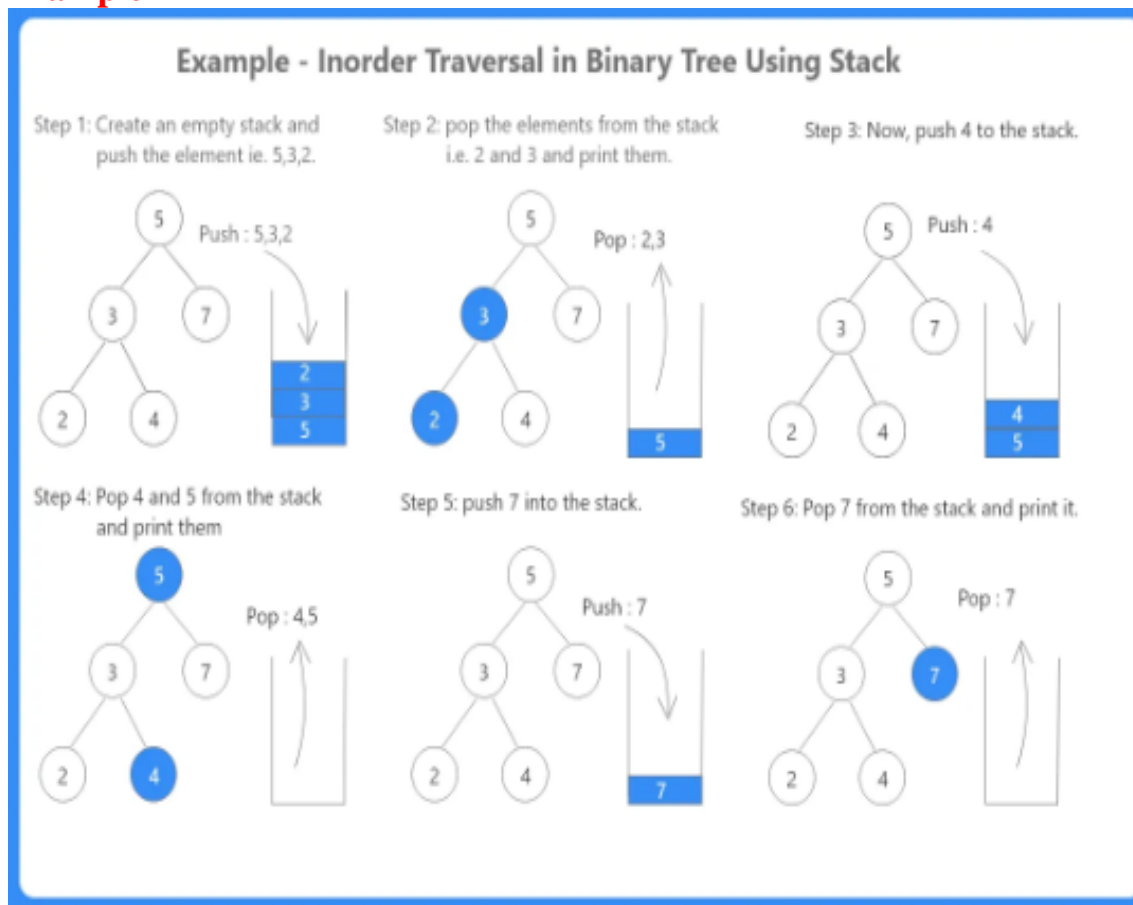
3.15. Level Order Traversal

- **Level Order Traversal** technique is defined as a method to traverse a Tree such that all nodes present in the same level are traversed completely before traversing the next level.
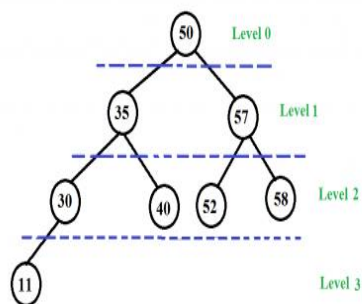
In level order traversal nodes of tree are traversed level-wise from left to right.

Queue data structure is used to store nodes level wise so that each node's children are visited.

```c
void levelOrder(struct Node *ptr) {

    // Queue structure to store pointer
    struct Queue q;

    // Print the root node as it is on the topmost level
    printf("%d\t", ptr -> data);
    // Enqueue root node to the queue
    enqueue(&q, ptr);

    while (!isEmpty(q)) {
        // Obtain root of subtree
        ptr = dequeue(&q);
        // Print & enqueue the left child of next level if present
        if (ptr -> left) {
            printf("%d\t", ptr -> left -> data);
            enqueue(&q, ptr -> left);
        }
        // Print & enqueue the left child of next level if present
        if (ptr -> right) {
            printf("%d\t", ptr -> right -> data);
            enqueue(&q, ptr -> right);
        }
    }

}
```
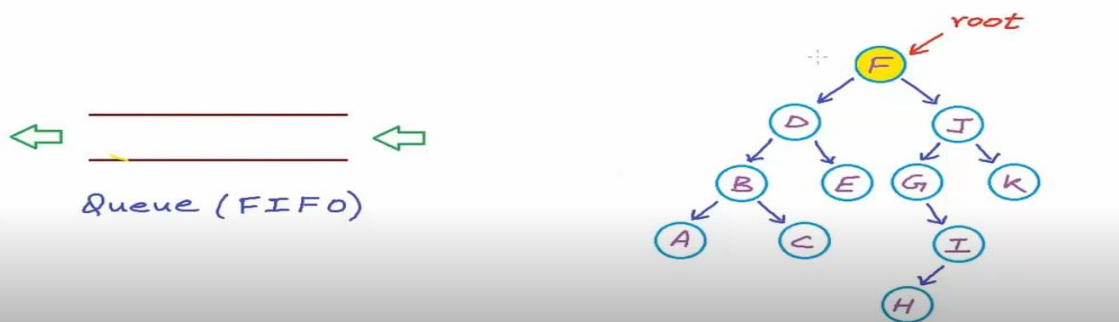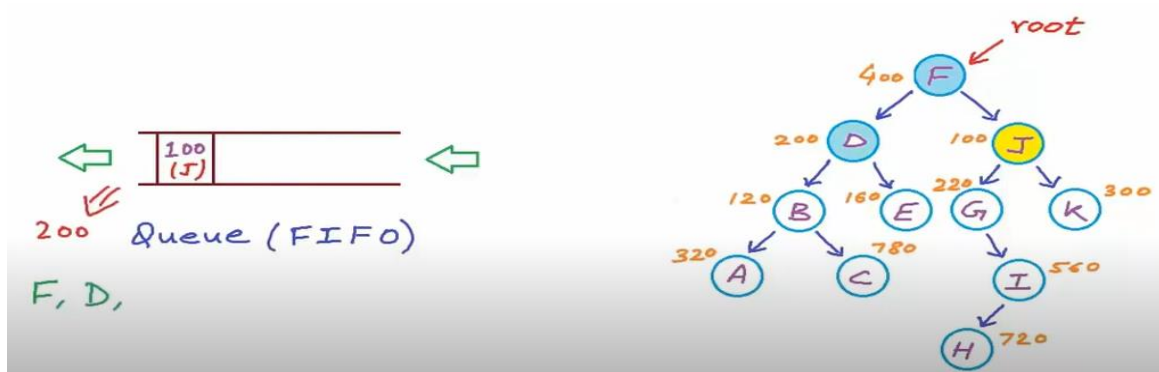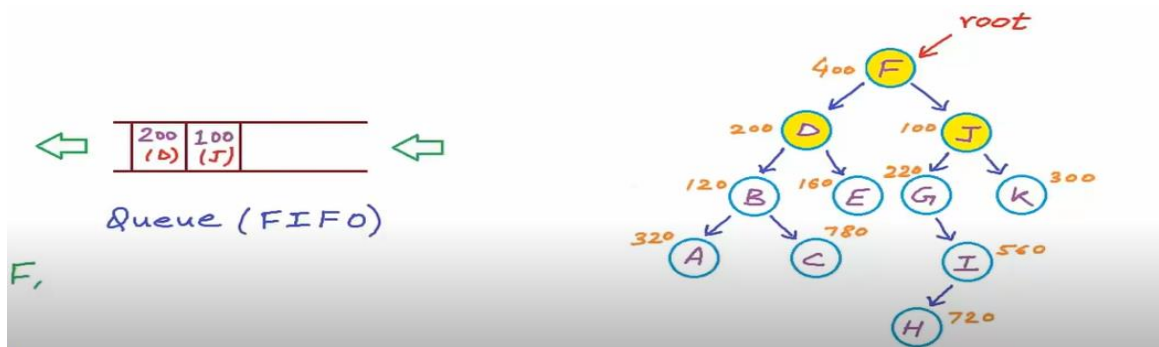
- At level 0, we process the root node first, followed by the left and right children at level 1. (assuming the order from left to right).
- Similarly, at the second level, we first process the children of the left child of the root then process the children of the right child. This process goes on for all the levels in the tree.

120 (B)	160 (E)	220 (G)	300 (K)

100

Queue (FIFO)

F, D, J,

100 (J)	120 (B)	160 (E)	

Queue (FIFO)

F, D,

160 (E)	220 (G)	300 (K)	320 (A)	780 (C)

120

Queue (FIFO)

F, D, J, B,

300 (K)	320 (A)	780 (C)	560 (I)

Queue (FIFO)

F, D, J, B, E, G
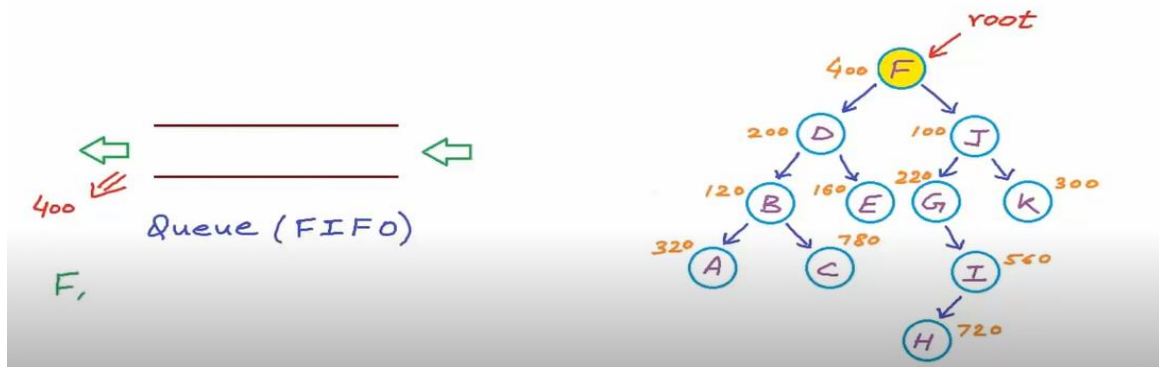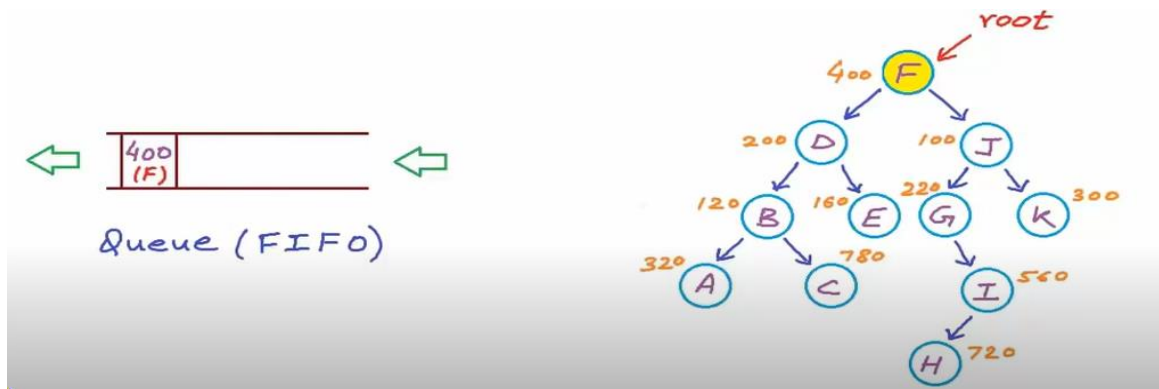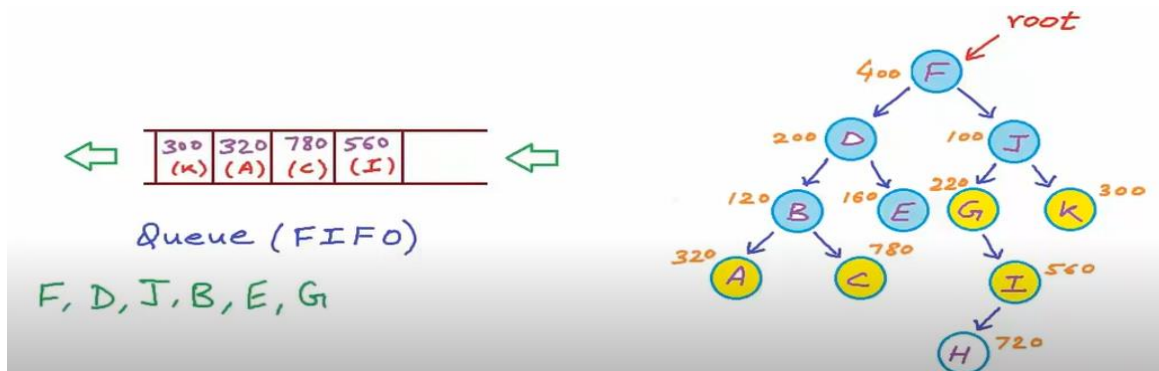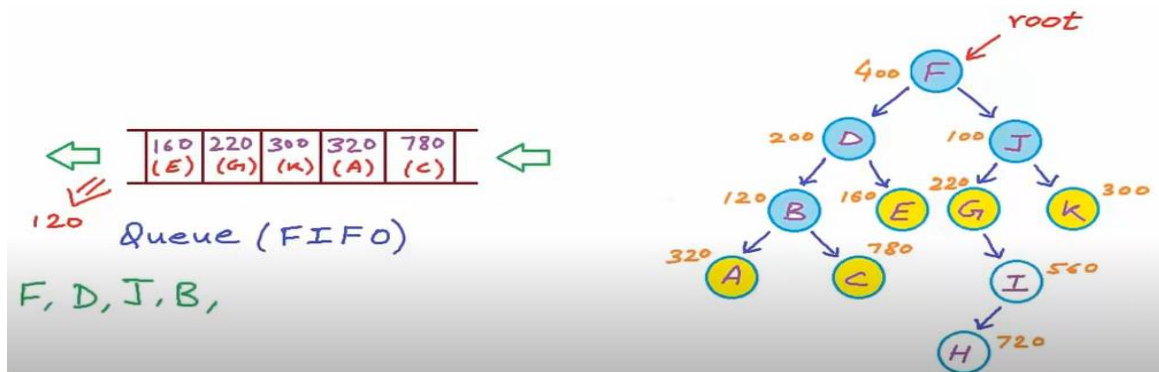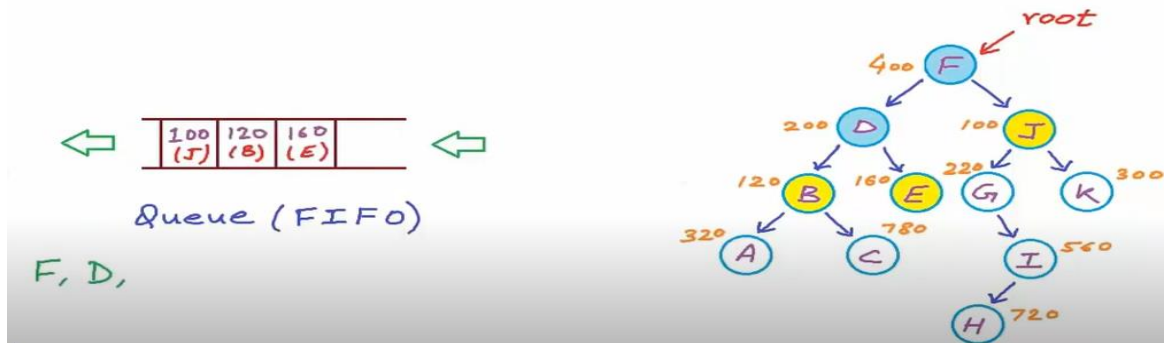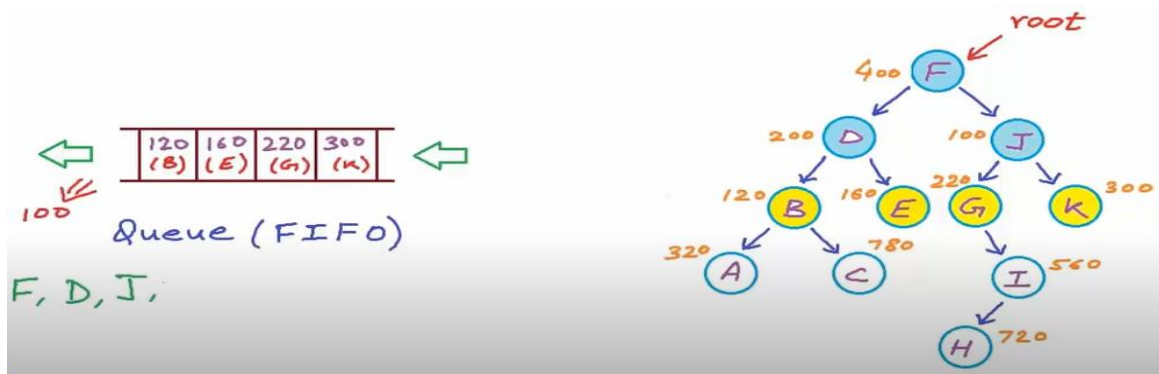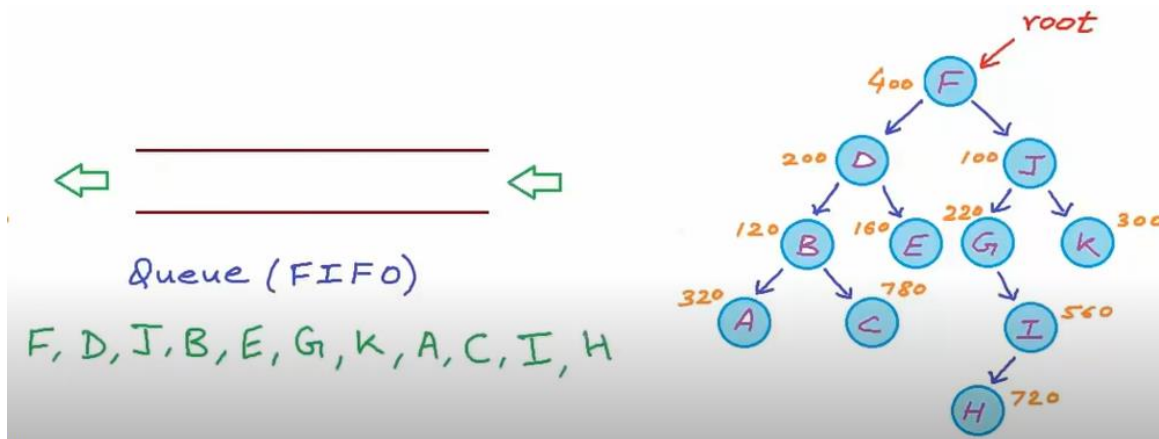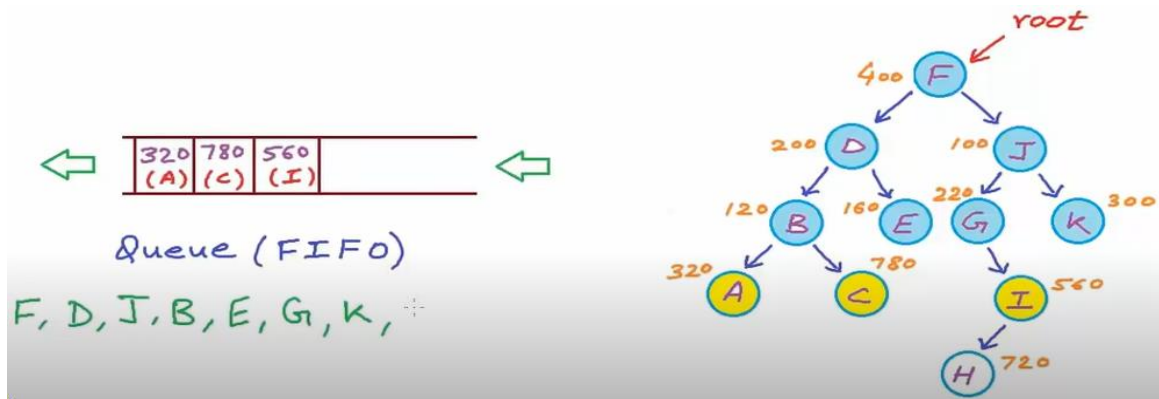
3.16. Threaded Binary Trees

The limitations of binary tree are:

- In binary tree, there are n+1 null links out of 2n total links.
- Traversing a tree with binary tree is time consuming.

These limitations can be overcome by threaded binary tree

- In the linked representation of any binary tree, there are more null links than actual pointers. These null links are replaced by the pointers, called **threads,** which points to other nodes in the tree.

To construct the threads use the following rules:

- Assume that ptr represents a node. If ptr→leftChild is null, then replace the null link with a pointer to the inorder predecessor of ptr.
- If ptr →rightChild is null, replace the null link with a pointer to the inorder successor of ptr.

  Ex: Consider the binary tree as shown in below figure:

Figure A: Binary Tree

- There should be no loose threads in threaded binary tree. But in Figure B two threads have been left dangling: one in the left child of H, the other in the right child of G.

Figure B: Threaded tree corresponding to Figure A

- In above figure the new threads are drawn in broken lines. This tree has 9 node and 10 Null-links which has been replaced by threads. When trees are represented in memory, it should be able to distinguish between threads and pointers. This can be done by adding two additional fields to node structure, ie., leftThread and rightThread

- If ptr→leftThread = TRUE, then ptr→leftChild contains a thread, otherwise it contains apointer to the left child.

- If ptr→rightThread = TRUE, then ptr→rightChild contains a thread, otherwise it contains pointer to the right child.

Node Structure: The node structure is given in C declaration

```
typedef struct threadTree *threadPointer
typedefstruct
{
short int leftThread;
threadPointer leftChild;
 char data;
threadPointer rightChild;
 short int rightThread;
}threadTree;
```

Figure An empty threaded tree

- The complete memory representation for the tree of figure is shown in Figure C .
- The variable root points to the header node of the tree, while root →leftChild points to the start of the first node of the actual tree.
- This is true for all threaded trees. Here the problem of the loose threads is handled by pointing to the head node called root.

Question Bank

.

1. Give a node structure to create a doubly linked list of integers and write a C function to perform the following.

    a. Create a three-node list with data 10, 20 and 30

   b. Inert a node with data value 15 in between the nodes having data values 10 and 20

    c. Delete the node which is followed by a node whose data value is 20

    d. Display the resulting singly linked list..

2. Write a note on: i. Linked representation of sparse matrix ii. Doubly linked list.

3. Write a function to insert a node at front and rear end in a circular linked list. Write down sequence of steps to be followed.

10. Write a C program to perform the following operations on doubly linked list: i. Insert a node ii. Delete a node.

11. Write a C function to insert a node at front and delete a node from the rear end in a circular linked list.

12. Describe the doubly linked lists with advantages and disadvantages. Write a C function to delete a node from a circular doubly linked list with header node.

13. Write a C function for the concatenation of linked lists.

14. Write a C function to perform the following i. Reversing a singly linked list ii. Concatenating singly linked list. iii. Finding the length of the circular linked list. iv. To search an element in the singly linked list

15. Write a node structure of linked stack. Write a function to perform push and pop operations on linked stack.

16. List out the differences between doubly linked list over singly linked list. Write a C functions to perform the following i. Inserting a node into a doubly linked circular list ii. Deletion from a doubly linked circular list.

17. Given 2 singly linked lists. LIST-1 and LIST-2. Write an algorithm to form a new list LIST-3 using concatenation of the lists LIST-1 and LIST-2.

18. Write a note on header linked list. Explain the widely used header lists with diagrams. 23. Illustrate with examples how to insert a node at the beginning, INSERT a node at intermediate position, DELETE a node with a given value

19. List out any 2 differences between doubly linked lists and singly linked list, Illustrate with example the following operations on a doubly linked list: i. Inserting a node at the beginning. ii. Inserting at the intermediate position. iii. Deletion of a node with a given value

20. For the given sparse matrix write the diagrammatic linked list representation

$$A = \begin{bmatrix} 5 & 0 & 0 & 0 \\ 3 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 8 & 0 & 0 & 2 \\ 0 & 0 & 7 & 0 \end{bmatrix}$$

21. Define the following
    a. Tree
    b. Node
    c. Root
    d. Degree
    e. Leaf nodes
    f. Non terminals
    g. Children
    h. Sibling
    i. Grand parent
    j. Degree of the tree
    k. Ancestor
    l. Level
    m. Depth of tree

22. Define binary trees. Explain different properties of binary trees
23. Explain 2 types of representations of binary trees
24. Explain the traversing of binary trees
25. Determine inorder, preorder and postorder traversal of the following tree. Write the C-routines
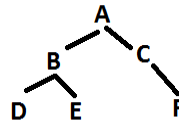
26. Suppose following sequence lists the node of binary tree in preorder and inorder respectively, draw diagram of the tree:
    Preorder: G B Q A C K F P D E R H
    Inodrer: Q B K C F A G P E D H R
27. What is threaded binary tree? Explain with example