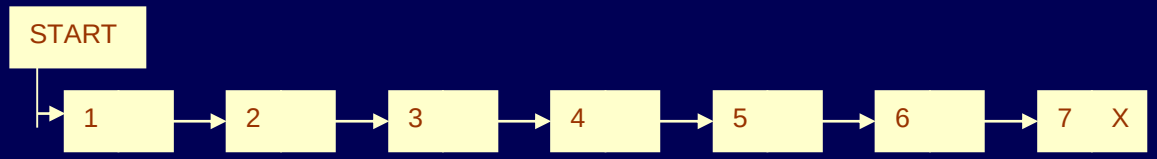
Unit – 3
Linear Data Structure

LINKED LISTS

INTRODUCTION

- A linked list in simple terms is a linear collection of data elements. These data elements are called nodes.
- Linked list is a data structure which in turn can be used to implement other data structures. Thus, it acts as building block to implement data structures like stacks, queues and their variations.
- A linked list can be perceived as a train or a sequence of nodes in which each node contain one or more data fields and a pointer to the next node.

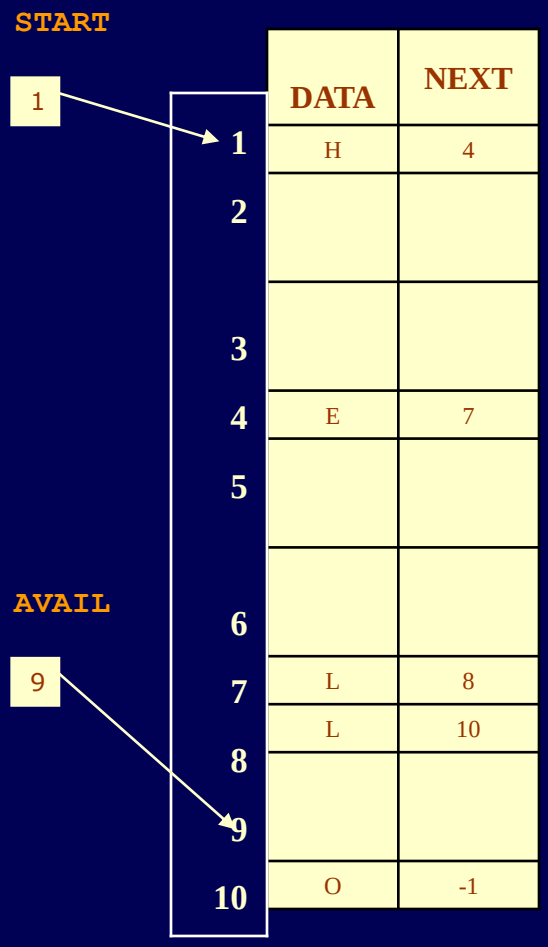


In the above linked list, every node contains two parts- one integer and the other a pointer to the next node. The left part of the node which contains data may include a simple data type, an array or a structure. The right part of the node contains a pointer to the next node (or address of the next node in sequence). The last node will have no next node connected to it, so it will store a special value called NULL.

INTRODUCTION contd.

- Linked list contains a pointer variable, START which stores the address of the first node in the list.
- We can traverse the entire list using a single pointer variable START. The START node will contain the address of the first node; the next part of the first node will in turn store the address of its succeeding node.
- Using this technique the individual nodes of the list will form a chain of nodes. If START = NULL, this means that the linked list is empty and contains no nodes.
- In C, we will implement a linked list using the following code:

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



START pointing to the first element of the linked list in memory

If we want to add a node to an already existing linked list in the memory, we will first find any free space in the memory and then use it to store the information.

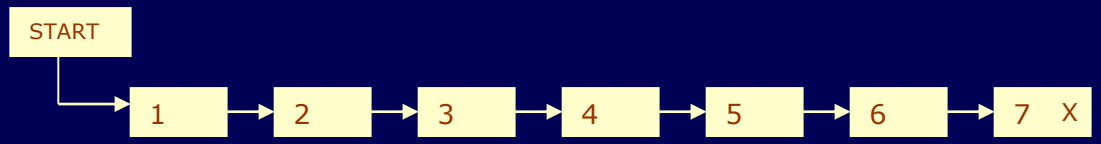
The operating system maintains a free pool which is a linked list of a of all free memory cells. It maintains a pointer variable AVAIL which stores the address of the first free space.

When you delete a node from the linked list, the operating system adds the freed memory to the free pool.

The operating system will perform this operation whenever it finds the CPU idle or whenever the programs are falling short of memory. The operating system scans through all the memory cells and mark the cells that are being used by some or the other program. Then, it collects all those cells which are not being used and add their address to the free pool so that it can be reused by the programs. This process is called garbage collection. The whole process of collecting unused memory cells (garbage collection) is transparent to the programmer.

Singly Linked List

- A singly linked list is the simplest type of linked list in which every node contains some data and a pointer to the next node of the same data type. By saying that the node contains a pointer to the next node we mean that the node stores the address of the next node in sequence.

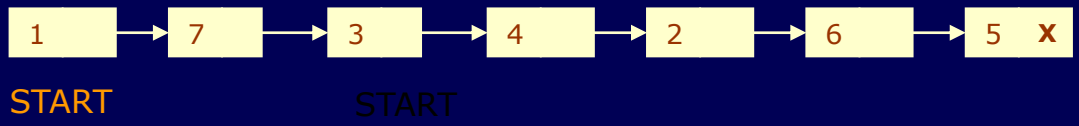


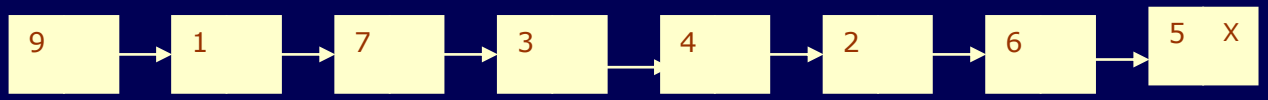

```
void printList() {
   struct node *ptr = start;
   printf("\n[ ");
   //start from the beginning
   while(ptr != NULL)
  {
      printf("%d ",ptr->value);
      ptr = ptr->next;
   }
}
```

```
Step 1: [INITIALIZE] SET PTR = START
Step 2: Repeat Steps 3 and 4 while PTR != NULL
Step 3:              Print PTR->DATA
Step 4:              SET PTR = PTR->NEXT
       [END OF LOOP]
Step 5: EXIT
```

Algorithm for traversing a linked list

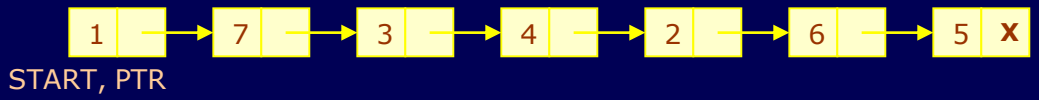
Inserting Node in the Linked List

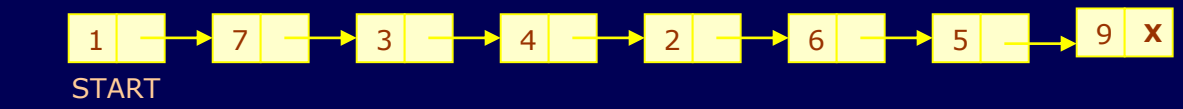





```
Step 1: IF AVAIL = NULL, then
              Write OVERFLOW
              Go to Step 7
        [END OF IF]
Step 2: SET New_Node = AVAIL
Step 3: SET New_Node->DATA = VAL
Step 4: SET New_Node->Next = START
Step 5: SET START = New_Node
Step 6: EXIT
```

Algorithm to insert a new node in the beginning of the linked list





PTR

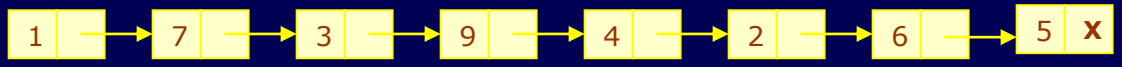
```
Step 1: IF AVAIL = NULL, then
              Write OVERFLOW
              Go to Step 10
        [END OF IF]
Step 2: SET New_Node = AVAIL
Step 3: SET New_Node->DATA = VAL
Step 4: SET New_Node->Next = NULL
Step 5: SET PTR = START
Step 6: Repeat Step 7 while PTR->NEXT != NULL
Step 7:                SET PTR = PTR ->NEXT
        [END OF LOOP]
Step 8: SET PTR->NEXT = New_Node
Step 9: EXIT
```

Algorithm to insert a new node at the end of the linked list

```
Step 1: IF AVAIL = NULL, then
                Write OVERFLOW
                Go to Step 12
        [END OF IF]
Step 2: SET New_Node = AVAIL
Step 3: SET New_Node->DATA = VAL
Step 4: SET PTR = START
Step 5: SET PREPTR = START
Step 6: Repeat Step 7 and 8 while PREPTR->DATA != NUM
Step 7:         SET PREPTR = PTR
Step 8:         SET PTR = PTR->NEXT
        [END OF LOOP]
Step 9: PREPTR->NEXT = New_Node
Step 10: SET New_Node->NEXT = PTR
Step 11: EXIT
```

Algorithm to insert a new node after a node (3) that has value NUM

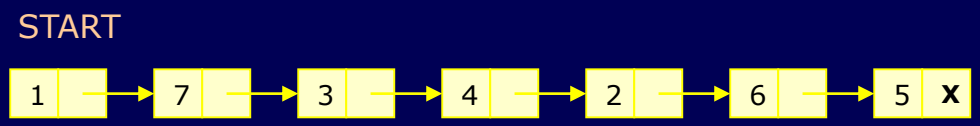
START, PTR, PREPTR

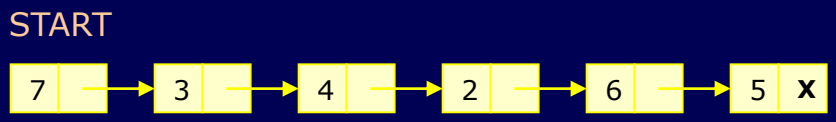


Deleting Node from the Linked List

```
Step 1: IF START = NULL, then
            Write UNDERFLOW
            Go to Step 5
        [END OF IF]
Step 2: SET PTR = START
Step 3: SET START = START->NEXT
Step 4: FREE PTR
Step 5: EXIT
```

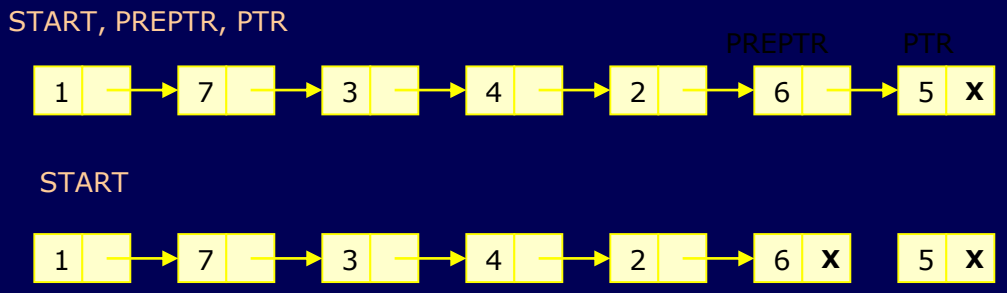
Algorithm to delete the first node from the linked list





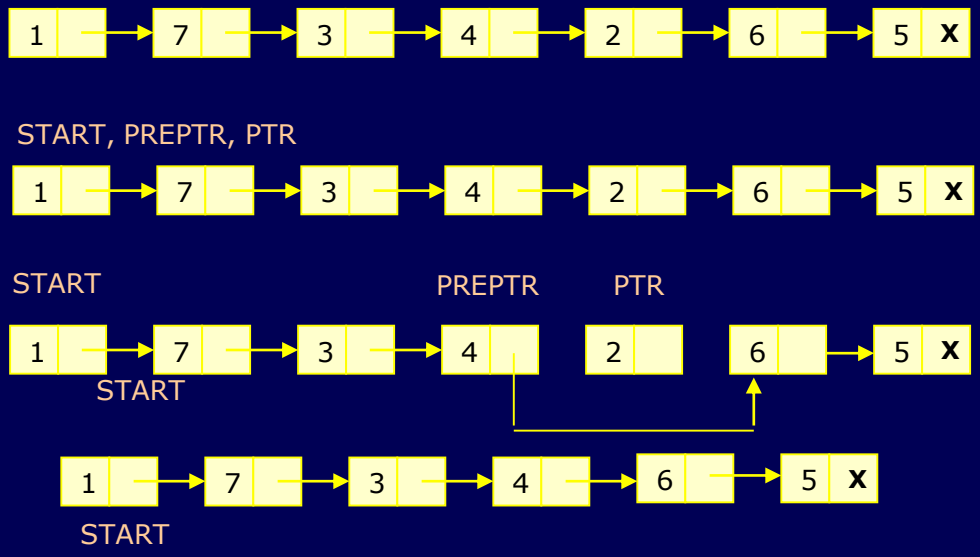
```
Step 1: IF START = NULL, then
                Write UNDERFLOW
                Go to Step 8
        [END OF IF]
Step 2: SET PTR = START
Step 3: Repeat Step 4 and 5 while PTR->NEXT != NULL
Step 4:                  SET PREPTR = PTR
Step 5:                  SET PTR = PTR->NEXT
        [END OF LOOP]
Step 6: SET PREPTR->NEXT = NULL
Step 7: FREE PTR
Step 8: EXIT
```

Algorithm to delete the last node of the linked list



```
Step 1: IF START = NULL, then
                Write UNDERFLOW
                Go to Step 10
        [END OF IF]
Step 2: SET PTR = START
Step 3: SET PREPTR = START
Step 4: Repeat Step 5 and 6 while PRETR->DATA != NUM
Step 5:                 SET PREPTR = PTR
Step 6:                 SET PTR = PTR->NEXT
        [END OF LOOP]
Step7: SET TEMP = PTR->NEXT
Step 8: SET PREPTR->NEXT = TEMP->NEXT
Step 9: FREE TEMP
Step 10: EXIT
```

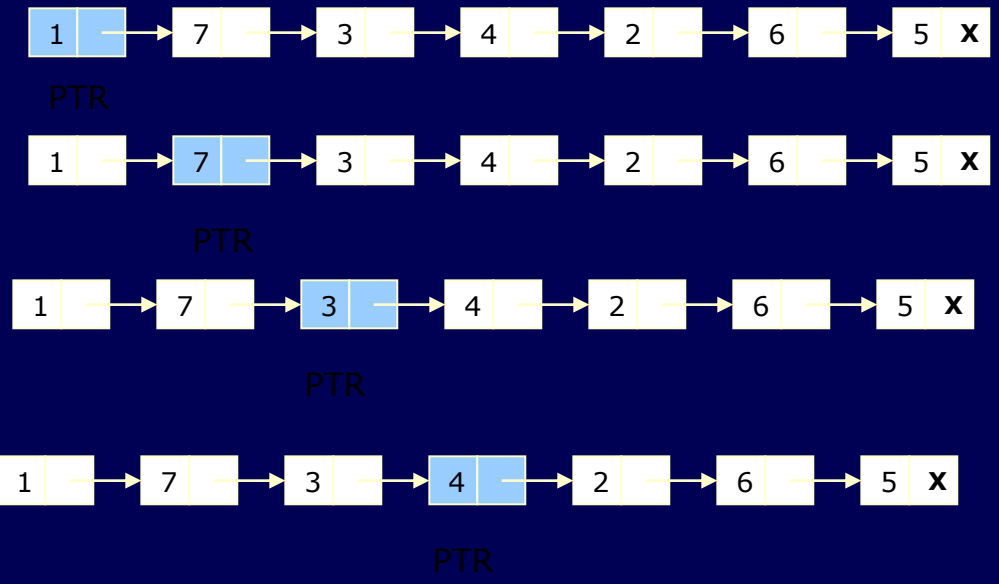
Algorithm to delete the node after a given node from the linked list



```
Algorithm to print the information stored in each node of the linked list

Step 1: [INITIALIZE] SET PTR = START
Step 2: Repeat Steps 3 and 4 while PTR != NULL
Step 3:              Write PTR->DATA
Step 4:              SET PTR = PTR->NEXT
          [END OF LOOP]
Step 5: EXIT
```

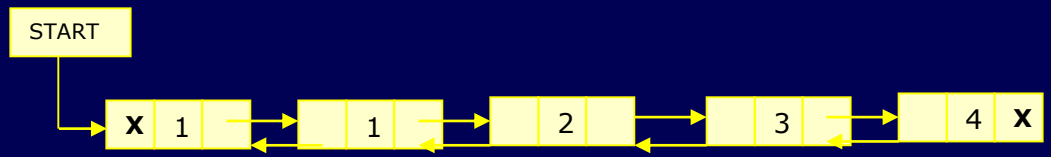
```
Algorithm to print the number of nodes in the linked list

Step 1: [INITIALIZE] SET Count = 0
Step 2: [INITIALIZE] SET PTR = START
Step 3: Repeat Steps 4 and 5 while PTR != NULL
Step 4:              SET Count = Count + 1
Step 5:              SET PTR = PTR->NEXT
          [END OF LOOP]
Step 6: EXIT
```

```
Algorithm to search an unsorted linked list

Step 1: [INITIALIZE] SET PTR = START
Step 2: Repeat Steps 3 while PTR != NULL
Step 3:              IF VAL = PTR->DATA
                                SET POS = PTR
                                Go To Step 5
                     ELSE
                                SET PTR = PTR->NEXT
                     [END OF IF]
          [END OF LOOP]
Step 4: SET POS = NULL
Step 5: EXIT
```



Doubly Linked List

A doubly linked list or a two way linked list is a more complex type of linked list which contains a pointer to the next as well as previous node in the sequence. Therefore, it consists of three parts and not just two. The three parts are data, a pointer to the next node and a pointer to the previous node



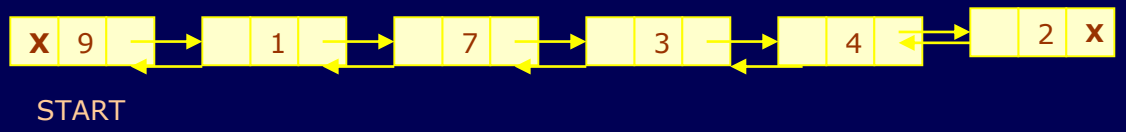
In C++ language, the structure of a doubly linked list is given as,

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

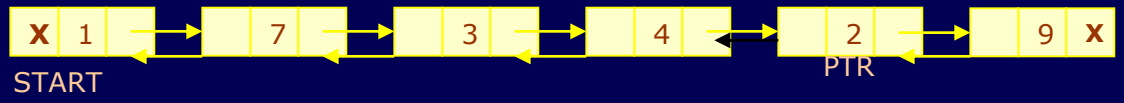
The prev field of the first node and the next field of the last node will contain NULL. The prev field is used to store the address of the preceding node. This would enable to traverse the list in the backward direction as well.

Algorithm to insert a new node in the beginning of the doubly linked list

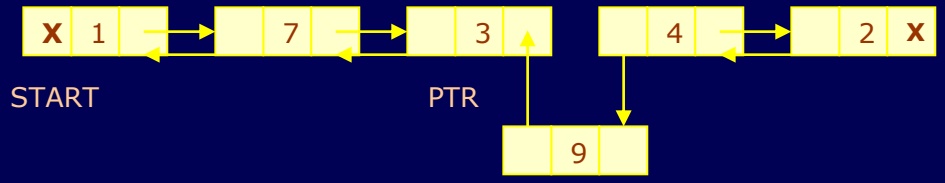
```
Step 1: IF Node_AVAIL = NULL, then
            Write OVERFLOW
            Go to Step 7
        [END OF IF] //Allocate memory if available
Step 2: SET New_Node = Node_AVAIL
Step 3: SET New_Node->DATA = VAL
Step 4: SET New_Node->PREV = NULL
Step 5: SET New_Node->Next = START
Step 6: SET START = New_Node
Step 7: EXIT
```

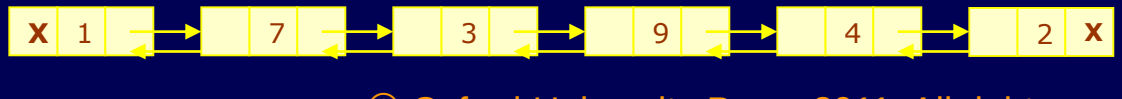


```
Algorithm to insert a new node at the end of the doubly linked list

Step 1: IF AVAIL = NULL, then
                    Write OVERFLOW
                    Go to Step 11
          [END OF IF]
Step 2: SET New_Node = AVAIL
Step 3: SET AVAIL = AVAIL->NEXT
Step 4: SET New_Node->DATA = VAL
Step 5: SET New_Node->Next = NULL
Step 6: SET PTR = START
Step 7: Repeat Step 8 while PTR->NEXT != NULL
Step 8:             SET PTR = PTR->NEXT
          [END OF LOOP]
Step 9: SET PTR->NEXT = New_Node
Step 10: New_Node->PREV = PTR
Step 11: EXIT
```



```
Algorithm to insert a new node after a node that has value NUM

Step 1: IF AVAIL = NULL, then
                    Write OVERFLOW
                    Go to Step 11
          [END OF IF]
Step 2: SET New_Node = AVAIL
Step 3: SET AVAIL = AVAIL->NEXT
Step 4: SET New_Node->DATA = VAL
Step 5: SET PTR = START
Step 6: Repeat Step 8 while PTR->DATA != NUM
Step 7:              SET PTR = PTR->NEXT
          [END OF LOOP]
Step 8: New_Node->NEXT = PTR->NEXT
Step 9: SET New_Node->PREV = PTR
Step 10: SET PTR->NEXT = New_Node
Step 11: EXIT
```

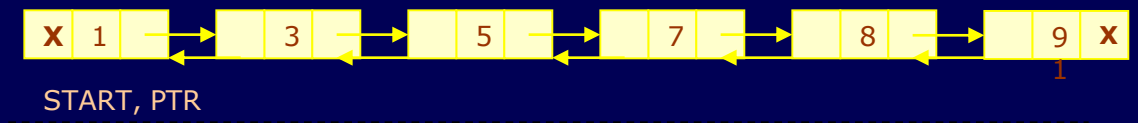


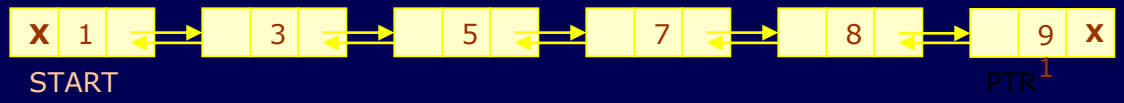


```
Algorithm to delete the first node from the doubly linked list

Step 1: IF START = NULL, then
                    Write UNDERFLOW
                    Go to Step 6
          [END OF IF]
Step 2: SET PTR = START
Step 3: SET START = START->NEXT
Step 4: SET START->PREV = NULL
Step 5: FREE PTR
Step 6: EXIT
```

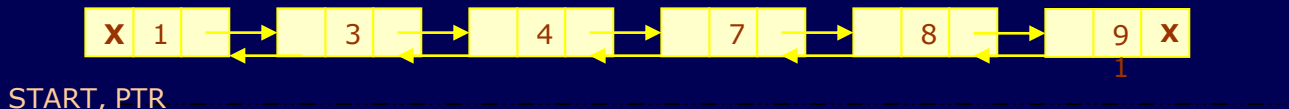
START, PTR

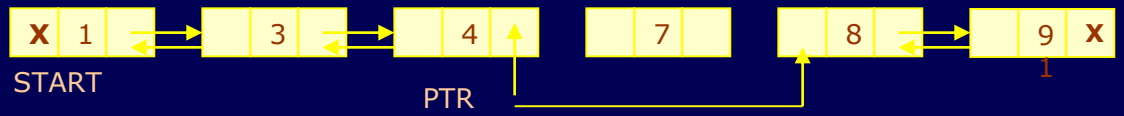
```
Algorithm to delete the last node of the doubly linked list

Step 1: IF START = NULL, then
                    Write UNDERFLOW
                    Go to Step 7
          [END OF IF]
Step 2: SET PTR = START
Step 3: Repeat Step 4 and 5 while PTR->NEXT != NULL
Step 4:             SET PTR = PTR->NEXT
          [END OF LOOP]
Step 5: SET PTR->PREV->NEXT = NULL
Step 6: FREE PTR
Step 7: EXIT
```





```
Algorithm to delete the node after a given node from the doubly linked list

Step 1: IF START = NULL, then
                    Write UNDERFLOW
                    Go to Step 9
          [END OF IF]
Step 2: SET PTR = START
Step 3: Repeat Step 4 while PTR->DATA != NUM
Step 4:             SET PTR = PTR->NEXT
          [END OF LOOP]
Step 5: SET TEMP = PTR->NEXT
Step 6: SET PTR->NEXT = TEMP->NEXT
Step 7: SET TEMP->NEXT->PREV = PTR
Step 8: FREE TEMP
Step 9: EXIT
```



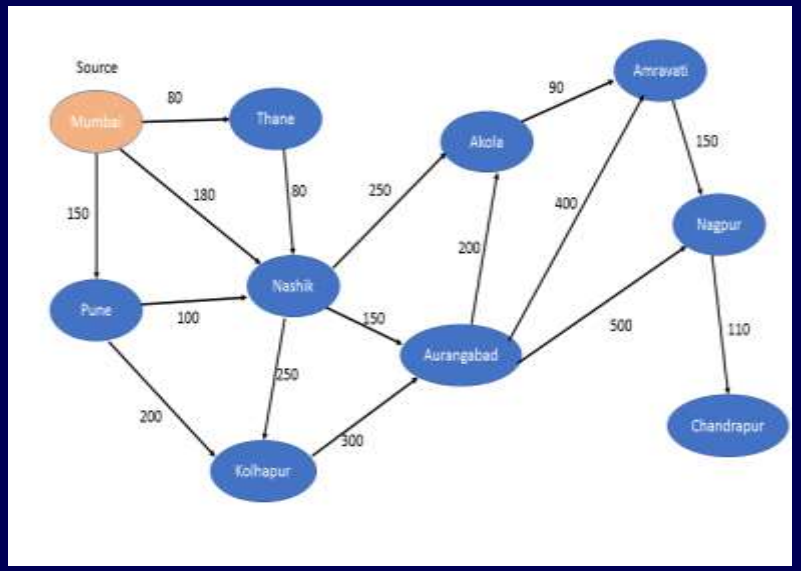


Unit - 4

TREES

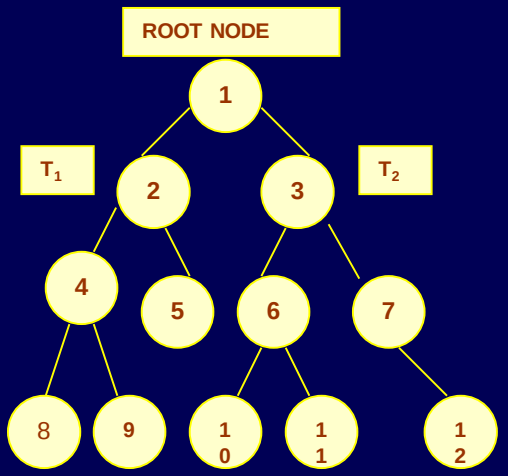
(Non-linear data structure)

General tree



BINARY TREES

- **A binary tree is a data structure** which **is defined as a collection of elements called nodes.**
- **Every node contains a "left" pointer, a "right" pointer, and a data element.**
- **Every binary tree has a root element pointed by a "root" pointer.**
- **The root element is the topmost node in the tree. If root = NULL, then it means the tree is empty.**
- **If the root node *R* is not NULL, then the two trees *T1* and *T2* are called the left and right subtrees of *R*.**
- **if T*1* is non-empty, then *T1* is said to be the left successor of *R*.**
- **likewise, if T2 is non-empty then, it is called the right successor of *R*.**



Note: In a binary tree every node has 0, 1 or at the most 2 successors. A node that has no successors or 0 successors is called the leaf node or the terminal node.

Binary tree terminologies

- ***Sibling:*** If N is any node in T that has *left successor* S1 and *right successor* S2, then N is called the *parent* of S1 and S2.
  1. Correspondingly, S1 and S2 are called the left child and the right child of N. Also, S1 and S2 are said to be *siblings*.
  2. Every node other than the root node has a parent.
  3. In other words, all nodes that are at the same level and share the same parent are called *siblings* (brothers).
- ***Level number:*** Every node in the binary tree is assigned a *level number*.
  1. The root node is defined to be at level 0.
  2. The left and right child of the root node has a level number 1.
  3. Similarly, every node is at one level higher than its parents.
  4. So all child nodes are defined to have level number as parent's level number + 1.
- ***Degree:*** Degree of a node is equal to the number of children that a node has. The degree of a leaf node is zero.
  1. *In-degree* of a node is the number of edges arriving at that node. The root node is the only node that has an in-degree equal to zero. Similarly,
  2. *Out-degree* of a node is the number of edges leaving that node.
- ***Leaf node:*** A leaf node has no children.

Binary tree terminologies

- *Directed edge:* Line drawn from a node N to any of its successor is called a *directed edge*. A binary tree of n nodes have exactly n – 1 edges
- *Path:* A sequence of consecutive edges is called a *path.*
- *Depth:* The *depth* of a node N is given as the length of the path from the root R to the node N. The depth of the root node is zero.

  The *height/depth* of a tree is defined as the length of the path from the root node to the deepest node in the tree.

  Ancestor and descendant nodes: Ancestors of a node are all the nodes along the path from the root to that node.

  Similarly, descendants of a node are all the nodes along the path from that node to the leaf node.

- Binary trees are commonly used to implement binary search trees, expression trees, tournament trees and binary heaps.

A binary tree of height h, has at least h+1 nodes and at most 2^{h+1} -1 nodes.

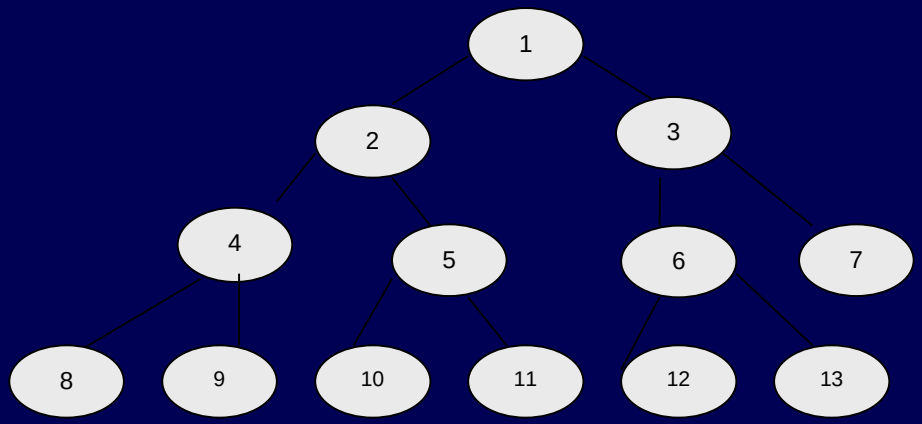
Complete Binary Trees

- A *complete binary tree* is a binary tree which satisfies two properties.
  1. First, in a complete binary tree every level, except possibly the last, is completely filled.
  2. Second, all nodes appear as far left as possible
- In a complete binary tree Tn, there are exactly n nodes and level r of T can have at most 2^{r+1} - 1 nodes.

- The formula to find the parent, left child and right child can be given as-
- if K is a parent node, then its left child can be calculated as 2 * K and its right child can be calculated as 2 * K + 1.

  For example, the children of node 4 are 8 (2*4) and 9 (2* 4 + 1).

  Similarly, the parent of the node K can be calculated as | K/2 |. Given the node 4, its parent can be calculated as | 4/2 | = 2.



The height of a tree Tn having exactly n nodes is given as,

$Hn = | \log_2 n + 1 |$

This means, if a tree T has 10,00,000 nodes then its height is 21.

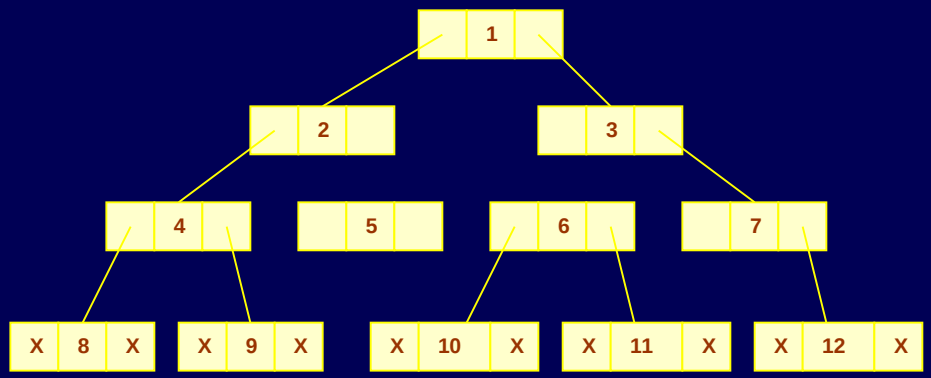
Representation of Binary Trees in Memory

- In computer's memory, a binary tree can be maintained either using a linked representation (as in case of a linked list) or using sequential representation (as in case of single arrays).

Linked Representation of Binary Trees

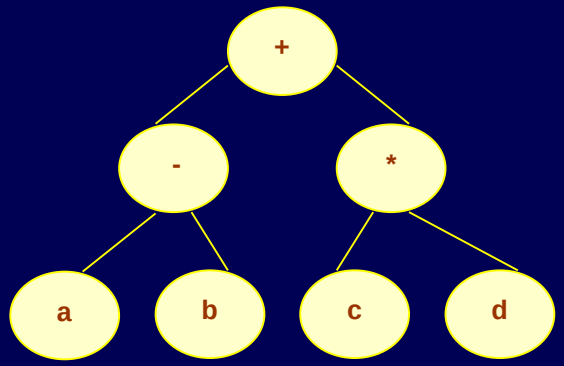
- In linked representation of binary tree, every node will have three parts, the data element, a pointer to the left node and a pointer to the right node. So in C, the binary tree is built with a node type given as below.

```
struct node {
    struct node* left;
    int data;
    struct node* right;
};
```



Binary tree as EXPRESSION TREES

- Binary trees are widely used to store algebraic expressions. For example, consider the algebraic expression Exp given as,
- Exp = (a – b) + (c * d)
- This expression can be represented using a binary tree as shown in figure



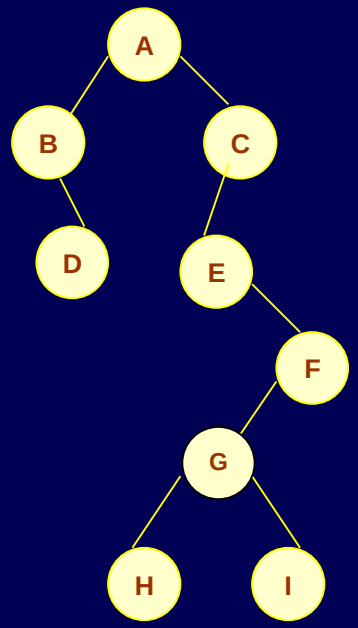
TRAVERSING OF A BINARY TREE

- Traversing a binary tree is the process of visiting each node in the tree exactly once, in a systematic way.
- Unlike linear data structures in which the elements are traversed sequentially, tree is a non-linear data structure in which the elements can be traversed in many different ways. There are different algorithms for tree traversals.
- These algorithms differ in the order in which the nodes are visited. In this section, we will read about these algorithms.
- **Pre-order algorithm**

  To traverse a non-empty binary tree in preorder, the following operations are performed recursively at each node. The algorithm starts with the root node of the tree and continues by,

  Repeat following steps using recursion:

  1. Visiting the root node. – **Print Node**
  2. Traversing the left subtree.
  3. Traversing the right subtree.



print sequence
A, B, D, C, E, F, G, H, I

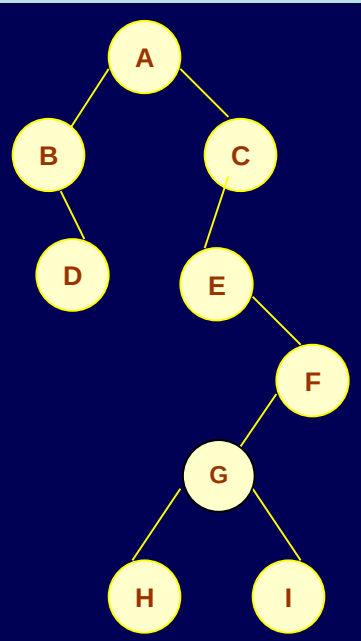
In-order algorithm

To traverse a non-empty binary tree in **in-order**, the following operations are performed recursively at each node. The algorithm starts with the root node of the tree and continues by,

Repeat following steps using recursion:

1. **Traversing the left subtree.**
2. **Visiting the root node. - Print Node**
3. **Traversing the right subtree.**

print sequence
B, D, A, E, H, G, I, F, C.



Post-order algorithm

To traverse a non-empty binary tree in **post-order**,
the following operations are performed recursively at each node.
The algorithm starts with the root node of the tree and continues

Repeat following steps using recursion:

1. **Traversing the left subtree.**
2. **Traversing the right subtree.**
3. **Visiting the root node. - Print Node**

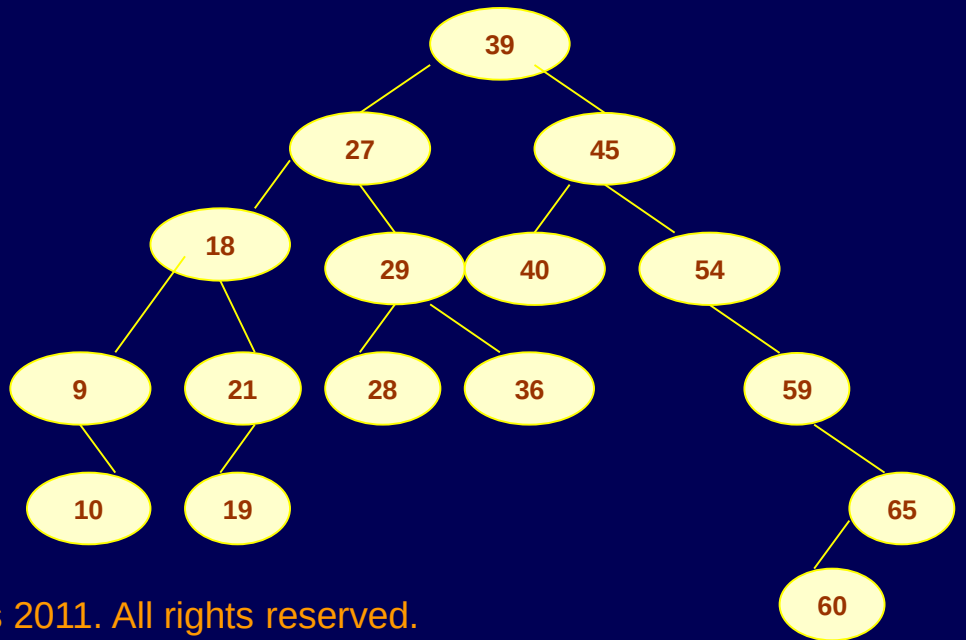
print sequence
D, B, H, I, G, F, E, C, A.

BINARY SEARCH TREES

BINARY SEARCH TREES

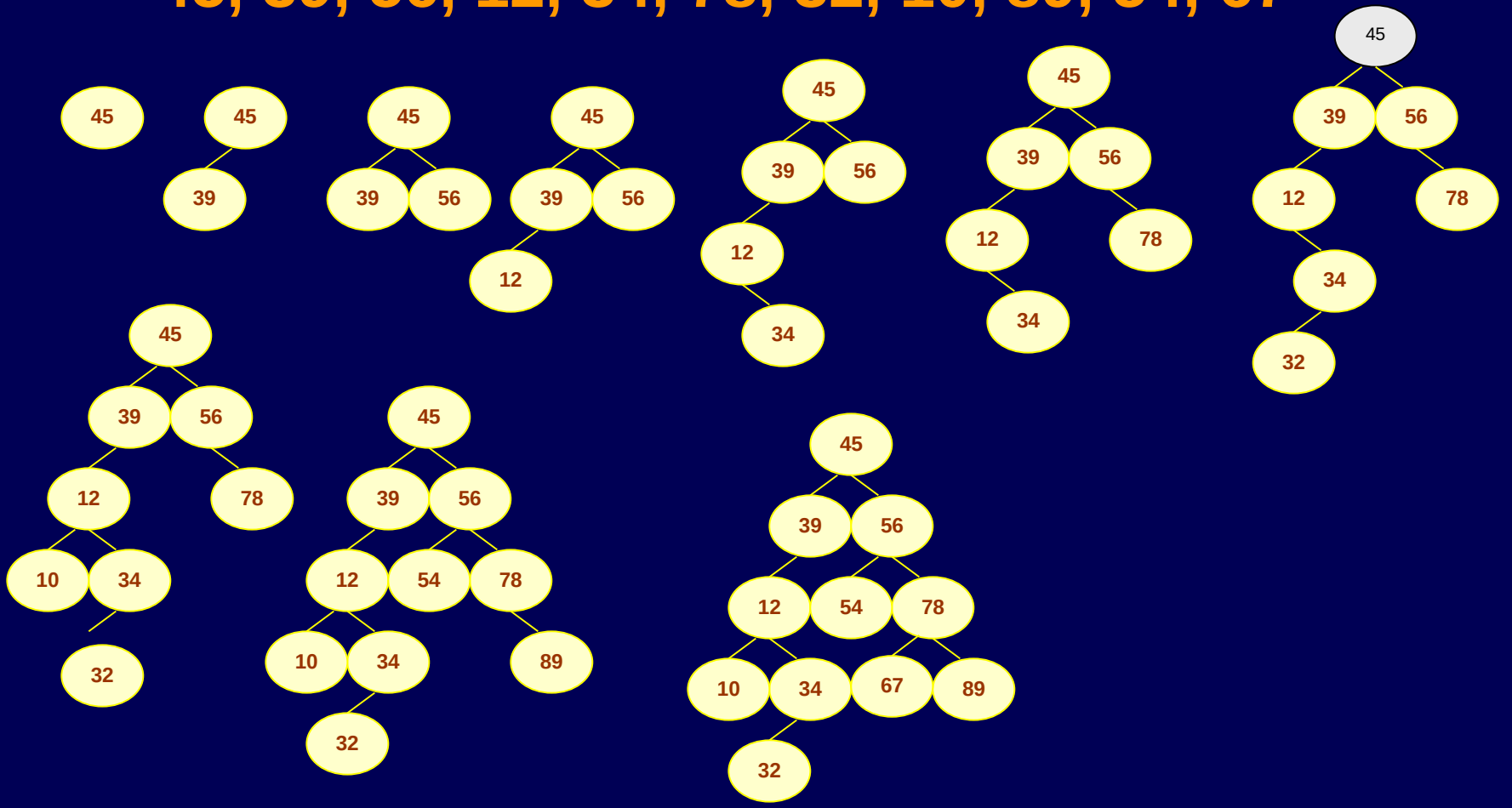
- A binary search tree, also known as an ordered binary tree is a variant of binary tree in which the nodes are arranged in order.
- In a binary search tree, all the nodes in the left sub-tree have a value less than that of the root node.
- Correspondingly, all the nodes in the right sub-tree have a value either equal to or greater than the root node.
- The same rule is applicable to every sub-tree in the tree.
- The running time of a search operation is $O(log_2 n)$, this is the worst case complexity of BST. We eliminate half of the sub-tree from the search process.

Due to its efficiency in searching elements, binary search trees are widely used in problems related to search space, where elements that are indexed by some key value.



Create a binary search trees using the following data values

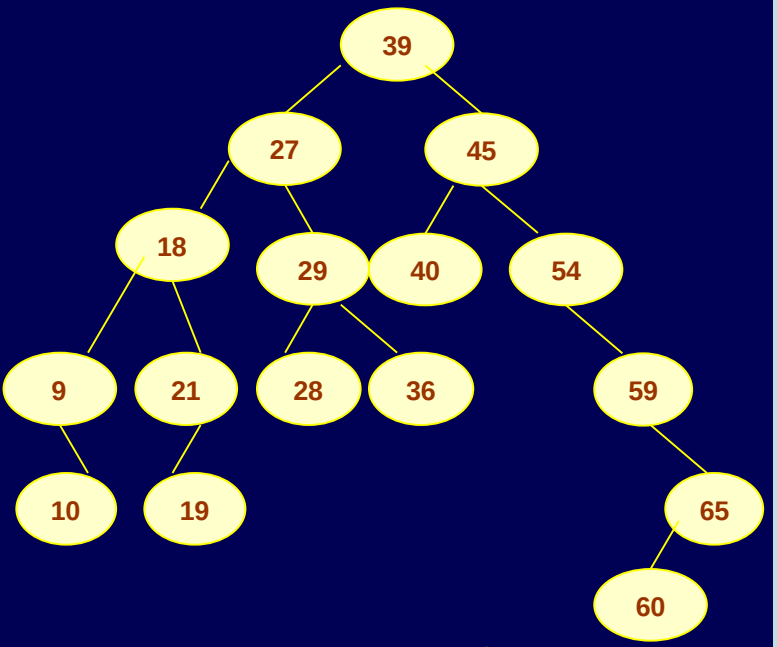
45, 39, 56, 12, 34, 78, 32, 10, 89, 54, 67



Insertion in a BST

```
Algorithm to insert a value in the binary
search tree-

Step 1: IF TREE = NULL, then
          Allocate memory for TREE
          SET TREE->DATA = VAL
         SET TREE->LEFT = TREE ->RIGHT = NULL
      ELSE
         IF VAL < TREE->DATA
             Insert(TREE->LEFT, VAL)
         ELSE
          Insert(TREE->RIGHT, VAL)
         [END OF IF]
       [END OF IF]
Step 2: End
```

```
struct node* new_node(int
x)
{
    node *p = new node();
    p->data = x;
    p->left_child = NULL;
    p->right_child = NULL;
    return p;
}
```




```
struct node * insertNode(node *root1, int x)
{
   //searching for the place to insert
if(root1==NULL)
       root1 = new_node(x);     //return new_node(x);
else
   if(x < root1->data) // x is smaller, will be inserted to left
       root1->left_child = insertNode(root1->left_child, x);

else // x is greater, should be inserted to right
       root1->right_child = insertNode(root1->right_child, x);

 //return root;
}
```

Insertion in a BST

```
struct node* new_node(int x)
{
    node *p = new node();
    p->data = x;
    p->left_child = NULL;
    p->right_child = NULL;
    return p;
}
```

```
struct node * insertNode(node *root1, int x)
{
   //searching for the place to insert
if(root1==NULL)
        root1 = new_node(x);     //return new_node(x);
else
   if(x < root1->data) // x is smaller, will be inserted to left
        root1->left_child = insertNode(root1->left_child, x);

else // x is greater, should be inserted to right
        root1->right_child = insertNode(root1->right_child, x);

 //return root;
}
```

Searching a Value in BST

- The search function is used to find whether a given value is present in the tree or not. The searching process begins at the root node.
- The function first checks if the binary search tree is empty. If it is, then it means that the value we are searching for is not present in the tree.
- So, the search algorithm terminates by displaying an appropriate message.
- However, if there are nodes in the tree then the search function checks to see if the key value of the current node is equal to the value to be searched. If not, it checks if the value to be searched for is less than the value of the node, in which case it should be recursively called on the left child node. In case the value is greater than the value of the node, it should be recursively called on the right child node.

```
Algorithm to search in a BST

Step 1:IF TREE->DATA = VAL OR TREE = NULL, then
          Return TREE
        ELSE
      IF VAL < TREE->DATA
        Return searchElelement(TREE->LEFT, VAL)
        ELSE
        Return searchElelement(TREE->RIGHT, VAL)
      [END OF IF]
    [END OF IF]
Step 2: End
```

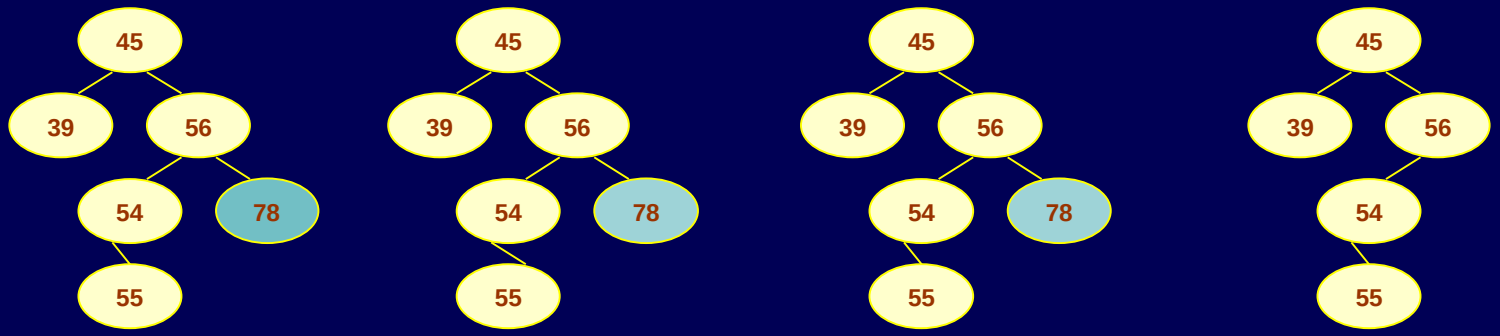
```
int searchNode(node *root, int value)
{
        if(root==NULL)
            return 0;
        else if(root->data == value)
            return 1;
        else if (value < root->data)
          searchNode (root->left_child, value);
        else if (value > root->data)
        searchNode (root->right_child, value);
}
```

Deletion from a BST

- **The delete function deletes a node from the binary search tree. However, utmost care should be taken that the properties of the binary search tree does not get violated and nodes are not lost in the process.**

The deletion of a node can be done in any of the three cases.

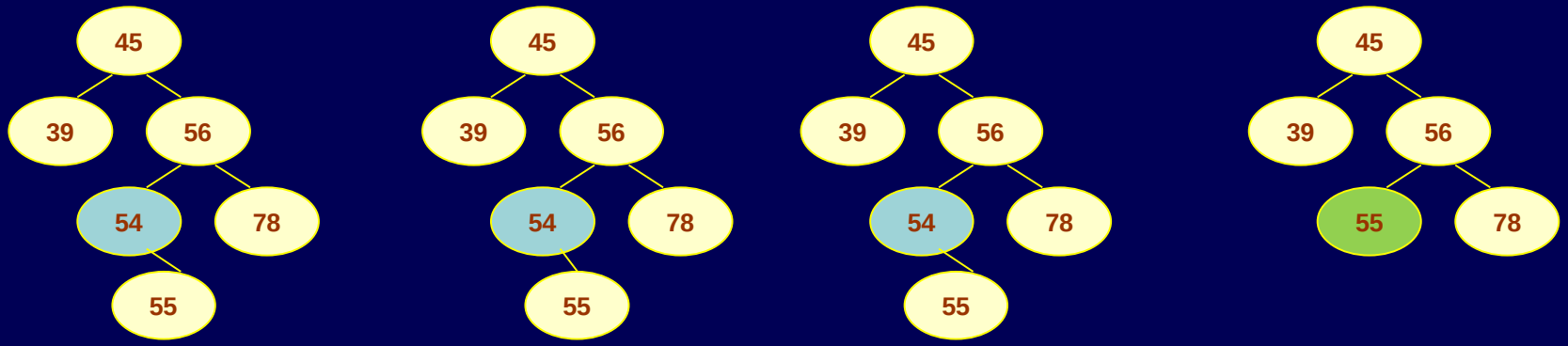
- ***Case 1:*** **Deleting a node that has no children.**



Partial code for Case -1

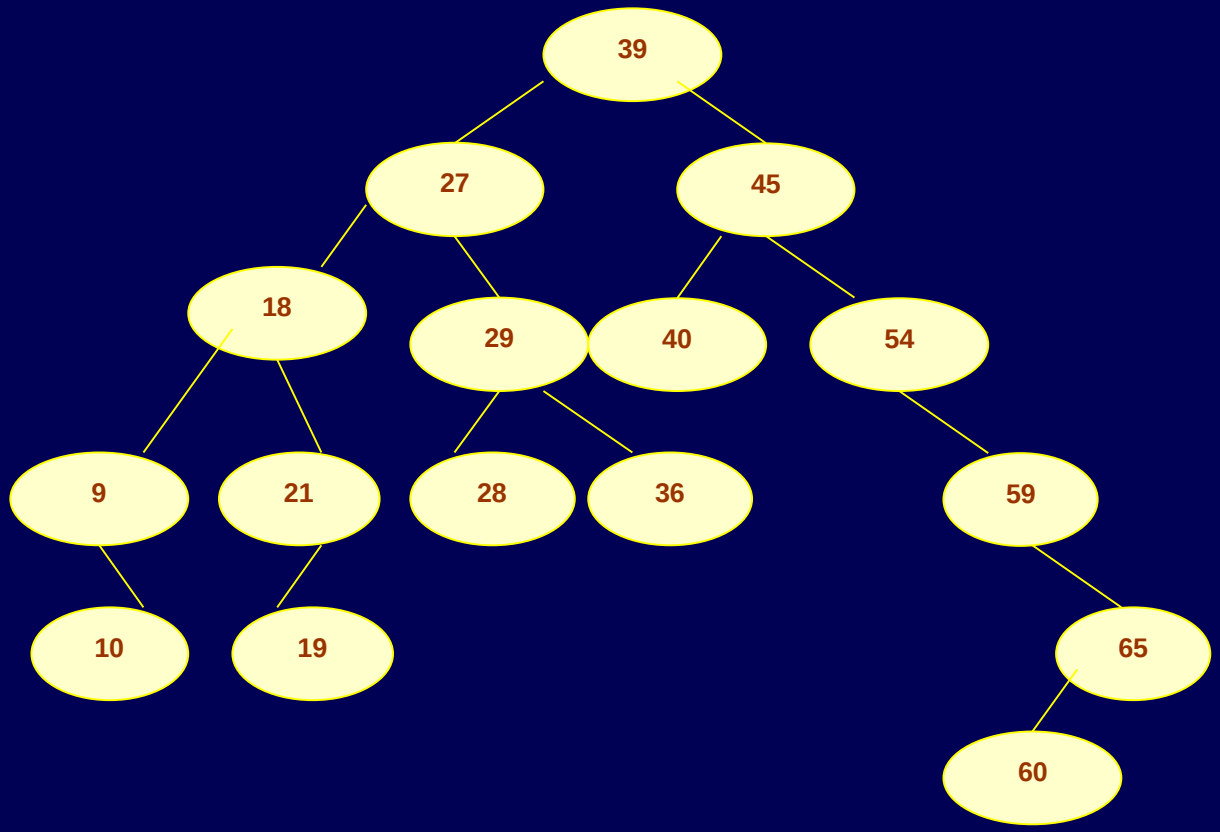
```
node* deleteNode (node* root1, int value)
{

        if(root1==NULL)
                return root1;
        else if(value < root1->data)
        {
                root1->left_child  = deleteNode (root1->left_child, value);
        }
        else if(value > root1->data)
        {
                root1->right_child = deleteNode(root1->right_child,value);
        }
        // Node deletion   (Case – 1 : Leaf Node)
        else
        {
                if(root1->left_child==NULL && root1->right_child==NULL)
                {
                        free(root1);
                        root1=NULL;
                        return root1;
                }
        }
}
```

- *Case 2:* Deleting a node with one child (either left or right).
- To handle the deletion, the node's child is set to be the child of the node's parent.
- In other words, replace the node with its child.
- Now, if the node was the left child of its parent, the node's child becomes the left child of the node's parent.
- Correspondingly, if the node was the right child of its parent, the node's child becomes the right child of the node's parent.



Partial code for Case -2

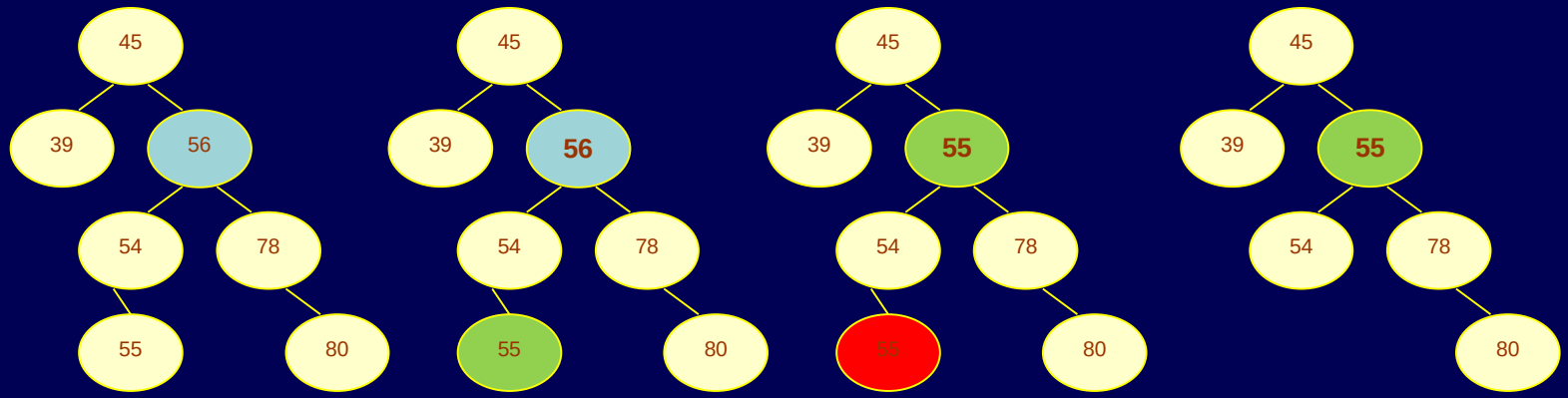
```
node* deleteNode(node* root1, int value)
{
        if(root1==NULL)   return root1;

        else if(value < root1 -> data)
        {    root1 -> left_child = deleteNode(root1 -> left_child,value);   }

        else if(value > root1 -> data)
        {    root1 -> right_child = deleteNode(root1 -> right_child,value);  }

        // Node deletion   (Case – 2 : either with left or right child)
        else if (root1->left_child == NULL)
                {
                        node* temp = root1;
                        root1=root1->right_child;
                        free(temp);
                        return root1;
                }
        else if (root1->right_child == NULL)
                {
                        node* temp=root1;
                        root1=root1->left_child;
                        free(temp);
                        return root1;
                }
}
```



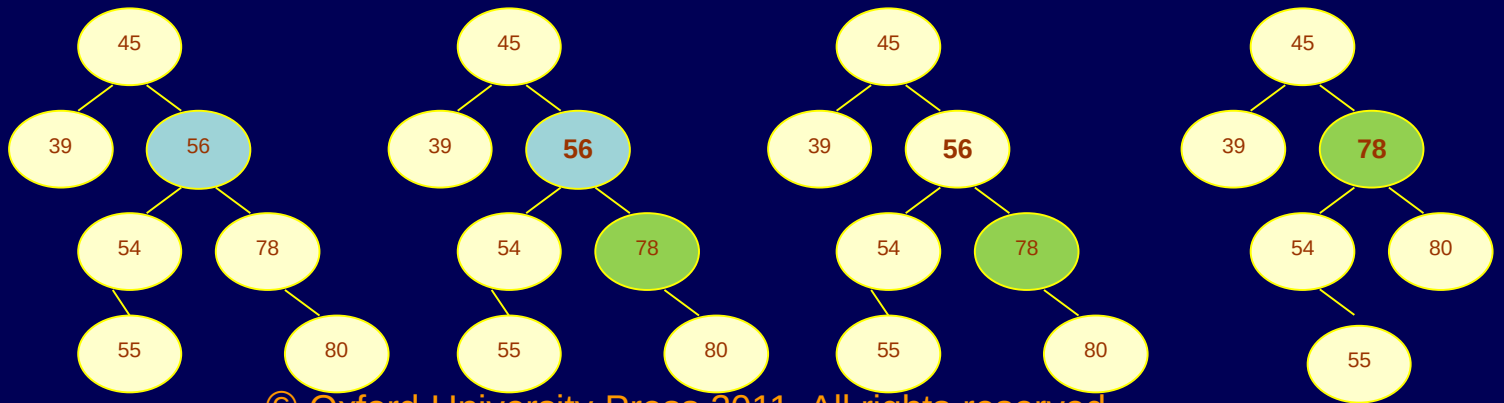
Case 3: Deleting a node with two children. To handle this case of deletion.

1. replace the node's value with its in-order predecessor (right most child of the left sub-tree) or
2. in-order successor (leftmost child of the right sub-tree).
3. The in-order predecessor or the successor can then be deleted using any of the above cases.

Finding Maximum of the left child – right most child



Finding Minimum of the right child – left most child



Partial code for Case -3

```
node* findMin(node*root1)
{
    while(root1->left_child!=NULL)
    {
        root1=root1->left_child;
    }
    return root1;
}
```

```
node* deleteNode(node* root1,int value)
{
        if(root1==NULL)
                return root1;
        else if(value< root1->data)
        {
                root1->left_child = deleteNode(root1->left_child, value);
        }
        else if(value> root1->data)
        {
                root->right_child= deleteNode(root1->right_child, value);
        }

        // Node deletion   (Case – 2 : Node has two children)

        else
                {
                        node*temp = findMin(root1->right_child);
                        root1->data=temp->data;
                        root1->right_child=deleteNode(root1->right_child, temp->data);
                }
```

Algorithm with all three cases

```
Algorithm to delete a value in the binary search tree

Step 1: IF TREE = NULL, then
          Write "VAL not found in the tree"

          ELSE IF VAL < TREE->DATA
             Delete(TREE->LEFT, VAL)

          ELSE IF VAL > TREE->DATA
             Delete(TREE->RIGHT, VAL)

         ELSE IF TREE->LEFT AND TREE->RIGHT
             SET TEMP = findLargestNode(TREE->LEFT)
             SET TREE->DATA = TEMP->DATA
             Delete(TREE->LEFT, TEMP->DATA)
          ELSE
            SET TEMP = TREE
            IF TREE->LEFT = NULL AND TREE->RIGHT = NULL
                   SET TREE = NULL
            ELSE IF TREE->LEFT != NULL
                   SET TREE = TREE->LEFT
                    ELSE
                   SET TREE = TREE->RIGHT
            FREE TEMP
Step 2: End
```

Expression tree

- **Expression: 9 - ((3 * 4) + 8) / 4**
- **Postfixed: 9 3 4 * 8 + 4 / -**
- **Prefixed: - / 4 + 8 * 4 3 9**

Huffman Encoding

Message: BCCABBDDAAACCBBAEEEEAAABB

Char	Count / Frequency	code
A	8	
B	7	
C	4	
D	2	
E	4	
40 bits	25 count	

Message: BCCABBDDAAACCBBAEEEEAAABB

Huffman coding

(i) Data can be encoded efficiently using Huffman Codes.
(ii) It is a widely used and beneficial technique for compressing data.
(iii) Huffman's greedy algorithm uses a table of the frequencies of occurrences of each character to build up an optimal way of represent
Suppose we have 10^5 characters in a data file. Normal Storage: 8 bits per character (ASCII) - 8 x 10^5 bits in a file. But we want to compress the file and save it compactly.
How can we represent the data in a Compact way?

(i) Fixed length Code: Each letter is represented by an equal number of bits. With a fixed-length code, at least 3 bits per character:

	a	b	c	d	e	f	Total
Frequency	45	13	12	16	9	5	100

100*3 = 300 bits required for saving this word

…

a=000

b=001

c=010

d=011

e=100

f=101

For a file with 10^5 characters, we need 3 x 10^5 bits.

...

(ii) A variable-length code: It can do considerably better than a fixed-length code, by giving many characters short code words and infrequent character long codewords.

a 0
b 101
c 100
d 111
e 1101
f 1100
Number of bits = (45 x 1 + 13 x 3 + 12 x 3 + 16 x 3 + 9 x 4 + 5 x 4) **= 224 bits**

Prefix Codes:

The prefixes of an encoding of one character must not be equal to complete encoding of another character, e.g., 1100 and 11001 are not valid codes because 1100 is a prefix of some other code word is called prefix codes.

Prefix codes are desirable because they clarify encoding and decoding. Encoding is always simple for any binary character code; we concatenate the code words describing each character of the file. Decoding is also quite comfortable with a prefix code. Since no codeword is a prefix of any other, the codeword that starts with an encoded data is unambiguous.

HUFFMAN CODING

```
Algorithm HUFFMAN_CODE (PQ)
// PQ is the priority queue, in which priority is frequency of each character.
// PQ contains all n characters in decreasing order of their priority
for i ← 1 to n – 1 do
    z ← CreateNode()
    x ← LeftChild[z] + deque(PQ)
    y ← RightChild[z] + deque(PQ)
    z.priority ← x.priority + y.priority
    enqueue (PQ, z)
end
return deque(Q)
```

Example

Example: Find Huffman code for each symbol in following text :
ABCCDEBABFFBACBEBDFAAAABCDEEDCCBFEBFCAE

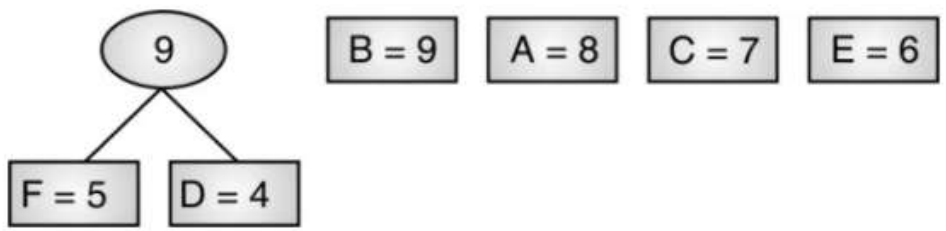
Solution:

For given text, occurrence of each character is as follow : S = <A, B, C, D, E, F>, P = <8, 9, 7, 4, 6, 5>

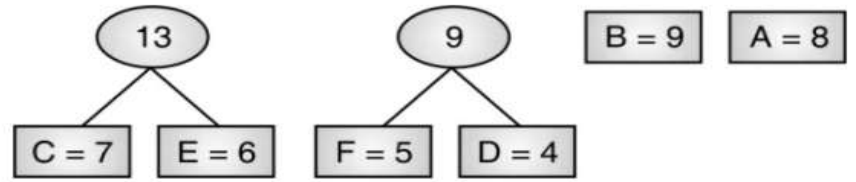
Step 1 : Arrange all characters in decreasing order of their frequency.

S = <A, B, C, D, E, F>, P= <8, 9, 7, 4, 6, 5>

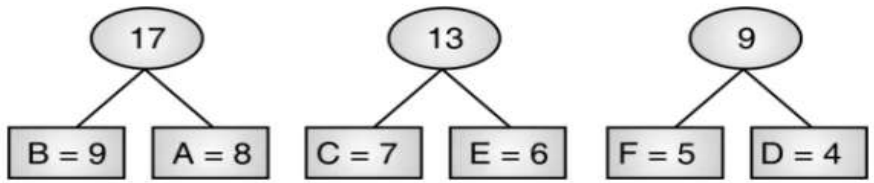
Step 2 : Merge last two nodes and arrange it again in order



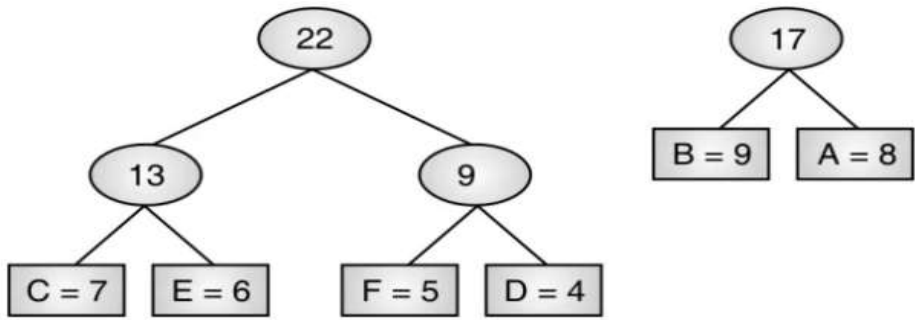
Step 3 : Merge last two nodes and arrange it again in order



Step 4 : Merge last two nodes and arrange it again in order



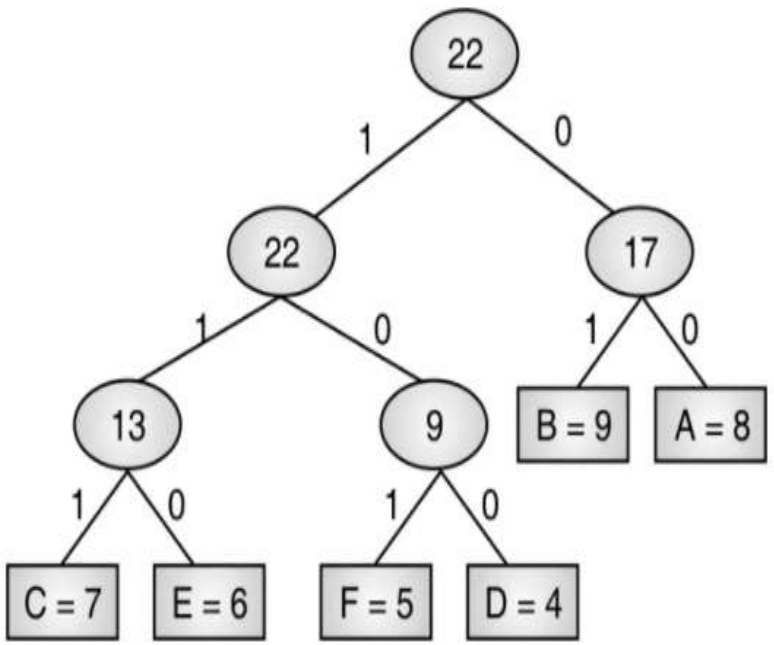
Step 5 : Merge last two nodes and arrange it again in order



...

Step 6 : Merge last two nodes and arrange it again in order.

Label all left arc by 1 and all right arc by 0



Visit all leaf nodes and read its edge value to find its prefix code.

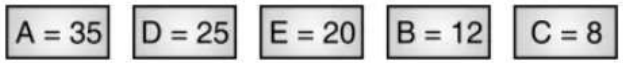
Character	Prefix Code
A	00
B	01
C	111
D	100
E	110
F	101

Example

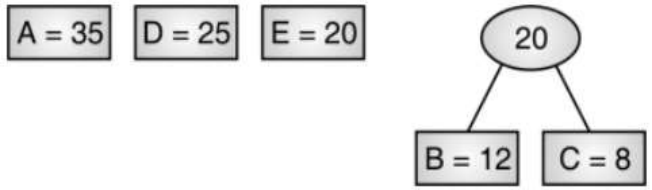
Example: Given that for character set S = <A, B, C, D, E> occurrence in text file is P = <35, 12, 8, 25, 20>. Find prefix code for each symbol.

Solution:

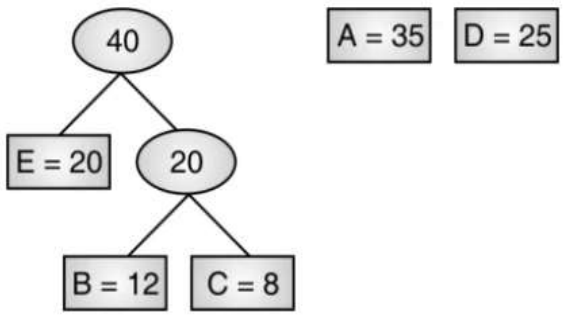
Step 1 : Arrange all characters in decreasing order of their frequency. S = <A, D, E, B, C> and corresponding P = <35, 25, 20, 12, 8>



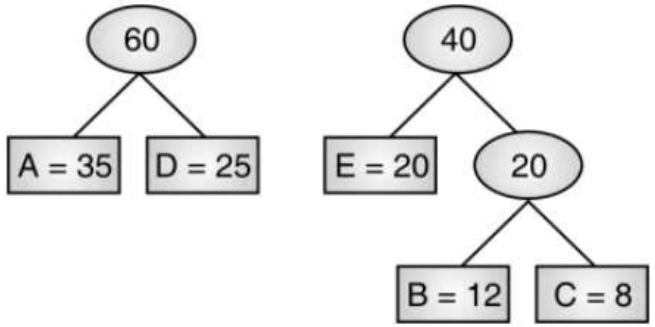
Step 2 : Merge last two nodes and arrange it again in order



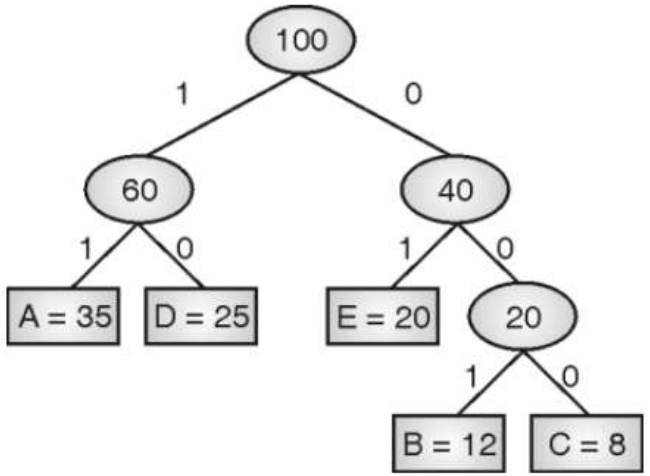
Step 3 : Merge last two nodes and arrange it again in order,



Step 4 : Merge last two nodes and arrange it again in order,



Step 5 : Merge last two nodes and arrange it again in order. Label all left arc by 1 and all right arc by 0



Visit all leaf nodes and read its edge value to find its prefix code.

Character	Prefix Code
A	11
B	001
C	000
D	10
E	01