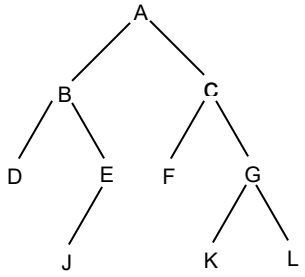
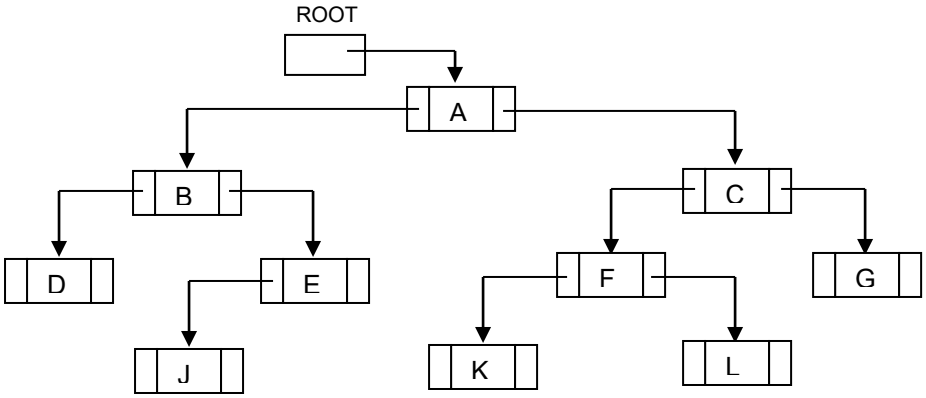
HEADER NODE

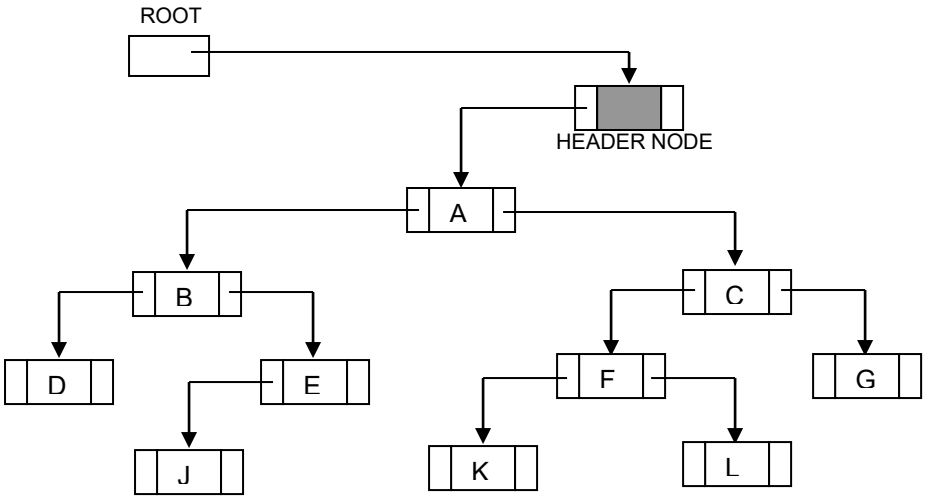
Suppose a binary tree **T** is maintained in memory by means of a linked representation. Sometimes an extra, special node, called a **HEADER NODE**, is added to the beginning of T. When this extra node is used, the tree pointer variable, which we will call **HEAD** (instead ROOT), will point to the header node, and the left pointer of the HEADER node will point to the root of T. The following schematic diagram elastic the mater.



Schematic representation of a binary search tree



Linked representation of a binary search tree without *header* node



Linked representation of a binary search tree with *header* node

Suppose a binary tree **T** is empty. Then **T** will still contains a header node, but the left pointer of the header node will contain *NULL* value. Thus the condition, *LEFT[HEAD]=NULL* will indicate an empty tree.

Another version of the above representation of a binary tree **T** is to use the header node as a sentinel, i.e., if a node has empty subtree, then the pointer field for the subtree will contain the address of the header node instead of the *NULL* value. Accordingly, no pointer will ever contain an invalid address and the condition; *LEFT[HEAD] = HEAD* will indicate an empty tree.

THREADS & INORDER THREADING

Let us consider the linked representation of a binary tree **T**. Approximately half of the entries in the pointer field **LEFT** and **RIGHT** will contain *NULL* elements. Replacing the NULL entries by some other types of information may more efficiently use this space. Specifically we will replace some *NULL* entries by special pointers, which point to nodes higher in the tree. These special pointers re called threads, and a binary tree with such pointers are called **Threaded Binary Trees**.

The threads in a diagram of a threaded binary tree are indicated by dotted lines. In computer memory, an extra 1-bit **TAG** field may be used to distinguish threads from ordinary pointers or alternatively, threads may be denoted by negative integers, whereas ordinary pointers are denoted by positive integers.

Threading of a threaded binary tree corresponds to a particular traversal of **T**. There are **1-way Threading** or **2-way Threading**. Unless otherwise stated, our threading will correspond to the inorder traversal of **T**. Accordingly, in 1 way threading of **T**, a thread will appear in the right field of a node and will point to the next node in the inorder traversal of **T**. In 2-way threading of **T**, a thread will point to the preceding node in the inorder traversal of **T**. Furthermore, the left pointer of the 1st. node and the right pointer of the last node (in the inorder traversal of **T**) will contain the *NULL* value when T does not have a header node.

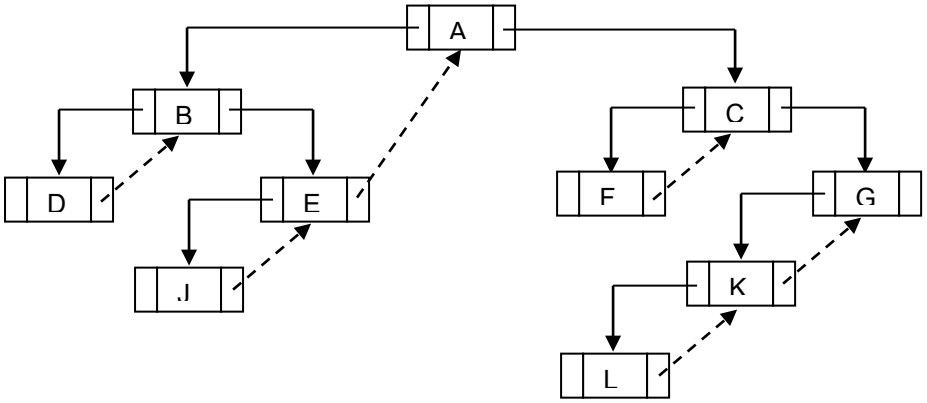


FIG: One-way inorder threading

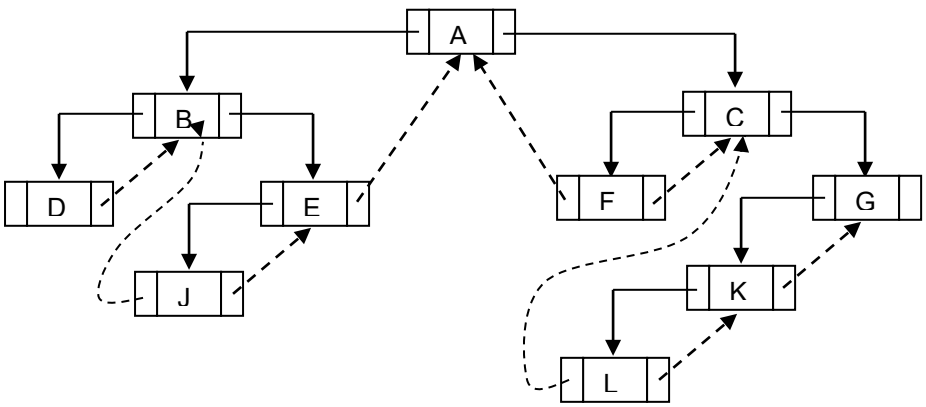


FIG: Two-way inorder threading

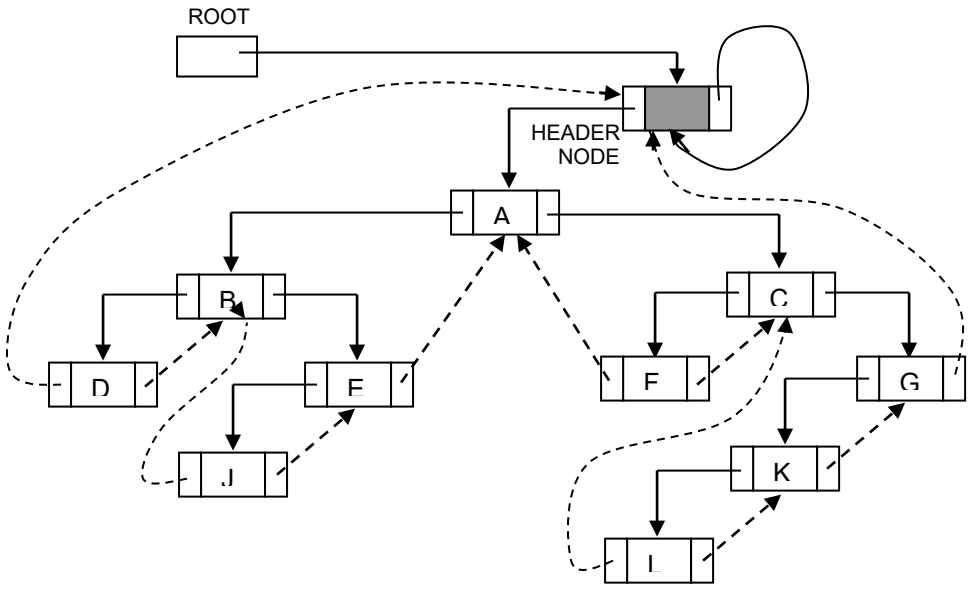


FIG: Two-way inorder threading with header node

H. W.

Q: Construct an inorder threaded binary tree using the following key values:
40, 20, 60, 50. 70, 65, 61, 75, 72, 71, 10, 30, 27, 25, 5, 7.

*/*Q: Write a program to perform the following operations on an inorder threaded binary tree,*

1. *Insert a node*
2. *Find inorder traversal*
3. *Find successor of a particular node*
4. *Find predessor of a particular node */*

```
# include <stdio.h>
# include <alloc.h>

# define POSITIVE 0 /* structural link */
# define NEGATIVE 1 /* threaded link */

struct node
  {
    int info ;
    struct node *left, *right;
    int lflag, rflag;
```

```
  };
typedef struct node ND ;

void insert ( ND **, int );
void insert_left(ND *, int);
void insert_right (ND *, int);
void inorder(ND *);
void predecessor (ND *, int);
void successor (ND *, int);

void main (void)
 {
  ND *tree;
  int i, x;
  tree=NULL;

  do
     {
        printf("\n 1 : Insert a node");
        printf("\n 2 : Display inorder traversal");
        printf("\n 3 : Find the successor of a particular node");
        printf("\n 4 : Find the predecessor of a particular node");
        printf("\n 0 : To exit");
        printf("\n\n Enter your choice : ");
        scanf("%d", &i);
        switch(i)
          {
            case 1: printf("\n Give the node value: ");
                    scanf("%d", &x);
                    insert( &tree, x);
                    break;
            case 2: inorder (tree);
                    break;
            case 3: printf("\n Give a particular noe value : ");
                    scanf("%d", & x);
                    successor(tree, x);
                    break;
            case 4: printf("\n Give a particular node value: ");
                    scanf("%d", &x);
                    predecessor(tree, x);
                    break ;
            case 0: printf("\n The end");
                    break ;
            default:printf("\n Invalid choice");
          }
     }while(i);
 }

void insert (ND  **ptr, int  item)
 {
   ND *temp, *current;
   if ( !*ptr )
```

```
    {
        temp = (ND *) malloc(sizeof(ND));
        if(!temp)
         {
            printf ("\n Insufficient memory");
            return ;
         }
        temp-> info =item;
        temp-> left =temp -> right = NULL ;
        temp-> lflag = temp->rflag = NEGATIVE;
        *ptr = temp;
        return;
    }
  current = *ptr;
  while(current->info != item)
   if(current->info > item)
     if(current ->lflag == POSITIVE)
         current = current ->left ;
     else
         {
           insert_left(current, item);
           return;
         }
    else
     if (current ->rflag == POSITIVE)
         current = current->right ;
     else
         {
           insert_right(current, item);
           return;
         }
 }/*end of function*/

 void insert_left(ND *p, int item)
  {
    ND *temp;
    temp = (ND *)malloc(sizeof(ND));
    if(!temp)
        {
          printf("\n Insufficient memory");
          return;
        }
    temp->info = item;
    temp->left = p->left;
    temp->right = p;
    temp->lflag = temp->rflag = NEGATIVE;
    p->left = temp;
    p->lflag = POSITIVE;
  }

 void insert_right(ND *p , int item)
  {
```

```
    ND *temp;
    temp = (ND *) malloc(sizeof(ND));
    if(!temp)
        {
          printf("\n Insufficient memory");
          return;
        }
    temp->info = item;
    temp->left = p;
    temp->right = p->right;
    temp->lflag = temp->rflag = NEGATIVE;
    p->right = temp;
    p->rflag = POSITIVE;
  }

 void inorder(ND *p)
  {
    while(p)
        {
          if(p->lflag == POSITIVE)
            p = p->left;
          while (p)
            {
                printf ("\t%d", p->info);
                if ( p -> rflag == POSITIVE)
                  {
                    p = p->right;
                    break;
                  }
                p = p->right;
            }
        }
  }

 void successor( ND *p, int item)
 {
   while (p!=NULL)
    {
        if ( p->info == item)
          break;
      if (p->info > item)
          if (p-> rflag == POSITIVE)
            p = p->left;
          else
           {
              p = NULL ;
              break;
           }
      else
          if ( p->rflag == POSITIVE)
            p = p->right;
          else
```

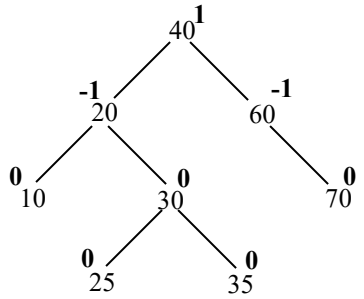
```
            {
              p = NULL;
              break ;
            }
      } /* end of while (p! = NULL) */
    if (!p)
      {
        printf ("\n Invalid node value");
        return;
      }
    if (p-> rflag == NEGATIVE)
      if (p->right != NULL)
          printf ("\n Successor of % d is %d ", p->info, p ->right->info);
      else
          printf ("\n Successor of %d does not exists", p -> info );
    else
      {
        p = p-> right;
          while ( p ->lflag == POSITIVE)
           p = p -> left;
        printf("\n Successor of % d is % d", item , p->info);
      }
  } /* end of function */

  void predecessor ( ND *p, int item)
     {
      while ( p != NULL)
           {
             if ( p-> info == item)
                   break;
             if (p->info > item)
               if (p->lflag == POSITIVE)
                     p = p->left;
               else
                     p = NULL;
              else
               if (p-> rflag == POSITIVE)
                     p = p->right;
               else
                     p = NULL;
           } /* end of while (p! =NULL) */
         if(!p)
           {
             printf ("\n  Invalid node value");
             return;
           }
         if(p-> lflag == NEGATIVE)
           if (p->left != NULL)
             printf ("\n predecessor of % d is %d", p->info , p->left->info);
           else
             printf ("\n predecessor of % d  does not exist", p->info);
         else
```

```
        {
          p = p->left;
          while (p->rflag == POSITIVE)
                p = p->right;
          printf ("\n Predecessor of % d is % d", item ,p->info);
        }
  }/* End of the function */
```

AVL TREE

Definition: A binary search tree is said to be an AVL tree or height-balanced tree if either it is left heavy or right heavy or balanced. The following binary search tree is an AVL –tree.

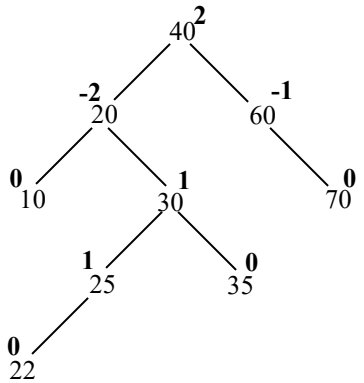


Balance factor: difference between the height of the left sub-tree and the height of the right sub-tree.

The balance factors of all the nodes must one of the following values:

1. (+1) ☐ *left heavy*
2. 0 ☐*balanced*
3. –1 ☐*right heavy*

The following tree (B.S.T) is not an AVL-tree, because it is not balanced.



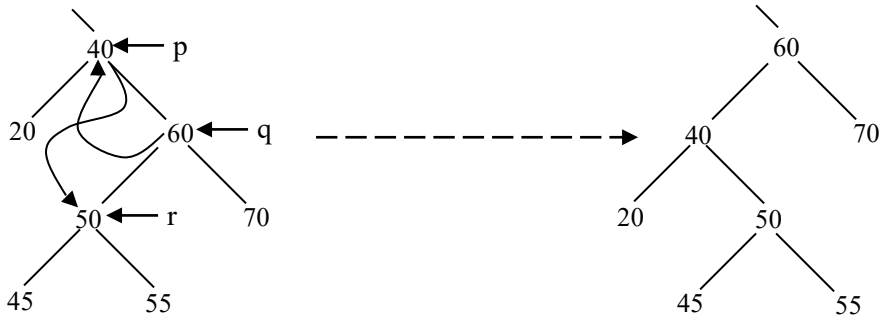
Left rotation algorithm: -

The left rotation algorithm about a node pointed by 'p' can be described as follows,

Step1: q = p -> right;

Step2: r = q -> left;
Step3: q -> left = p;
Step4: p -> right = r;

Left rotation about the node 40.

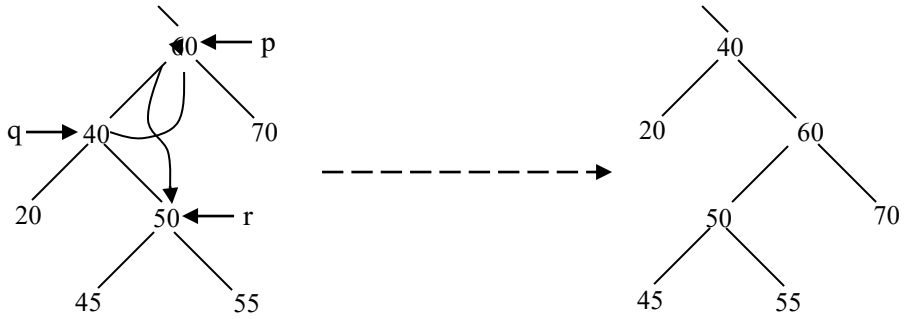


Right rotation algorithm: -

The right rotation algorithm about a node pointed by 'p' can be described as follows,

Step1: q = p -> left;
Step2: r = q -> right;
Step3: q -> right = p;
Step4: p -> left = r;

Right rotation about the node 60.



Rules for construction of an AVL tree: -

1. ***Left – left (LL) rule*:** Right rotation only about the unbalanced node.
2. ***Right-right (RR) rule***: Left rotation only about the unbalanced node.
3. ***Left-right (LR) rule***: Left rotation about the left child of the unbalanced node and then right rotation about the unbalanced node.
4. ***Right-left (RL) rule***: Right rotation about the right child of the unbalanced node and
   then. left rotation about the unbalanced node.

1. Q: Construct an AVL tree (Height balanced tree) from the following key values:

1. A , Z , B , Y , C , X , D , W , E , V , T.
2. 50 , 45, 80, 95, 26, 48, 105, 2.
3. JAN, FEB, MAR, … … … ,DEC.