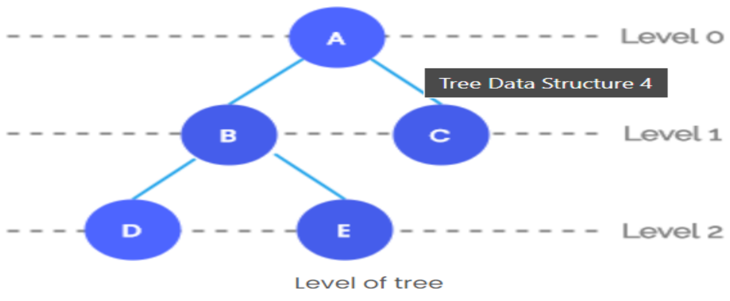
# SCHEME

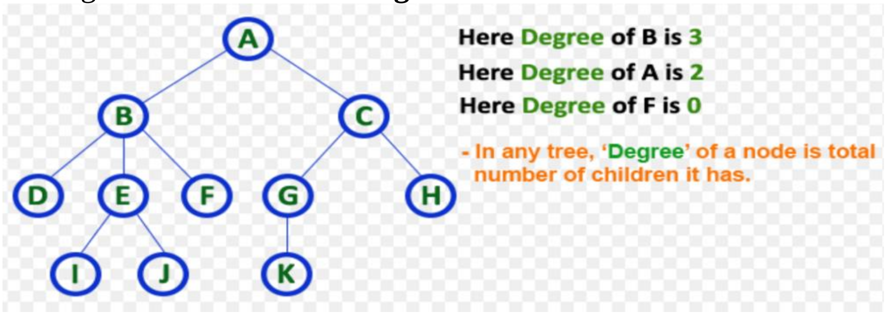
**1a. What is a tree? With suitable examples, define: i)Level of binary tree, ii)Degree of the tree, iii)Ancestors, iv)Complete binary tree, v)Skewed binary tree**

A tree is a non-linear data structure defined as a collection of objects or entities known as nodes that are linked together to represent or simulate hierarchy.

i) In a tree, each step from top to bottom is called as level of a tree. The level count starts with 0 and increments by 1 at each level or step.

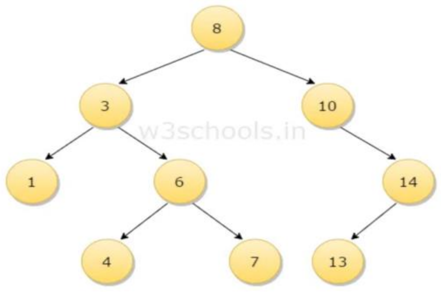


ii) The total number of children of a node is the **degree** of a node. The number of edges arriving at that node is called the **in-degree** of a node. The number of edges leaving that node is the **out-degree** of a node.

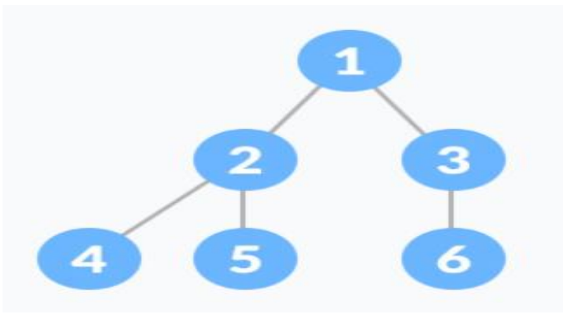


iii) Any node *y* on the (unique) path from root *r* to node *n* is an *ancestor* of node *n*. Every node is an ancestor of itself.

iv) A binary tree is a special type of tree in which every node or vertex has either no child node or one child node or two child nodes. A binary tree is an important class of a tree data structure in which a node can have at most two children.



v) A complete binary tree is a binary tree in which all the levels are completely filled except possibly the lowest one, which is filled from the left.



**b. What are the advantages of the threaded binary tree over binary tree? Explain he construction of threaded binary tree for 10, 20, 30, 40 and 50.**

- **No need for stacks or recursion:** Unlike binary trees, threaded binary trees do not require a stack or recursion for their traversal.
- **Optimal memory usage:** Another advantage of threaded binary tree data structure is that it decreases memory wastage. In normal binary trees, whenever a node's left/right pointer is NULL, memory is wasted. But with threaded binary trees, we are overcoming this problem by storing its inorder predecessor/successor.
- **Time complexity:** In-order traversal in a threaded binary tree is fast because we get the next node in O(1) time than a normal binary tree that takes O(Height). But insertion and deletion operations take more time for the threaded binary tree.
- **Backward traversal:** In a double-threaded binary tree, we can even do a backward traversal.

**2. Design, develop and implement a menu driven program in C for the following operations on Binary Search Tree(BST) of integers,**
**i) Create a BST of N integers: 6, 9, 5, 2, 8, 15, 24, 14, 7, 8, 5, 2**
**ii) Traverse the BST in Inorder, Preorder and Postorder**
**iii) Search the BST for a given element(KEY) and report the appropriate message**
**iv) Exit.**

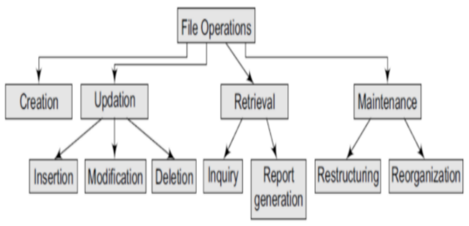
```
#include <stdio.h>

#include <stdlib.h>

int flag=0;

typedef struct BST

{

  int data;

  struct BST *lchild,*rchild;

} node;

void insert(node *, node *);

void inorder(node *);
void preorder(node *);
void postorder(node *);
node *search(node *, int, node **);
void main()
{
  int choice;
  int ans =1;
  int key;
  node *new_node, *root, *tmp, *parent;
  node *get_node();
  root = NULL;
  printf("\nProgram For Binary Search Tree ");
  do
  {
    printf("\n1.Create");
    printf("\n2.Search");
    printf("\n3.Recursive Traversals");
    printf("\n4.Exit");
    printf("\nEnter your choice :");
    scanf("%d", &choice);
    switch (choice)
    {
      case 1:
             do
             {
```

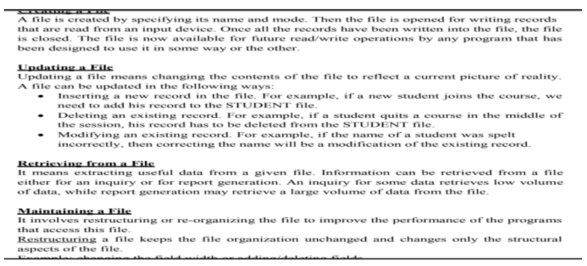
```
          new_node = get_node();

          printf("\nEnter The Element ");

          scanf("%d", &new_node->data);

          if (root == NULL)

          root = new_node;

          else

          insert(root, new_node);

          printf("\nWant To enter More Elements?(1/0)");

          scanf("%d",&ans);

        } while (ans);

        break;

case 2:

        printf("\nEnter Element to be searched :");

        scanf("%d", &key);

        tmp = search(root, key, &parent);

        if(flag==1)

        {

          printf("\nParent of node %d is %d", tmp->data, parent->data);

        }

        else

        {

          printf("\n The %d Element is not Present",key);

        }

        flag=0;

        break;
```

```
   case 3:
       if (root == NULL)
       printf("Tree Is Not Created");
       else
       {
         printf("\nThe Inorder display :");
         inorder(root);
         printf("\nThe Preorder display : ");
         preorder(root);
         printf("\nThe Postorder display : ");
         postorder(root);
       }
       break;

  }
  while(choice!=4);
 }
}
node *get_node()
{
 node *temp;
 temp = (node *) malloc(sizeof(node));
 temp->lchild = NULL;
 temp->rchild = NULL;
 return temp;
```

```
}

void insert(node *root, node *new_node)

{

  if (new_node->data < root->data)

  {

    if(root->lchild==NULL)

      root->lchild=new_node;

    else

      insert(root->lchild, new_node);

  }

  if (new_node->data > root->data)

  {

    if (root->rchild == NULL)

      root->rchild = new_node;

    else

      insert(root->rchild, new_node);

  }

}

node *search(node *root, int key, node **parent)

{

  node *temp;

  temp = root;

  while (temp != NULL)

  {

    if (temp->data == key)
    {
      printf("\nThe %d Element is Present", temp->data);
      flag=1;
      return temp;
    }
    *parent = temp;
    if (temp->data > key)
      temp = temp->lchild;
    else
      temp = temp->rchild;
  }
  return NULL;
}
void inorder(node *temp)
{
  if (temp != NULL)
  {
    inorder(temp->lchild);
    printf("%d\t", temp->data);
    inorder(temp->rchild);
  }
}
void preorder(node *temp)
{
  if (temp != NULL)
  {
    printf("%d\t", temp->data);
    preorder(temp->lchild);
    preorder(temp->rchild);
  }
}
void postorder(node *temp)
{
  if (temp != NULL)
  {
    postorder(temp->lchild);
    postorder(temp->rchild);
    printf("%d\t", temp->data);
  }
}
```

**5. Briefly explain basic operations that can be performed on a file. Explain different types of file organisation.**




Creating a File
A file is created by specifying its name and mode. Then the file is opened for writing records that are read from an input device. Once all the records have been written into the file, the file is closed. The file is now available for future read/write operations by any program that has been designed to use it in some way or the other.

Updating a File
Updating a file means changing the contents of the file to reflect a current picture of reality. A file can be updated in the following ways:

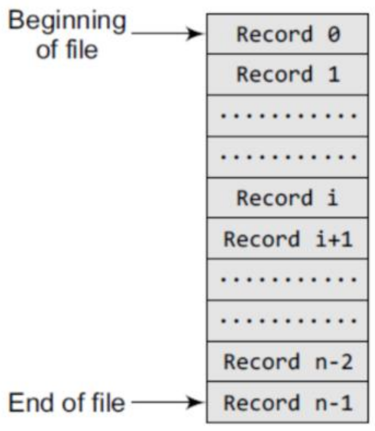
- Inserting a new record in the file. For example, if a new student joins the course, we need to add his record to the STUDENT file.
- Deleting an existing record. For example, if a student quits a course in the middle of the session, his record has to be deleted from the STUDENT file.
- Modifying an existing record. For example, if the name of a student was spelt incorrectly, then correcting the name will be a modification of the existing record.

Retrieving from a File
It means extracting useful data from a given file. Information can be retrieved from a file either for an inquiry or for report generation. An inquiry for some data retrieves low volume of data, while report generation may retrieve a large volume of data from the file.

Maintaining a File
It involves restructuring or re-organizing the file to improve the performance of the programs that access this file.
Restructuring a file keeps the file organization unchanged and changes only the structural aspects of the file.
Example: changing the field width or adding/deleting fields

Types of file organisations are:

i) Sequential Organization: A sequentially organized file stores the records in the order in which they were entered. Sequential files can be read only sequentially, starting with the first record in the file. Sequential file organization is the most basic way to organize a large collection of records in a file.



Features:

- Records are written in the order in which they are entered
- Records are read and written sequentially
- Deletion or updation of one or more records calls for replacing the original file with a new file that contains the desired changes
- Records have the same size and the same field format
- Records are sorted on a key value
- Generally used for report generation or sequential reading

Advantages:

- Simple and easy to Handle
- No extra overheads involved
- Sequential files can be stored on magnetic disks as well as magnetic tapes
- Well suited for batch– oriented applications

Disadvantages:

- Records can be read only sequentially. If ith record has to be read, then all the i–1 records must be read
- Does not support update operation. A new file has to be created and the original file has to be replaced with the new file that contains the desired changes
- Cannot be used for interactive applications

ii) Relative File Organization: If the records are of fixed length and we know the base address of the file and the length of the record, then any record i can be accessed using the following formula:

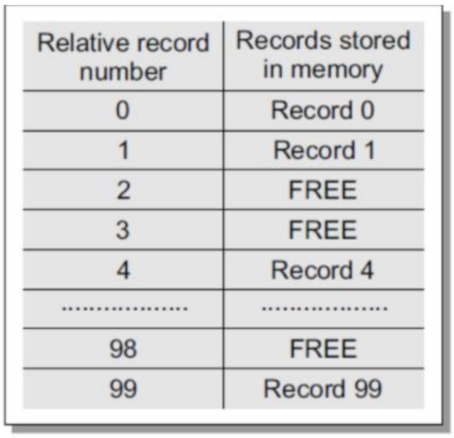
Address of ith record = base_address + (i–1) * record_length

Consider the base address of a file is 1000 and each record occupies 20 bytes, then the address of the 5th record can be given as:

1000 + (5–1) * 20

= 1000 + 80

= 1080



| Relative record number | Records stored in memory |
|---|---|
| 0 | Record 0 |
| 1 | Record 1 |
| 2 | FREE |
| 3 | FREE |
| 4 | Record 4 |
| .................. | .................. |
| 98 | FREE |
| 99 | Record 99 |

Features:

- Provides an effective way to access individual records
- The record number represents the location of the record relative to the beginning of the file
- Records in a relative file are of fixed length
- Relative files can be used for both random as well as sequential access
- Every location in the table either stores a record or is marked as FREE

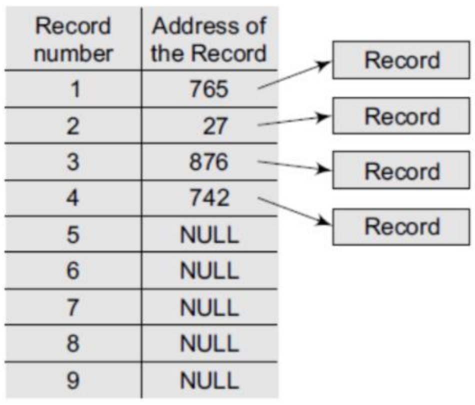
Advantages:

- Ease of processing
- If the relative record number of the record that has to be accessed is known, then the record can be accessed instantaneously
- Random access of records makes access to relative files fast
- Allows deletions and updations in the same file
- Provides random as well as sequential access of records with low overhead
- New records can be easily added in the free locations based on the relative record number of the record to be inserted
- Well suited for interactive applications

Disadvantages:

- Use of relative files is restricted to disk devices
- Records can be of fixed length only
- For random access of records, the relative record number must be known in advance

iii) Indexed Sequential File Organization



| Record number | Address of the Record |
|---|---|
| 1 | 765 |
| 2 | 27 |
| 3 | 876 |
| 4 | 742 |
| 5 | NULL |
| 6 | NULL |
| 7 | NULL |
| 8 | NULL |
| 9 | NULL |

Features:

- Provides fast data retrieval
- Records are of fixed length
- Index table stores the address of the records in the file
- The ith entry in the index table points to the ith record of the file
- While the index table is read sequentially to find the address of the desired record, a direct access is made to the address of the specified record in order to access it randomly
- Indexed sequential files perform well in situations where sequential access as well as random access is made to the data

Advantages:

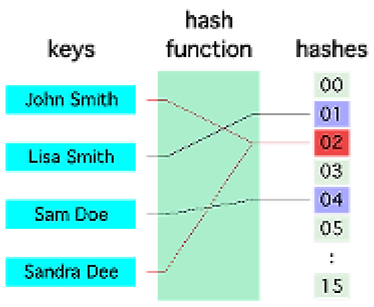
- The key improvement is that the indices are small and can be searched quickly, allowing the database to access only the records it needs
- Supports applications that require both batch and interactive processing
- Records can be accessed sequentially as well as randomly
- Updates the records in the same file

Disadvantages:

- Indexed sequential files can be stored only on disks
- Needs extra space and overhead to store indices
- Handling these files is more complicated than handling sequential files
- Supports only fixed length records.

**4a. What is collision? What are the methods to resolve collision? Explain linear probing with an example.**

Situation of collision occurs when more than one keys (hash functions) map to the same location of hashes. In this situation, two or more data elements qualify to be mapped to the same location in hash table.



Collision resolution can be done using two techniques:

**1. Open Addressing**

**2. Chaining**

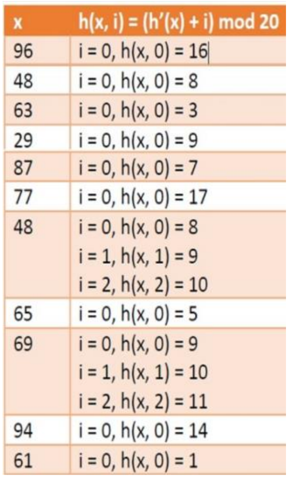
**Open Addressing:** In this technique a hash table with per-identified size is considered. All items are stored in the hash table itself. In addition to the data, each hash bucket also maintains the three states: EMPTY, OCCUPIED, DELETED. While inserting, if a collision occurs, alternative cells are tried until an empty bucket is found. For which one of the following technique is adopted.

- 1.Liner Probing(this is prone to clustering of data + Some other constrains)
- 2.Quadratic probing
- 3.Double hashing(in short in case of collision another hashing function is used with the key value as an input to identify where in the open addressing scheme the data should actually be stored.)

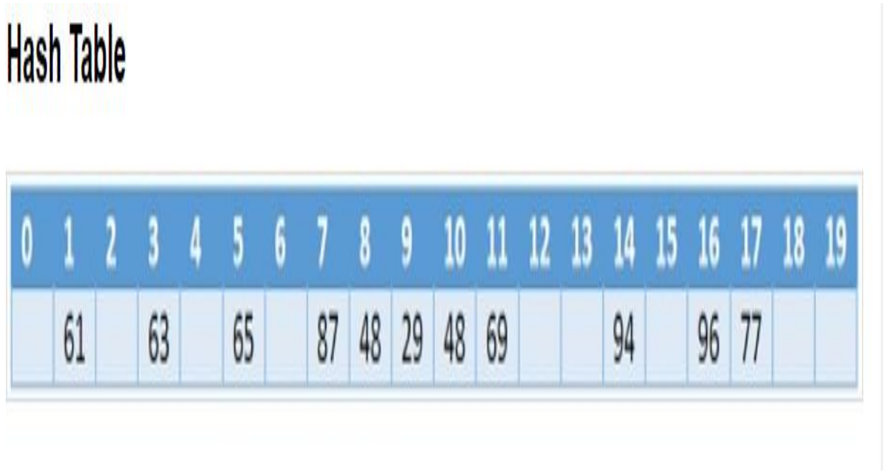
**Chaining:** Open Hashing, is a technique in which the data is not directly stored at the hash key index (k) of the Hash table. Rather the data at the key index (k) in the hash table is a pointer to the head of the data structure where the data is actually stored. In the most simple and common implementations the data structure adopted for storing the element is a linked-list.

Linear Probing:

Suppose we have a list of size 20 (m = 20). We want to put some elements in linear probing fashion. The elements are {96, 48, 63, 29, 87, 77, 48, 65, 69, 94, 61}



| x | h(x, i) = (h′(x) + i) mod 20 |
|---|---|
| 96 | i = 0, h(x, 0) = 16 |
| 48 | i = 0, h(x, 0) = 8 |
| 63 | i = 0, h(x, 0) = 3 |
| 29 | i = 0, h(x, 0) = 9 |
| 87 | i = 0, h(x, 0) = 7 |
| 77 | i = 0, h(x, 0) = 17 |
| 48 | i = 0, h(x, 0) = 8<br>i = 1, h(x, 1) = 9<br>i = 2, h(x, 2) = 10 |
| 65 | i = 0, h(x, 0) = 5 |
| 69 | i = 0, h(x, 0) = 9<br>i = 1, h(x, 1) = 10<br>i = 2, h(x, 2) = 11 |
| 94 | i = 0, h(x, 0) = 14 |
| 61 | i = 0, h(x, 0) = 1 |



Hash Table

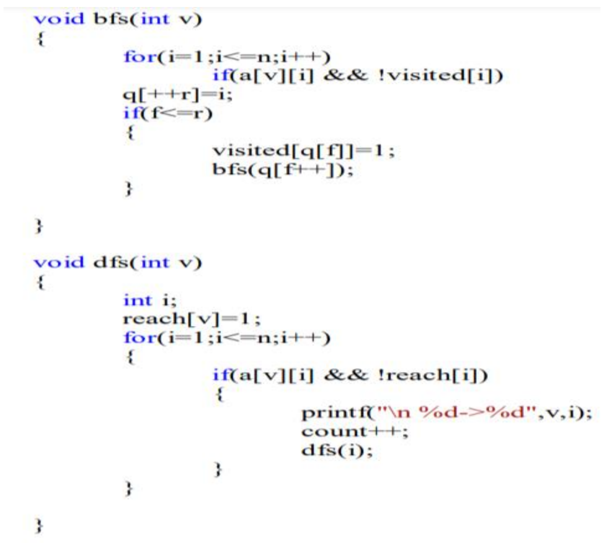
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
|  | 61 |  | 63 |  | 65 |  | 87 | 48 | 29 | 48 | 69 |  |  | 94 |  | 96 | 77 |  |  |

**4b. Write a C function for insertion sort. Sort the following list using insertion sort. 50,30,10,60,40,20,60.**

```
void insertionSort(element all, int n)
      {
         int j;
         for (j = 2i j <= n : j++)
         {
                  element temp = a[j];
                  insert (temp, a, j-1);
         }
      }
```

| j | [10] | [20] | [30] | [40] | [50] | [60] | [70] |
|---|---|---|---|---|---|---|---|
| - | 50 | 30 | 10 | 70 | 40 | 20 | 60 |
| 20 | 30 | 50 | 10 | 70 | 40 | 20 | 60 |
| 30 | 10 | 30 | 50 | 70 | 40 | 20 | 60 |
| 40 | 10 | 30 | 50 | 70 | 40 | 20 | 60 |
| 50 | 10 | 30 | 40 | 50 | 70 | 20 | 60 |
| 60 | 10 | 20 | 30 | 40 | 50 | 70 | 60 |
| 70 | 10 | 20 | 30 | 40 | 50 | 60 | 70 |

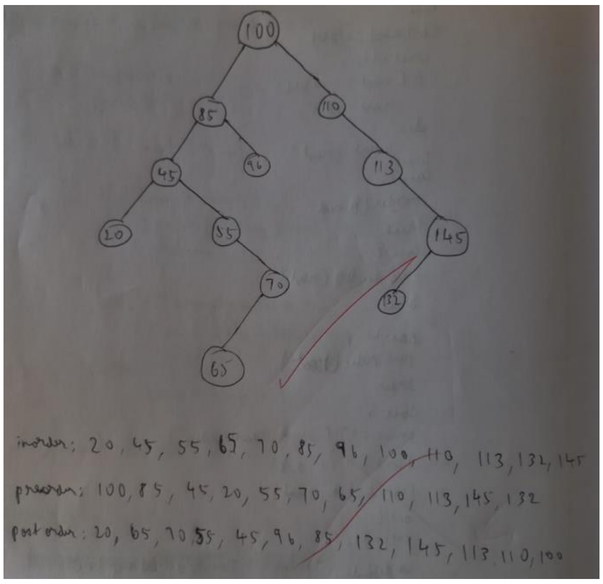
**6. Write a C function for BFS and DFS, show BFS and DFS traversal for the graph given below. Also give the adjacency matrix and adjacency list representation.**



```
void bfs(int v)
{
        for(i=1;i<=n;i++)
                if(a[v][i] && !visited[i])
        q[++r]=i;
        if(f<=r)
        {
                visited[q[f]]=1;
                bfs(q[f++]);
        }

}

void dfs(int v)
{
        int i;
        reach[v]=1;
        for(i=1;i<=n;i++)
        {
                if(a[v][i] && !reach[i])
                {
                        printf("\n %d->%d",v,i);
                        count++;
                        dfs(i);
                }
        }

}
```

**3. Construct:**

**i) Binary Search Tree: 100, 85, 45, 55, 110, 20, 70, 65, 113, 145, 132, 96. Also give in-oder, pre-order and post-order traversal.**



inorder: 20, 45, 55, 65, 70, 85, 96, 100, 110, 113, 132, 145

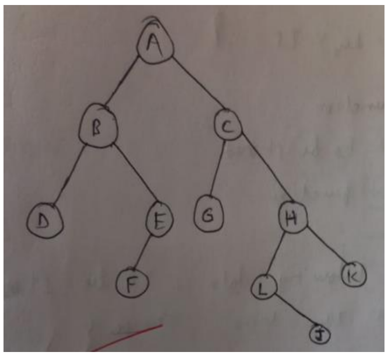
preorder: 100, 85, 45, 20, 55, 70, 65, 110, 113, 145, 132

post order: 20, 65, 70, 55, 45, 96, 85, 132, 145, 113, 110, 100

**ii) Binary Tree:**

**Preorder: A, B, D, E, F, C, G, H, L, J, K**

**Inorder: D, B, F, E, A, G, C, L, J, H, K**


A
B
C
D
E
G
H
F
L
K
J