Binary tree traversal

Binary Tree traversals

- In whatever way we store the data, either as a linked list or array, we should be able to traverse all the data or access all the data
- Linked list and array are linear data structures and we can visit all the elements in a sequence without the possibility of visiting an element twice or more
- But, in a Binary Tree, there is no such linear order
- So, how to enumerate or visit or traverse all the elements of a binary tree such that we pass through the nodes only once???

Binary Tree traversals

Three methods of traversing

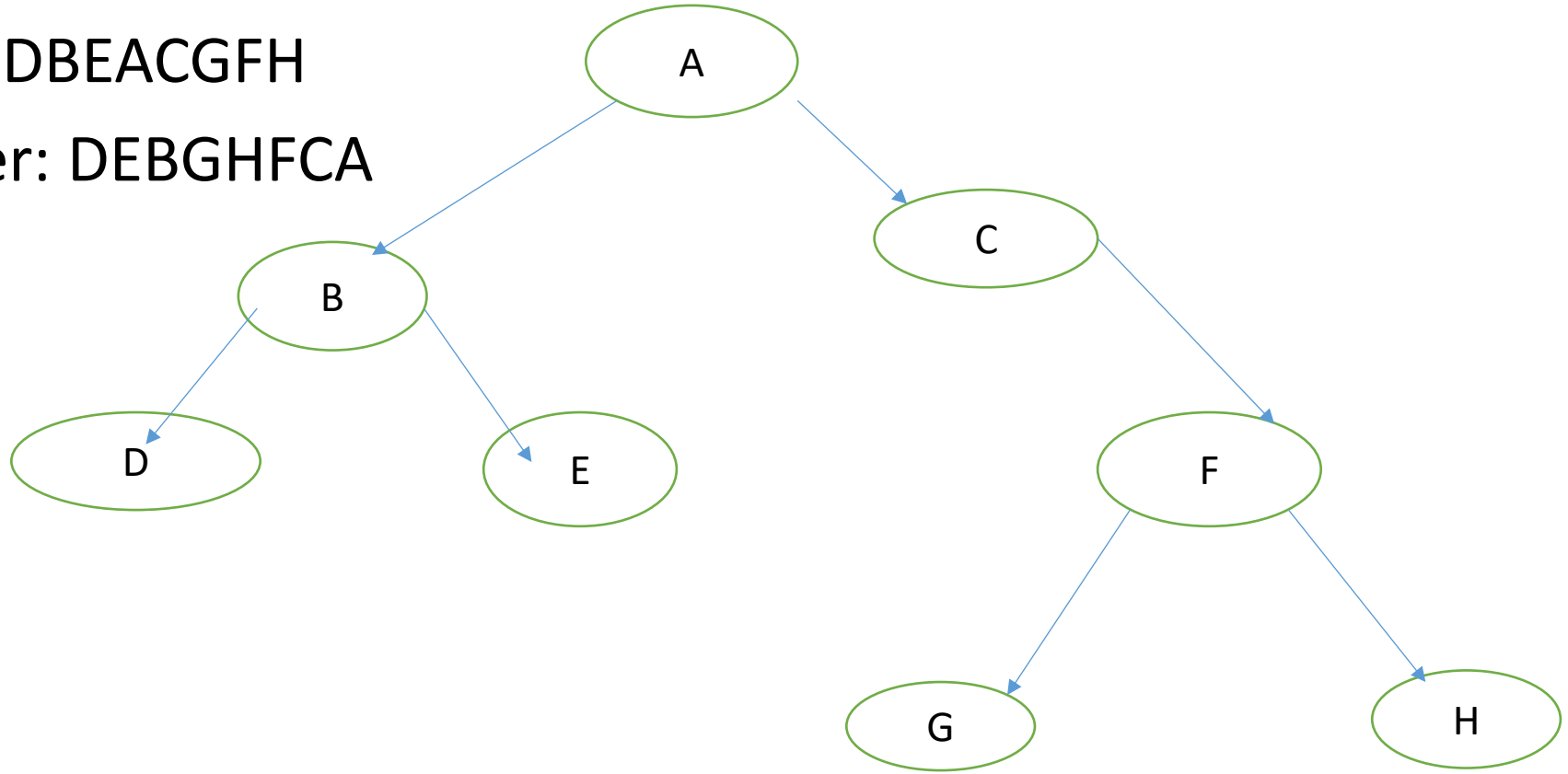
- **Preorder:** **1.** Visit the root
  2. Traverse left subtree in preorder
  3. Traverse right subtree in preorder
- **Inorder:** **1**. Traverse left subtree in inorder
  2. Visit the root
  3. Traverse right subtree in inorder
- **Postorder:** **1.** Traverse left subtree in postorder
  2. Traverse right subtree in postorder
  3. Visit the root

Binary Tree Traversal

Preorder: ABDECFGH

Inorder: DBEACGFH

Postorder: DEBGHFCA

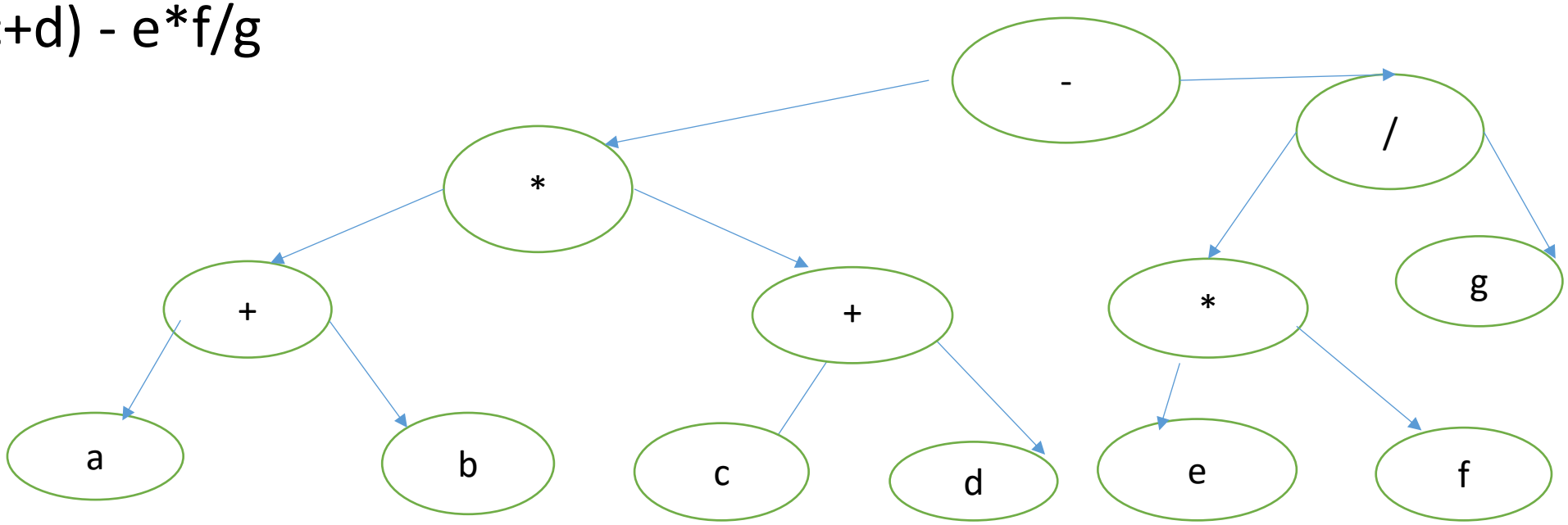


Applications of Binary Tree

- We can represent mathematical expression containing operands and binary operators by a strictly binary tree
- The root of the strictly binary tree contains an operator that is to be applied to the results of expressions represented by left and right subtrees
- A node representing operator is always a non leaf node whereas node representing the operand will be a leaf node
- For the expression (a+b) *(c+d) - e*f/g, the tree will be

Binary Tree for mathematical expression

(a+b) *(c+d) - e*f/g



Traversals of expression tree

- Preorder is equivalent to prefix
- Postorder is postfix

Few functions of binary tree

- The number of nodes in a binary tree
- The sum of the contents of all nodes of a binary tree
- The depth or height of a binary tree
- Whether we should have recursive function for above operations or we can have non recursive also??

Count the nodes of a Binary tree

```
int count_nodes( binarytree * root)
{
   if (root == null)
      return 0;
   else
        return(1 + count_nodes(root->left) + count_nodes(root->right));
}
```

Height of a Binary tree

```
int height( binarytree * root)
{ int h1,h2;
   if (root == null)
      return 0;
   else
       { h1 = 1 + height(root->left);
        h2 = 1+ height(root->right);
       }
    If(h1>h2) return h1;
   else return h2;
}
```

Applications of Binary Tree

Find the list of duplicates in a list of numbers or some data

What are the solutions??

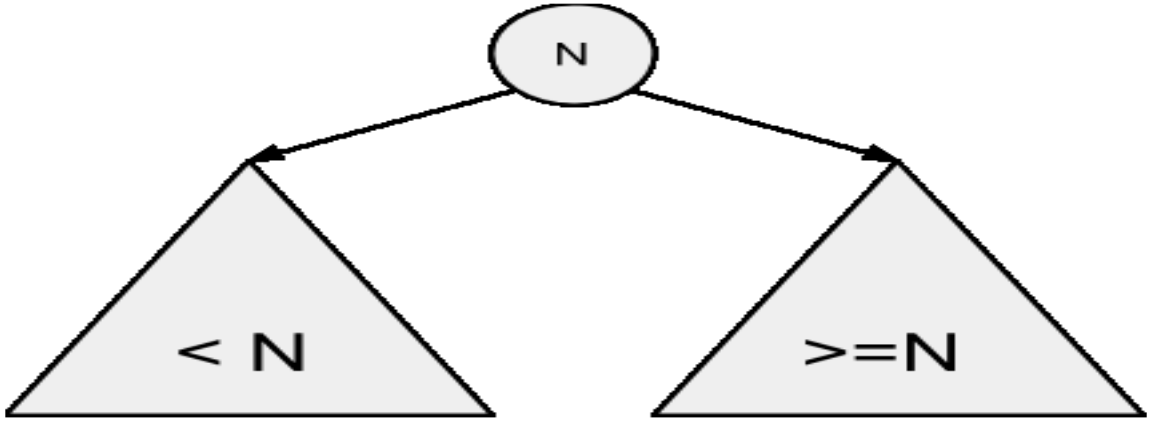
1.

2.

3.

Binary Search Tree

- A binary search tree is a binary tree with additional properties:
- The value stored in root node is greater than the value stored in its left child and all its descendants
- The value stored in root node is smaller than or equal to the value stored in its right child and all its descendants



BST

- A binary search tree (BST) is a tree in which all nodes have the following properties −
- The left sub-tree of a node has key less than its parent node's key.
- The right sub-tree of a node has key greater than or equal to its parent node's key.
- Thus, a binary search tree (BST) divides all its sub-trees into two segments; *left* sub-tree and *right* sub-tree and can be defined as −
- ***All the contents in the nodes of left subtree are less than the contents of root and all the contents of nodes in the right subtree are greater or equal to the contents of the root node.***
- **In order traversal of BST prints the data in ascending order**

Node representation of Binary tree

- ***Only complete binary tree and almost complete binary tree can be implemented using implicit array representation i.e. without using explicit pointers***
- In other cases, linked representation is used which can be implemented as array elements or using dynamic nodes. Mostly dynamic nodes are used for the representation of binary tree.
- **The two structures of a dynamic node of a binary tree are:**
- struct treenode {int info; struct treenode * left; struct treenode * right;}
- struct nodetype2 { int info;
  struct nodetype2 * left;
  struct nodetype2 * right;
  struct nodetype2 * father;
  }

Main Operations on binary tree

- ***Traverse the tree***
- ***Insert a node in the tree***
- ***Delete a node from the tree***
- ***Search the tree for some data***

How to build a binary search tree

1. Start with the root node

2. Inserting the next node by attaching it to its left or its right starting from the root

3. Repeat the step two till all the data is inserted

Insert data in a BST

Whenever a data element is to be inserted in the tree:

- First create a new node
- Insert data in the node
- Find correct location for node in the tree and connect it properly.

Steps are:

1. Start search from the root node then if data is less than key value, search empty location in left subtree and insert the data.
2. Otherwise search empty location in right subtree and insert the data.
3. To connect, keep the address of the father node of empty location and put the address of ***newnode*** in left child if value is less otherwise put it in the right child

Steps for Building/creating a BST or /storing data in a BST

- Create the structure of the node : bstnode
- Create a root pointer and initialize it to NULL : root
- Read the data in a variable
- Create a node and store the data into that node: new_node
- Set left and right pointers of new_node to NULL
- If (root ==NULL) root = new_node;

  else

    find_father(root,

Insertion steps

```
 struct treenode * temp;
temp = (treenode*) malloc(sizeof(treenode)):
temp->data = x;
temp->left = NULL;
temp->right = NULL;
If(root == NULL) root = temp;
else // after searching the location of node to which this node must be attached and finding the address:
If (father->left == NULL) father->left = temp; else father->right = temp;
```

Search a value in BST

```
struct treenode* search(int x, struct treenode *root)
{ struct treenode *current = root;
while(current->data != x && current !=NULL)
{  if(current->data > x){ current = current->left; }
  else { current = current->right  ; }
}
if(current == NULL){ return NULL; }
else
 return current; }
```

Insert function

```
void insert(int x, struct node *root)
{ struct node *temp = (struct node*) malloc(sizeof(struct node));
struct node *current; struct node *parent;
temp->data = x;
temp->left = NULL; temp->right = NULL;
if(root == NULL){ root = temp; }
```

```
else { current = root; parent = NULL;
        while(1){ parent = current;
                    if(x < parent->data){
              current = current->left;
              if(current == NULL){ parent->left = temp; return; } }
                    else
            { current = current->right;
             if(current == NULL){ parent->right = temp; return; } } } } }
```

Deleting a node from a BST

Delete a node (dnode) having key x from a BST

- There are three cases to consider

1. dNode is a leaf
2. dNode has only one subtree
3. dNode has two subtrees

Case 1: Find the dnode and keep the address of its father

Simply delete the dnode by making left or right child of father as NULL

Case 2: ***If the dnode has one subtree then attach this subtree with the father node (attach descendent of dnode to dnode's father)***

Deleting a node from a BST

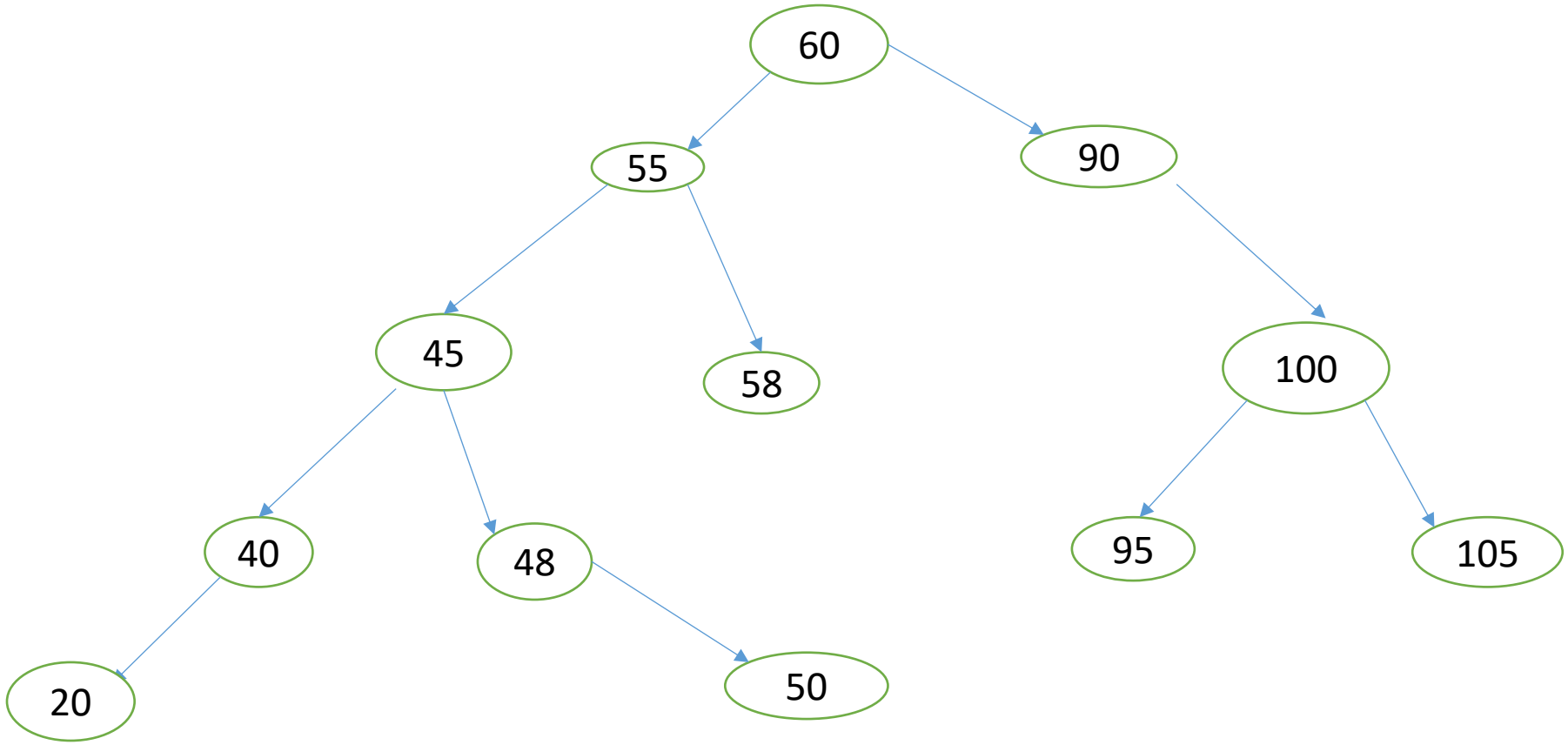
Node (d**node**) has two subtrees:

Case 3: If dnode has both the subtrees then dnode can not be deleted

Let us examine the immediate successor or predecessor of dnode:

- The immediate successor node or predecessor node of the **dnode** will always be a leaf node or a node with one subtree only
- So, we replace the value of **dnode** with the value of immediate successor or predecessor node and then delete that node using case1 or case2

Examples…



Code for node deletion (case 1)

Prob: delete a node having some value X

Steps:

1. Search the tree for value X and if value X exists in the tree (*if value X is not there just come out of delete function)*
2. Keep the address of that node(say ***mynode***)
3. Keep the address of its father (say ***father***) and then perform following operations:

If(mynode->left == NULL and mynode->right == NULL) //check mynode is a leaf node or not

{if (father->left == mynode) father->left = NULL; (Checking whether left child or right child)

else father->right = NULL;

free(mynode);}

Code for node deletion(case 2)

```
else {If((mynode->left == NULL and mynode->right != NULL) ||
            (mynode->left != NULL and mynode->right == NULL)) //mynode has one subtree
{if (father->left == mynode)   //If mynode is left child of father node
          {if (mynode->left == NULL)  //checking whether mynode has left subtree or right subtree
            father->left = mynode->right;
            else father->left = mynode->left;
            }
```

Cont..

```
else// If mynode is right child of father node
{if (mynode->left == NULL) //checking whether mynode has left subtree or right subtree
father->right = mynode->right;
else father->right = mynode->left;}
}
```

Code for node deletion(case 3)

```
else {  //when mynode has two children
        temp = mynode->right; // finding immediate successor
        father_temp = mynode;
  while(temp->left !=NULL)
 {father_temp = temp; temp = temp->left;}
mynode->data =  temp->data; // replacing with successor
If(temp->left == NULL and temp->right == NULL) //node to be deleted is leaf
fun-case1(temp,father_temp);
else
fun-case2(temp,father_temp);   /node to be deleted has one subtree
}
```

Efficiency of binary tree operations like search, insert and delete..

- The complexity of insertion, deletion is same as searching a key in the binary search tree or binary tree.
- The time required to search a binary search tree varies between O(n) and $O(\log_2 n)$ (Due to the structure of the tree)
- Structure of tree depends on the order in which elements are entered
- It has been seen that if data is presented in random order, balanced trees occur more often and on average search time remains closer to $O(\log_2 n)$

Balanced trees

- If the probability of searching a key in a table is same then a balanced tree gives the most efficient search results
- What do we mean by balanced??