Data Structure

Lecture 9

Dr. Ahmed Fathalla

Binary Search

Searching for 653

[061	087	154	170	275	426	503	509	512	612	653	677	703	765	897	908]
061	087	154	170	275	426	503	509	[512	612	653	677	703	765	897	908]
061	087	154	170	275	426	503	509	[512	612	653]	677	703	765	897	908
061	087	154	170	275	426	503	509	512	612	[653]	677	703	765	897	908

Searching for 400

[061	087	154	170	275	426	503	509	512	612	653	677	703	765	897	908]
[06	087	154	170	275	426	503]	509	512	612	653	677	703	765	897	908
061	087	154	170	[275	426	503]	509	512	612	653	677	703	765	897	908
061	087	154	170	[275]	426	503	509	512	612	653	677	703	765	897	908
061	087	154	170	275]	[426	503	509	512	612	653	677	703	765	897	908

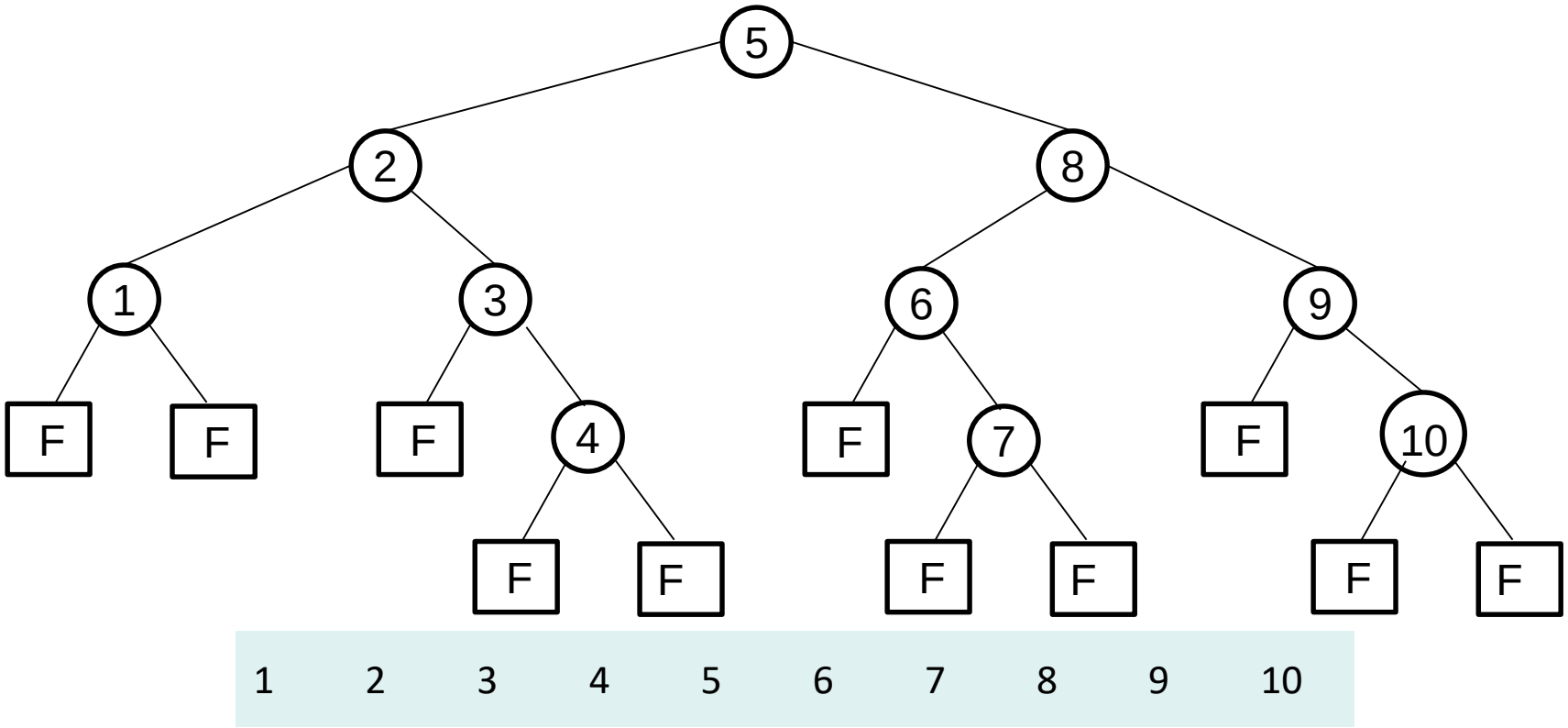
```
/*pre: list is ordered     Post: location returned, O.W. -1*/
int RecBinary(List *pl, KeyType k, int bottom, int top){
   int middle;
   KeyType mid_item;
   if (bottom<=top){
       middle=(bottom+top)/2;
       RetrieveItem(middle, &mid_item, pl)
       if ( k == mid_item )
         return middle;
       else if ( k < mid_item )
          return RecBinary(pl, k, bottom, middle-1);
       else
          return RecBinary(pl, k, middle+1, top);
   }
   return -1;
}
int RecBinarySearch(KeyType k, List *pl){
       return RecBinary(pl, k, 0, pl->size-1);
}
```

[061	087	154	170	275	426	503	509	512	612	653	677	703	765	897	908]
061	087	154	170	275	426	503	509	[512	612	653	677	703	765	897	908]
061	087	154	170	275	426	503	509	[512	612	653]	677	703	765	897	908

```
/*pre: list is ordered  Post: location returned, O.W. -1*/
int BinarySearch(KeyType k, List *pl){
    int middle, bottom=0, top=pl->size-1;
    KeyType mid_item;
    while(bottom<=top){
        middle=(bottom+top)/2;
        RetrieveItem(middle, &mid_item, pl)
        if ( k == mid_item )
            return middle;
        else if ( k < mid_item )
            top=middle-1;
        else
            bottom=middle+1;
    }
    return -1;
}
```

[061	087	154	170	275	426	503	509	512	612	653	677	703	765	897	908]
061	087	154	170	275	426	503	509	[512	612	653	677	703	765	897	908]
061	087	154	170	275	426	503	509	[512	612	653]	677	703	765	897	908

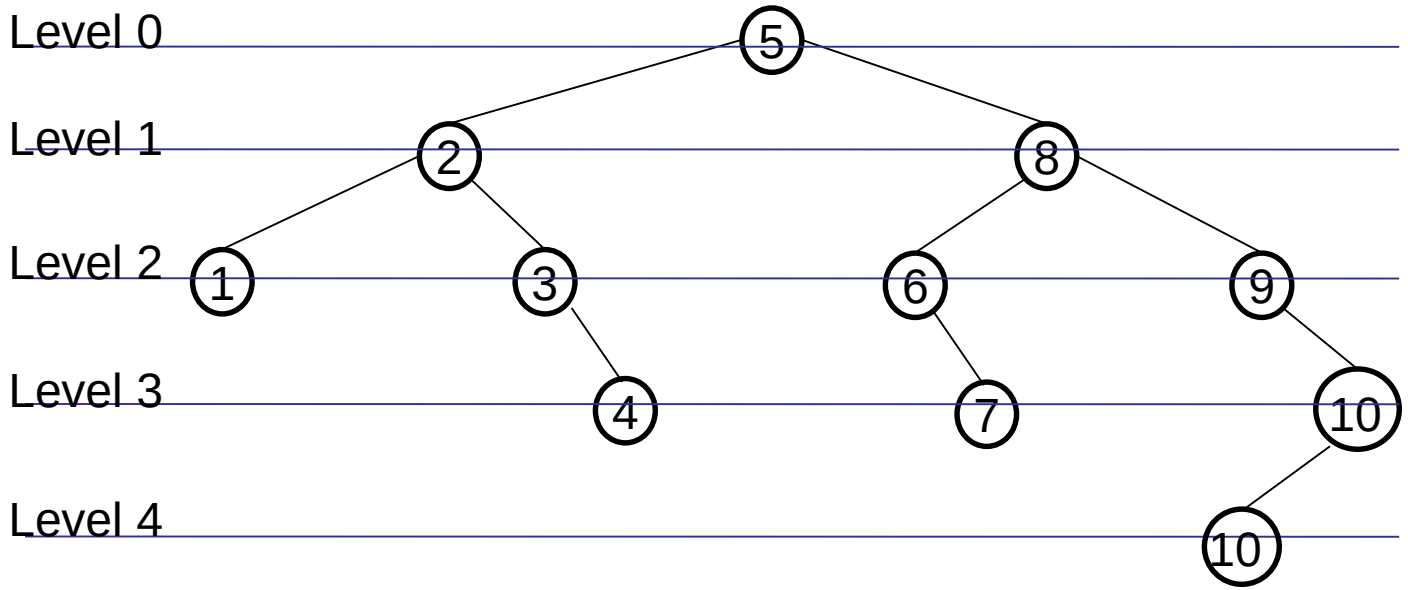
Another important connection between algorithms and data structures is this. **Binary Search is suitable only for the contiguous implementation**. However, if we have to place the data in **a linked structures** (linked list) and be able in the same time to fasten the search what should we do?



Basic definitions of Binary Trees

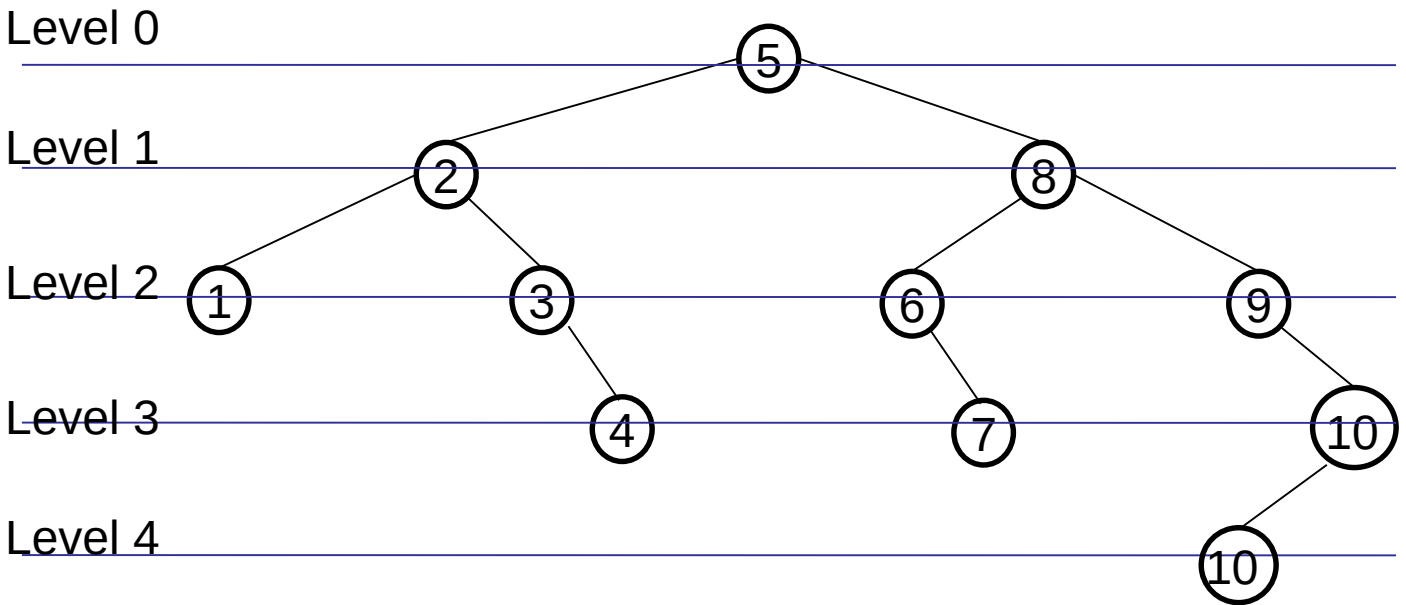
Definition: A binary tree is either empty, or it consists of a **node** (**vertex**) called the **root** together with two **binary trees** called the left subtree and the right subtree of the root.

- The only node at level 0 is the root.
- A node may have up to two children in the next level.
- The children of a node are joined to their parents by links called edges.
- There are Multi-way trees (degree higher than two) which will not be taught in our introductory course.



Definition:

- The **depth** (or **height**) of a node in the tree is the node's distance from the root; i.e., its level.
- **Equivalent Definition (recursive):** The depth of a node is 1 + the depth of its parent. And the root is at depth 0.
- The collection of nodes in the subtree rooted at a node (excluding the node itself) is referred to as its **descendants**. E.g., descendants of 8 are: 6, 7, 9, 10, 10.
- The collection of nodes encountered by climbing down (to the root) a path from a node to the root of the whole tree is the node's **ancestors** (or **predecessors**). E.g., ancestors of 3 are: 2, 5.

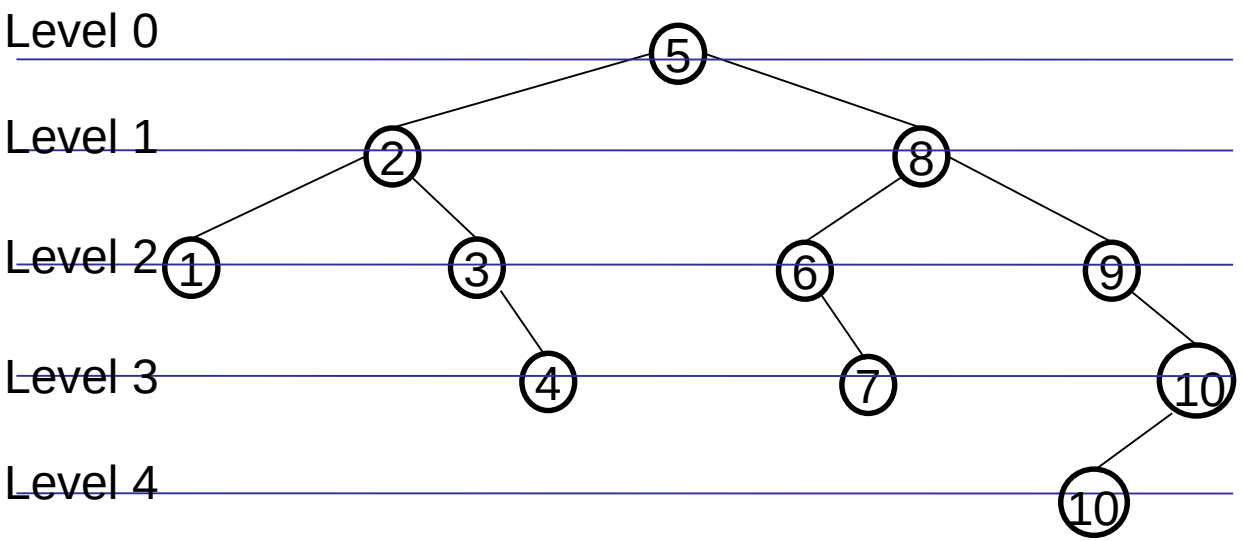


- **The depth or height h of a tree:** is the maximum height among nodes. Also, we can say

$$h = \begin{cases} 0 & \text{tree has only root} \\ 1 + \max(h(\text{left tree}), h(\text{right tree})) \end{cases}$$

- **Outdegree:** is the number of edges coming out of a node. In binary trees this is at most 2.

- **Indegree:** is the number of edges coming to a node. In any tree this is 1.



Before ADT and coding, How to traverse a BT? More motivations and benefits

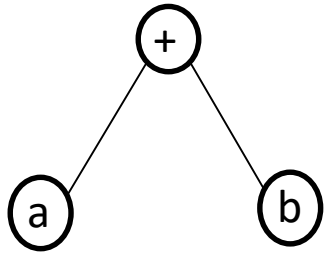
There was no problem for traversing a linear structure like a list. For a BT, at each node V having left and right subtrees L and R (respectively) we can do the following visiting:

VLR	VRL	LVR	LRV	RVL	RLV
Preorder		Inorder	Postorder		

The standards are Pre, In, and Post order. In the three of them, L precedes R; then V is before them (Pre) or in between (In) or after them (Post).

Example:

Preorder: +ab
Inorder: a+b
Postorder: ab+

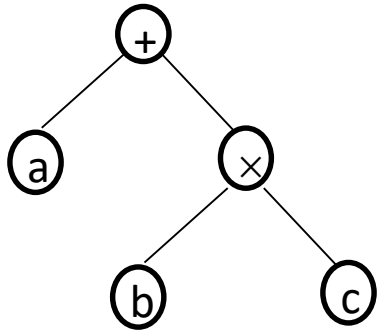


a + b

The *Polish Forms* (Ch 12) are related to these orders.
Preorder → *Prefix*
Inorder → *Infix*
Postorder → *Postfix*

Example:

Preorder: +a×bc
Inorder: a+b×c
Postorder: abc×+



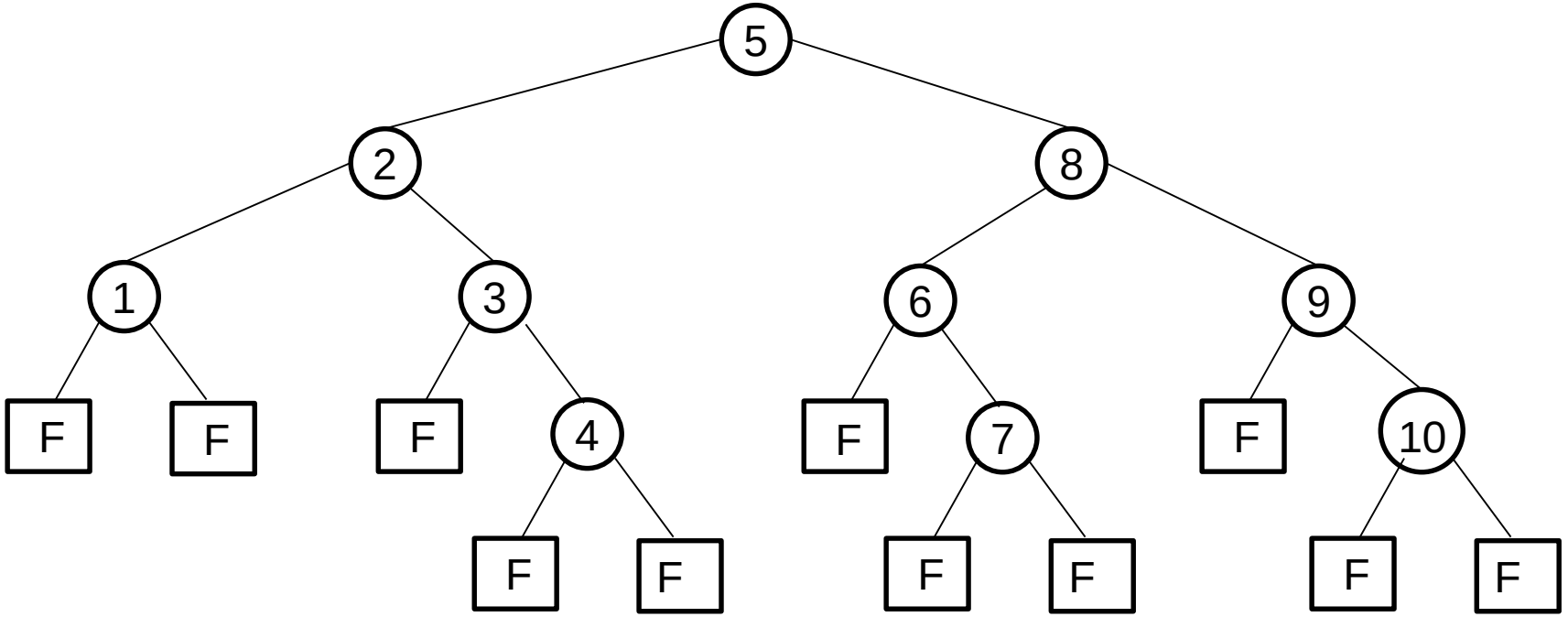
a + b × c

Also notice the strong connection between these traversal modes on a hand and recursion and stacks on the other hand.

Before ADT and coding, How to traverse a BT? More motivations and benefits

Notice that:

For a BT produced by binary search, the Inorder traversal produces sorted elements.



Now, we have enough motivation for having BTs as data structures. Let us define the ADT and start coding as linked implementation.

Definition: A ***Binary Tree ADT*** *is* either empty, or it consists of a **node** (**vertex**) called the root together with two binary trees called the left subtree and the right subtree of the root. This is together with the following operations:

1. **Create** the tree, leaving it empty.
2. Determine whether the tree is **empty** or not
3. Determine whether the tree is **full** or not
4. Find the **size** of the tree.
5. **Traverse** the tree, visiting each entry
6. **Clear** the tree to make it empty

We will define the next operations for a special type of **BT**, i.e., **Binary Search Trees** (which will be defined later).

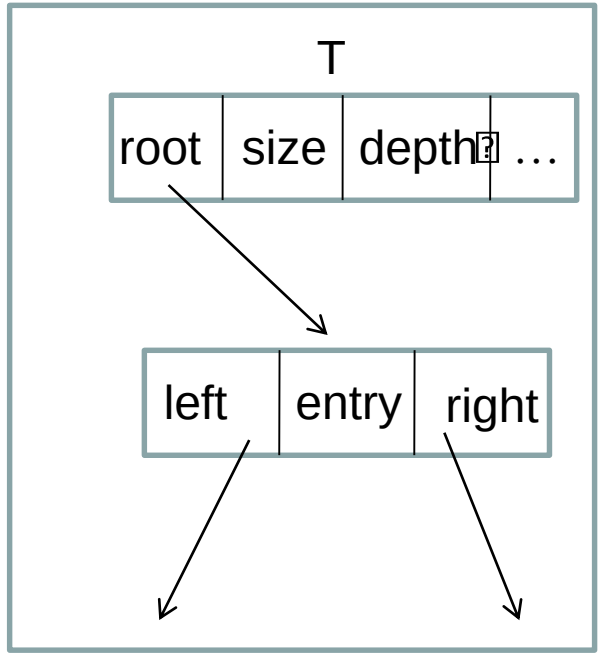
1. Insert a new entry (we have to define where)
2. Delete an entry (we have to define from where)
3. Search for an element.
4. (Any other operation to be defined later).

Linked Implementation

```
Struct TreeNode{
        TreeEntry entry;
        struct TreeNode *left, *right;
};

struct Tree{
        TreeNode *    root;
        int           size;
        int           depth;
       //Other fields are possible if needed
};


void    CreateTree     (Tree *);
void    ClearTree      (Tree *);
int     TreeSize       (Tree *);
Int     TreeDepth      (Tree *);
void    Preorder       (Tree *);
void    Inorder        (Tree *);
void    Postorder      (Tree *);
```



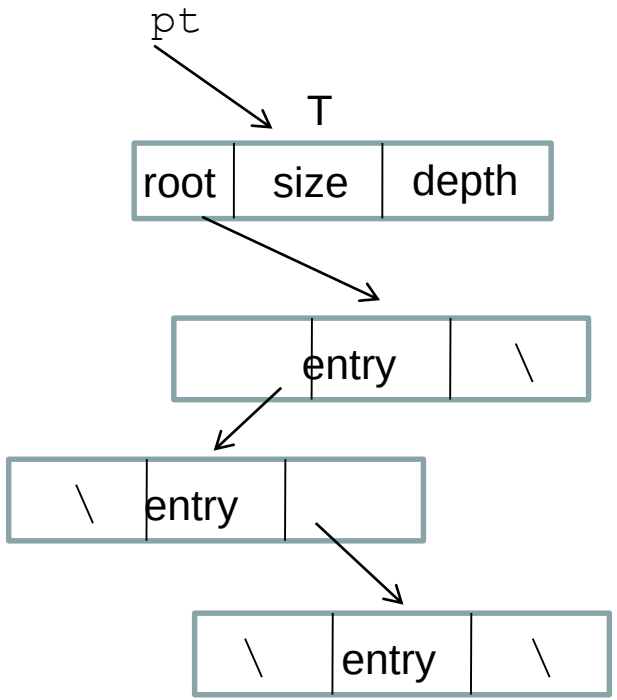
Notice that: the book defined the tree to be a pointer to a node directly rather than a `struct` that contains a pointer (along with other fields) to a node. The book itself did not agree with that style before (see page 86).
Study both implementations.

```
void CreateTree(Tree *pt){
        pt->root=NULL;
        pt->depth=0;
        pt->size=0;
}

int TreeEmpty(Tree *pt){
        return (!pt->root);
}

int TreeFull(Tree *pt){
        return 0;
}
```

```
//User level
Tree t;
CreateTree(&t);
```

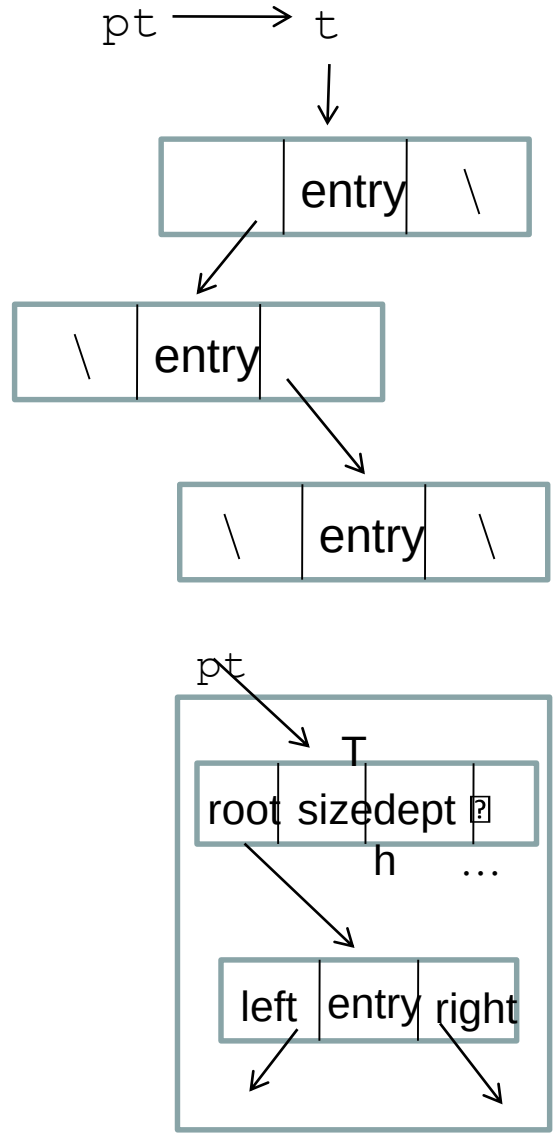


```
void InorderRec(Tree *pt){
    InorderRec_aux(pt->root);
}
```

```
//Pre: Tree has been created and
intialized.
//Post: Inorder traversal.
void InorderRec_aux(TreeNode *pt){
    if (*pt){
       InorderRec_node(pt->left);
       cout<<pt->entry<<" ";
       InorderRec_node(pt->right);
    }
}
```

Notice that, we pass a pointer to the tree to
be consistent with other functions.

```
//User level
Tree t;
InorderRec(&t)
```



```
void ClearTreeRec(Tree *pt){
    ClearTreeRec_aux(pt->root);
    pt->size=0;
    pt->depth=0;
}
```

```
void ClearTreeRec_aux(TreeNode *pt){
      if (pt){
            ClearTreeRec_node(pt->left);
            ClearTreeRec_node(pt->right);
            delete pt;
            pt=NULL;
      }
}
```

```
//User level
Tree t;
ClearTreeRec(&t);
```

Binary Search Trees (BST)

Motivation: To be able to search efficiently in lower complexity than the linked lists.

Definition: A ***Binary Search Tree ADT*** *is* either empty, or it consists of a **node** (**vertex**) called the root together with two binary trees called the left subtree and the right subtree of the root. Every node contains a key and satisfies the following conditions:

- The key in the left child of a node (if it exists) is less than the key in its parent
- The key in the right of a node (if it exists) is greater than the key in its parent node.

This is together with the following operations:

1. Insert a new entry (we have to define where)
2. Delete an entry (we have to define from where)
3. Search for an element.
4. (Any other operation to be defined later).

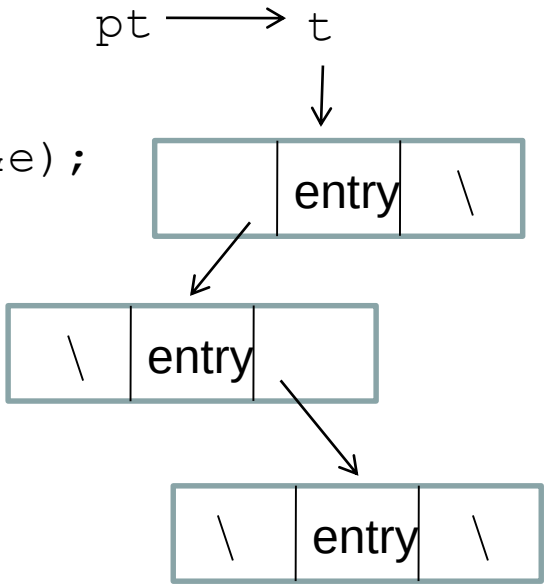
Note:

- The red statement is the only addition to the definition of the Binary Trees.
- The `typedef` part will be exactly the same as the binary trees, along with all the previous functions (since they were common for any binary tree).
- We can simply use: `typedef ListEntry TreeEntry;`

```
//Pre: Tree has been created and intialized.
void InsertTreeRec_aux(TreeNode *pt, TreeEntry *pe)
{
    if (!pt)
    {
        pt = new TreeNode;
        pt->entry=*pe;
        pt->left=NULL;
        pt->right=NULL;
    }
    else if (*pe < pt->entry)
        InsertTreeRec_aux(pt->left, pe);
    else
        InsertTreeRec_aux(pt->right, pe);
}
```

```
//User level
Tree t;
TreeEntry e;

InsertTreeRec(&t, &e);
```

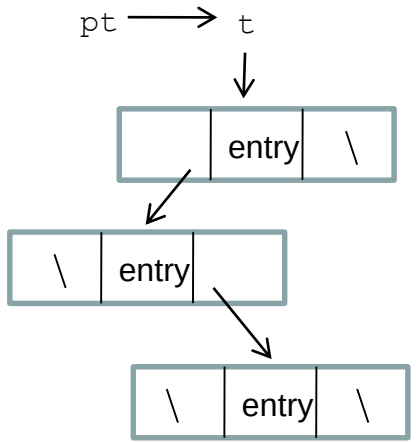


Important issues:

- pe is pointer to element rather than the element itself to save space in recursion.
- new code is inside the if statement not at the beginning of the function for same reason.
- Equal keys are inserted to the right. Therefore, Inorder traversal will visit duplicate keys in the same order of insertion.

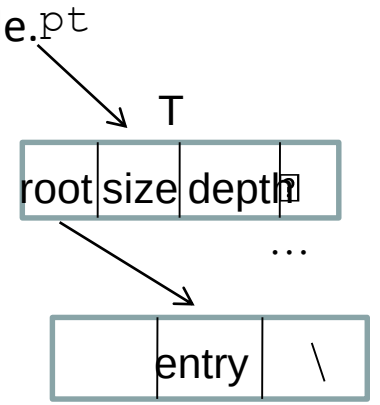
-**Now, how to write this function for Tree2? (We need to account for size and depth!); We need a similar function that returns at which depth the node is inserted.**

```
void InsertTree_aux(TreeNode *pt, TreeEntry *pe, int *pdepth){
       if (!pt)
        {
            pt = new TreeNode;
            pt->entry=*pe;
            pt->left=NULL;
            pt->right=NULL;
        }
       else if (*pe < pt->entry)
            InsertTreeRec_aux(pt->left, pe, pdepth);
       else
            InsertTreeRec_aux(pt->right, pe, pdepth);
      (*pdepth)++;//This is the only difference from previous
}                 //of course, this is not provided for the user.
```



Note: We pass a pointer to int, i.e., `pdepth`, so that every recursive call increments the same variable.

```
void InsertTreeRec(Tree *pt, TreeEntry *pe){
       int d=0;
       InsertTree_aux(pt->root, pe, &d);
       if (pt->depth<d)
           pt->depth=d;
       pt->size++;
}
```



Tasks:

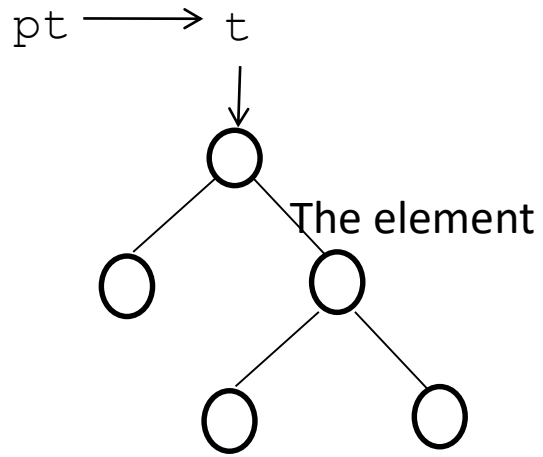
```
int TreeSizeRec(Tree *pt);

int TreeDepthRec(Tree *pt);
```

recursive to iterative and vice versa

- Insert
- Inorder
- Find

```
//Pre: Tree has been created and initialized (can be empty)
int FindItemTreeRec_aux(Tree *pt, KeyType k){
   if (!*pt)
      return 0;
   if ( pt->entry == k){
       return 1;
   }
   if (k < pt->entry)
      return FindItemTreeRec(pt->left, k);
   else
      return FindItemTreeRec(pt->right, k);
}//Very similar to Insert
```




```
int FindItemTreeRec(Tree *pt, KeyType k){
   return FindItemTreeRec_aux(pt->root, k);
}
```