Tree Data structure

Different arrangements of Data

- Array
- Linked list
- Stack
- Queue
- Priority queue

University

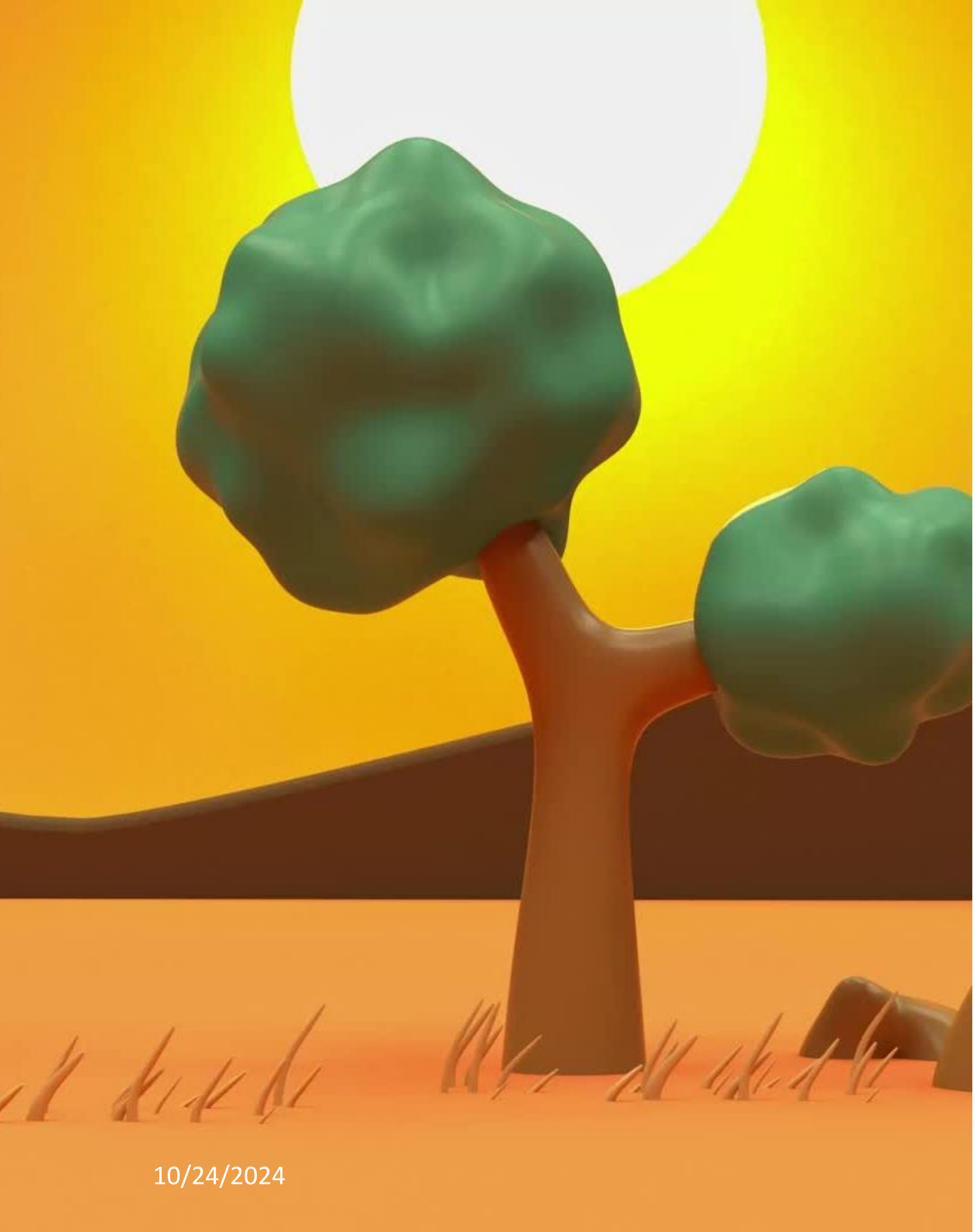
1. Board of management
2. Executive leadership : Chancellor, Vice Chancellor, Pro-Vice Chancellors
3. Academic affairs:
   I. Dean of Faculties
   II. Department Heads
   III. Professors
   IV. Associate Professors

4. Student affairs

Introducing Tree Data structure (non-linear)

A hierarchical relationship defined among data elements to make the search/update faster

Data items are stored in such a way that with every search a fraction of dataset gets reduced, or search space keeps reducing making the search more efficient



Tree Definition

- A tree is a finite set of elements that is either empty or partitioned into ***three or more disjoint sets***
- The first subset contains one element and is called ***root of the tree***
- The other subsets are themselves ***tree*** and are called the ***subtrees*** of original tree
- Each element of the tree is called a ***node***
- ***This node*** is different from the node of a singly linked list in the sense that it always contains ***more than one address field.***

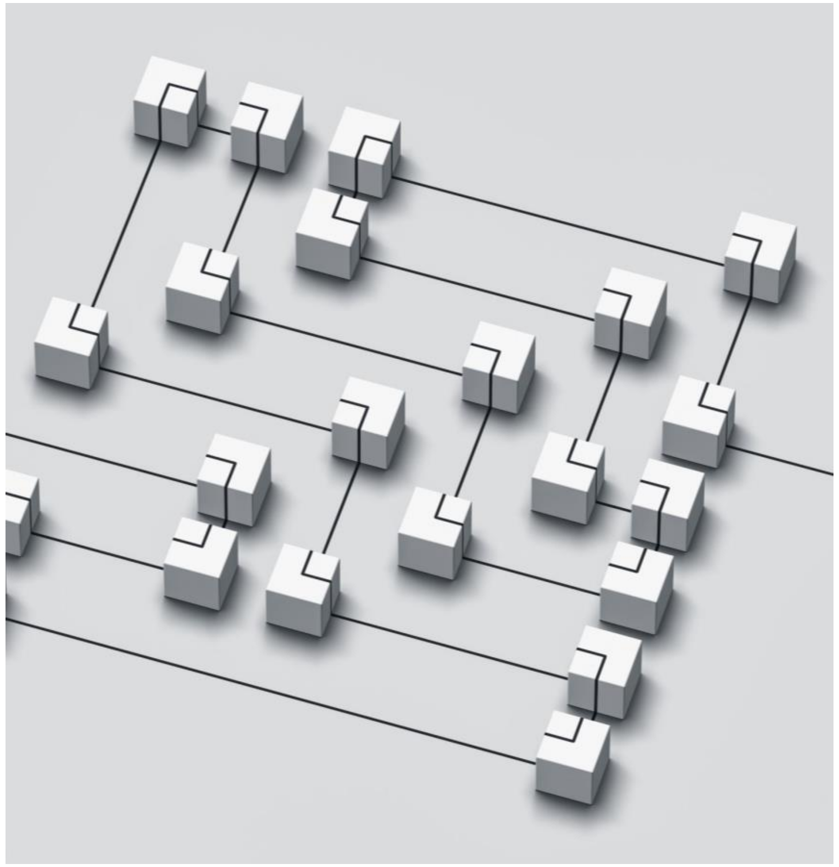
Difference between linear and non-linear DS

Linear Data Structure:

- Data elements are arranged in a sequential way where every element is attached to its previous and next adjacent.
- All the data is on the same level and data elements can be traversed in a single run only.
- Its implementation is easy in comparison to non-linear data structure.
- But mostly memory is not utilized in an efficient way.
- Array, stack, queue, linked list are linear.
- Applications are mostly in application software development.

Non-linear Data Structure

- **Data elements are attached in a hierarchical manner.**
- **Multiple levels are involved.**
- **Implementation is complex in comparison to linear.**
- **Data elements can't be traversed in a single run only.**
- **Memory is utilized in an efficient way. Exp: trees and graphs.**
- **Applications of non-linear DS are in AI and image processing**
- **Non-linear data structures can represent complex relationships and data hierarchies easily, such as in social networks, file systems, or computer networks.**
- **Performance can vary depending on the structure and the implementation.**



Binary Tree

- A binary tree is a finite set of elements that is either empty or partitioned into **three disjoint sets**
- The first subset contains single element and is called **root of the tree**
- The other subsets are themselves **binary tree** and are called ***left subtree and right subtree of original tree***
- Each element of the binary tree is called a node having two address fields.

Another definition

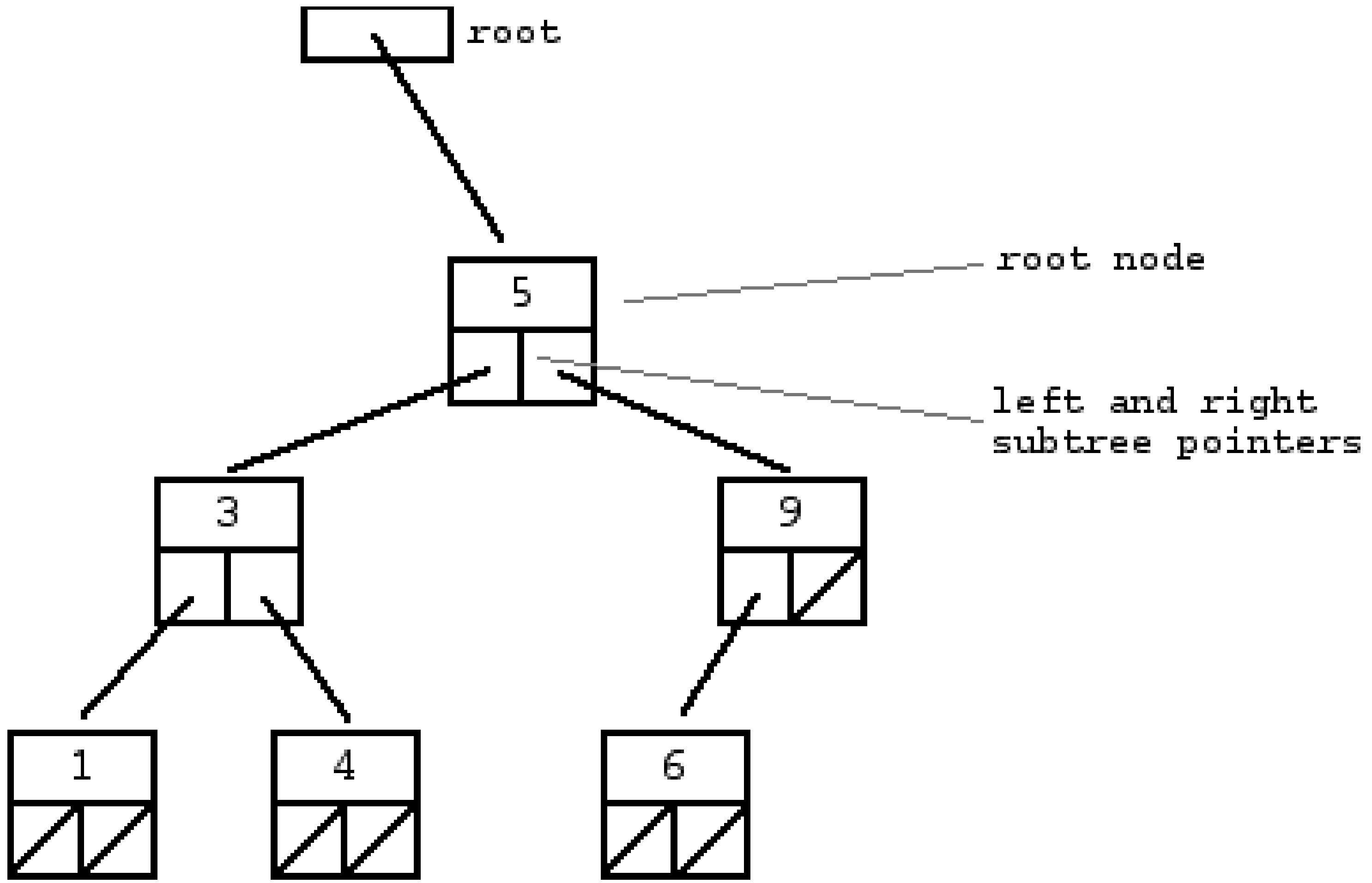
A binary tree is a hierarchical structure of nodes, where each node contains a "left" pointer, a "right" pointer, and a data element.

The "root" pointer points to the topmost node in the tree.

The left and right pointers recursively point to smaller "subtrees" on either side.

A null pointer represents a binary tree with no elements -- the empty tree.

Binary tree (Pictorial rep)



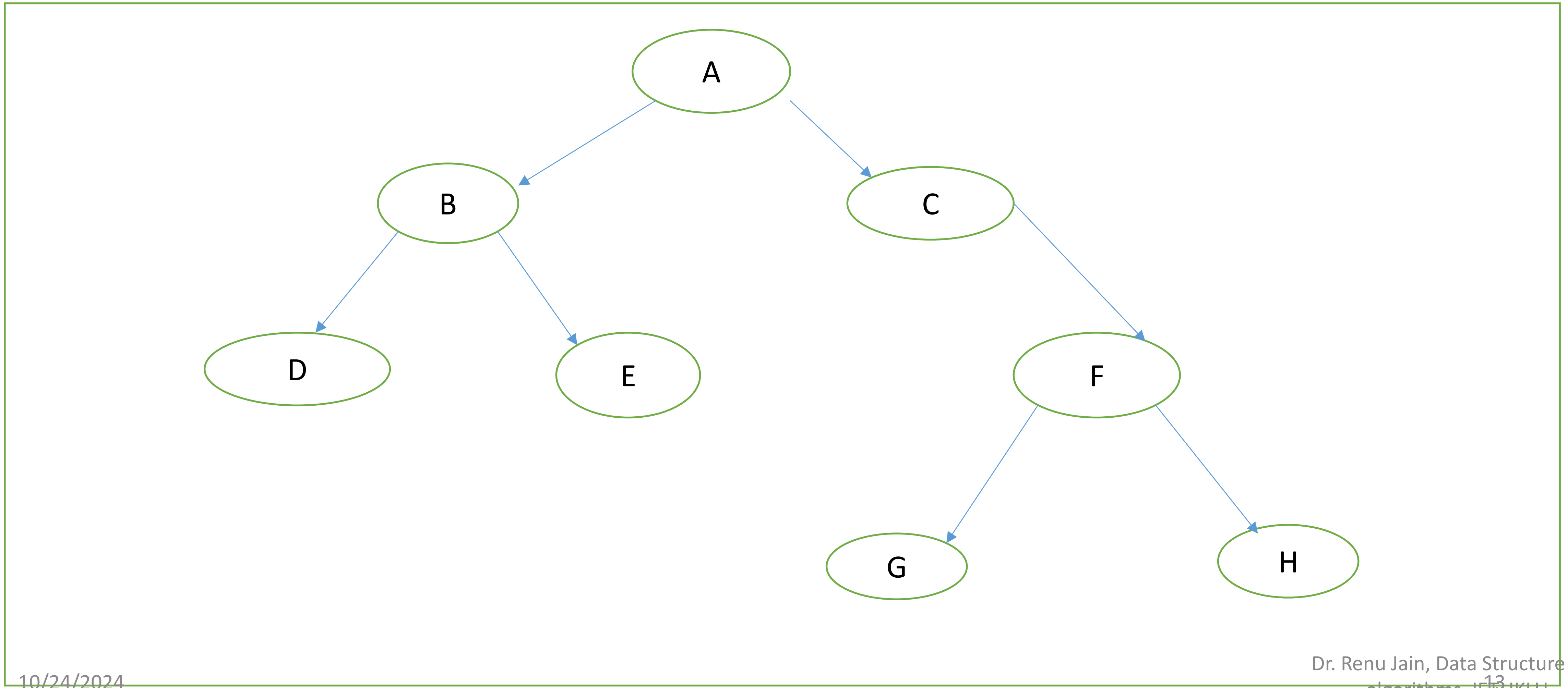
Binary Tree: Things to note

- In DS, **the root of a tree is at the top of the tree** in contrary to our real-life tree.
- In a tree, there is always a unique path from the root of a tree to every other node in a tree
- This has an important consequence: there are no cycles in a tree (think of a cycle as a closed path that allows us to go in a cycle infinitely many times).
- ***Any node within a tree can be viewed as a root of its own subtree -just take any node, cut of the branch that connects above to the rest of a tree, and it becomes a root with a smaller (may be empty also) tree of its own.***

Tree Terminology

- The node at the end of each path that leads from the root downwards is called the **leaf node**.
- ***Leaves*** are the nodes that point to null. In trees we have many such nodes while in a linked list, there was only one such node indicating the end of the list.
- Given a node in a tree, its ***successors*** (nodes connected to it in a level below) are called its **children**.
- ***Descendants*** of a node are its children, and the children of its children, and the children of the children of its children, and ....
- Given a node in a tree, its **predecessor** (node that connects to it in a level above - there is only one such node) is called its parent.
- ***Ascendants*** of a node are its parent, and the parent of the parent, and ... - all the nodes along the path from itself to the root.

Binary Tree



Tree Terminology……

- **Level** of a node refers to the distance of a node from the root. **Root** is at a distance zero from itself, so the level of the root is 0.
- Children of the root are at level 1, ”***grandchildren***” or children of the children of the root are at level 2, etc.
- ***Height*** is defined as the largest level of any node.
- Given N nodes, the ”shortest” binary tree that we can construct has ⌊log2N⌋+1 levels.
- Given N nodes, the ”tallest” binary tree that we can construct has N−1 levels. Why?

Types of Binary Tree

- ***Strictly binary tree***: If every non-leaf node in a binary tree has non-empty left and right subtree, the tree is called strictly binary tree.
- A strictly binary tree with n leaves has 2n-1 nodes

- ***Complete binary tree:*** A complete binary tree of depth d or height d is the strictly binary tree, all of whose leaves are at level d***.***
- ***No of nodes in Complete binary tree =***$2^0 + 2^1 + \ldots\ldots 2^d$

(Also called Perfect binary tree)

Almost Complete binary tree

In an almost complete binary tree, all the levels of a tree are filled entirely except the last level.

In the last level, nodes might or might not be filled fully, but all the nodes should be filled from the left to right.

Cont..

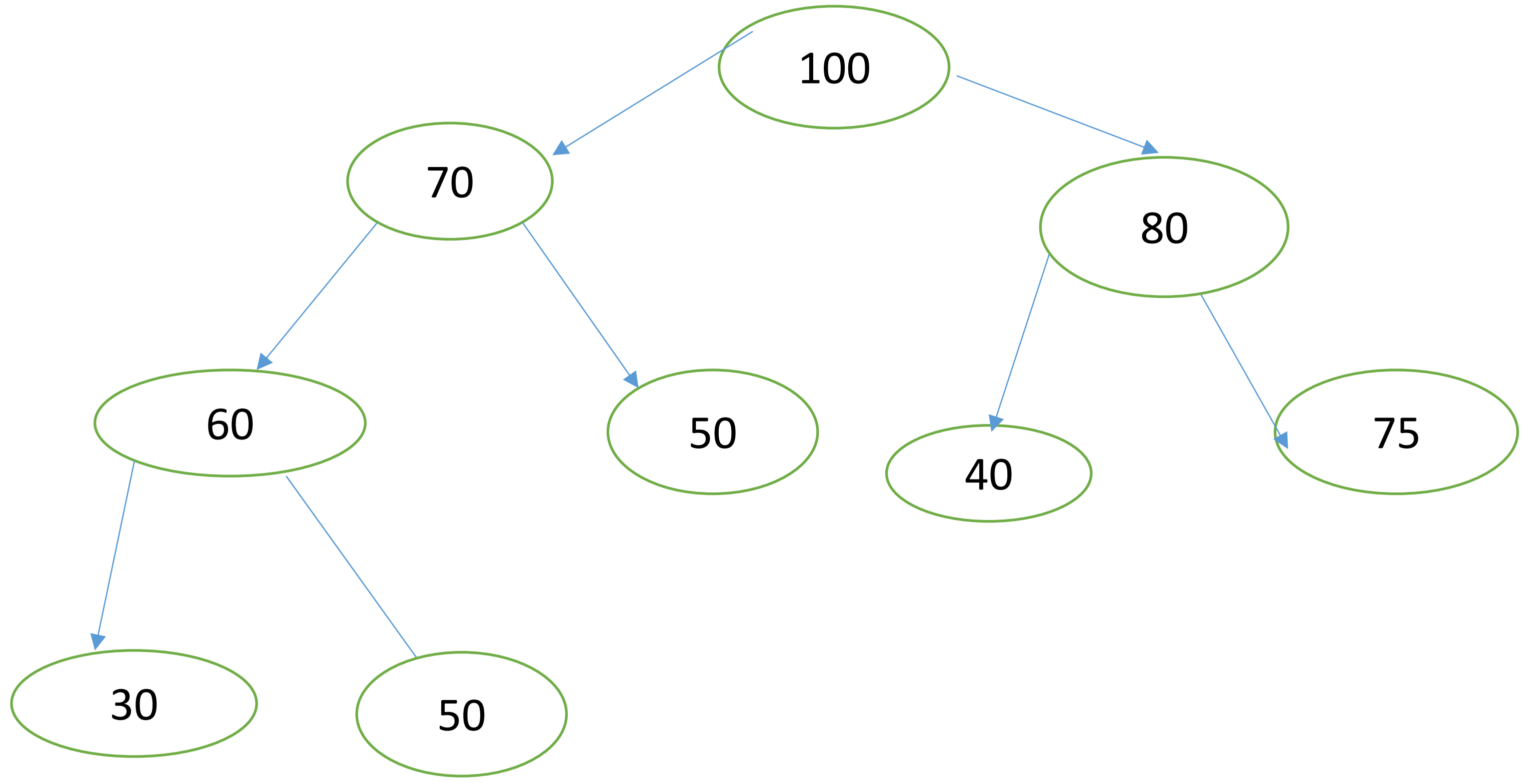
- **Almost complete binary tree**: A binary tree of depth ‘d’ is an almost complete binary tree if

1. Any node ‘nd’ at level less than d-1 has two sons (this means if depth of tree is ‘d’ then any node less than d-1 should have two children)
2. For any node ‘nd’ in the tree with a right descendent at level ‘d’, nd must have a left son and every left descendent of nd is either a leaf at level ‘d’ or has two children. Or This can be broken into two parts:

a. For all the nodes in tree, Find the height d of a right descendent node

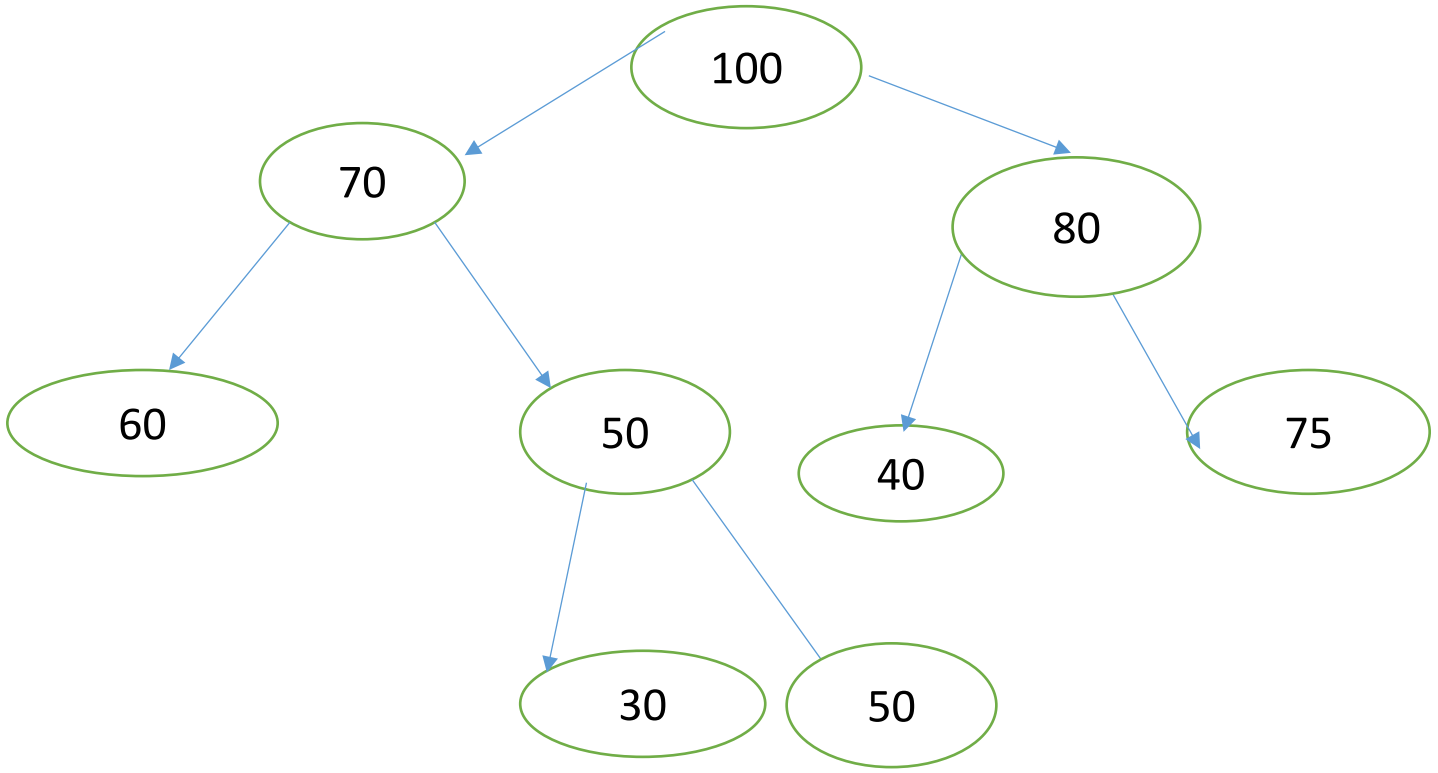
b. Now look at the left descendants one by one and if it is a leaf, it should be at height d and other nodes should have two children

Example(almost complete binary tree)



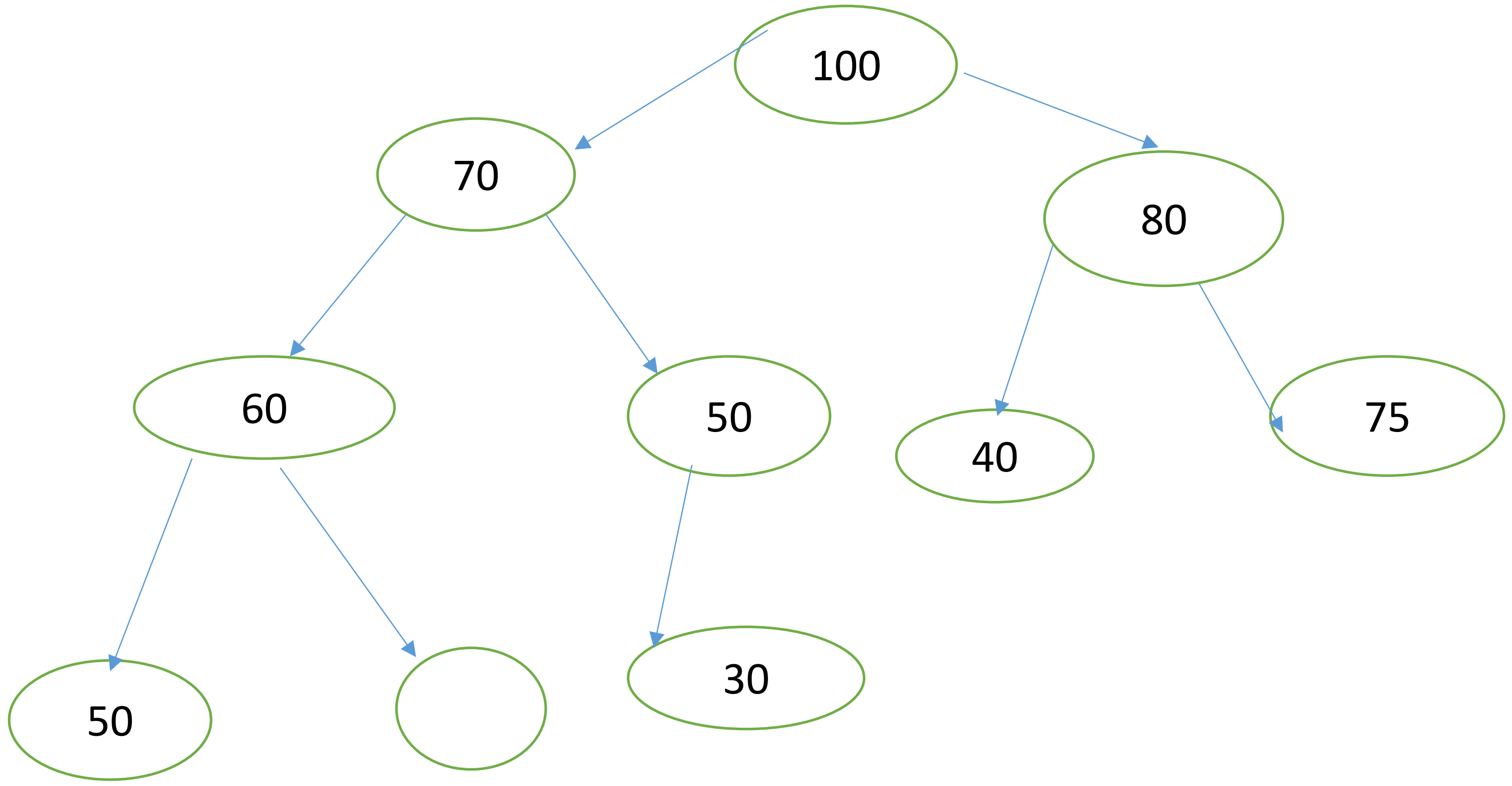
<u>yes</u>

Example(almost complete binary tree)



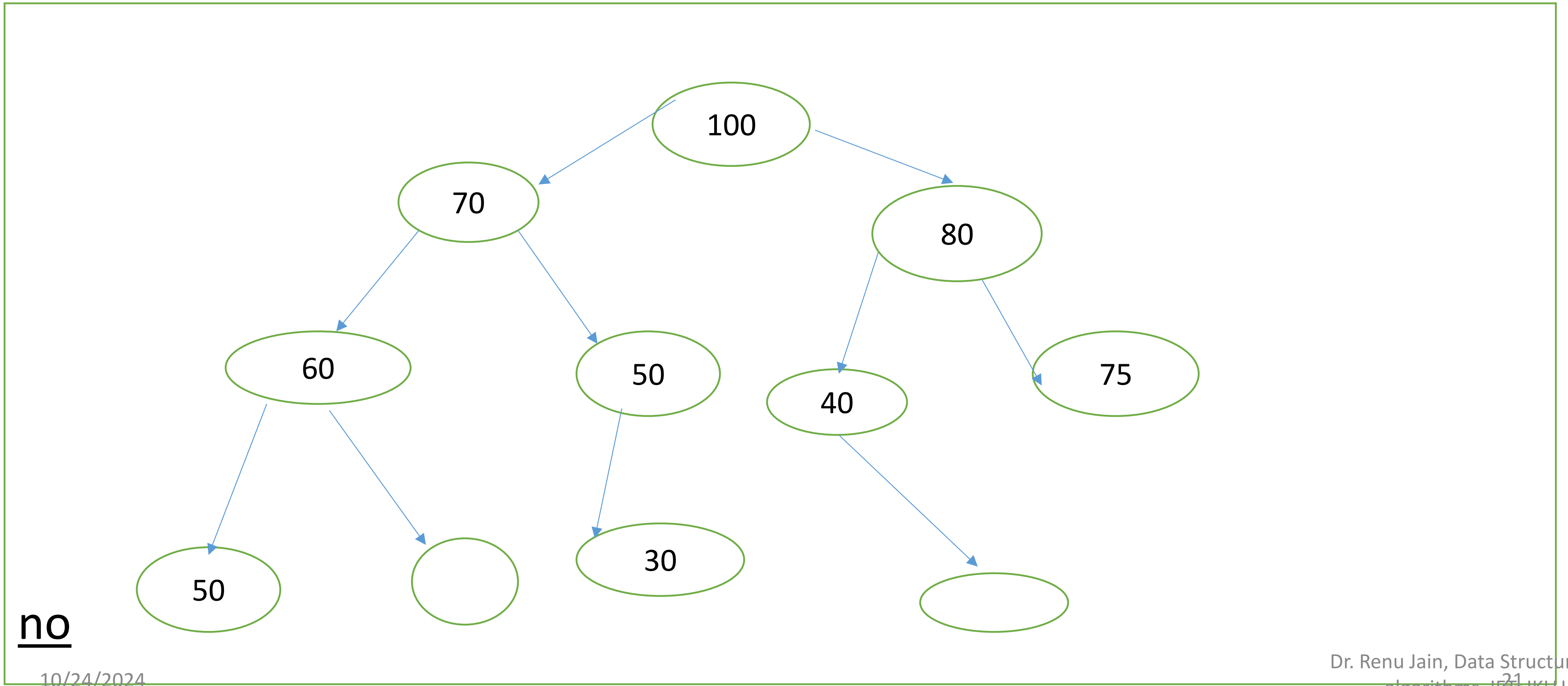
<u>no</u>

Example(almost complete binary tree)



<u>yes</u>

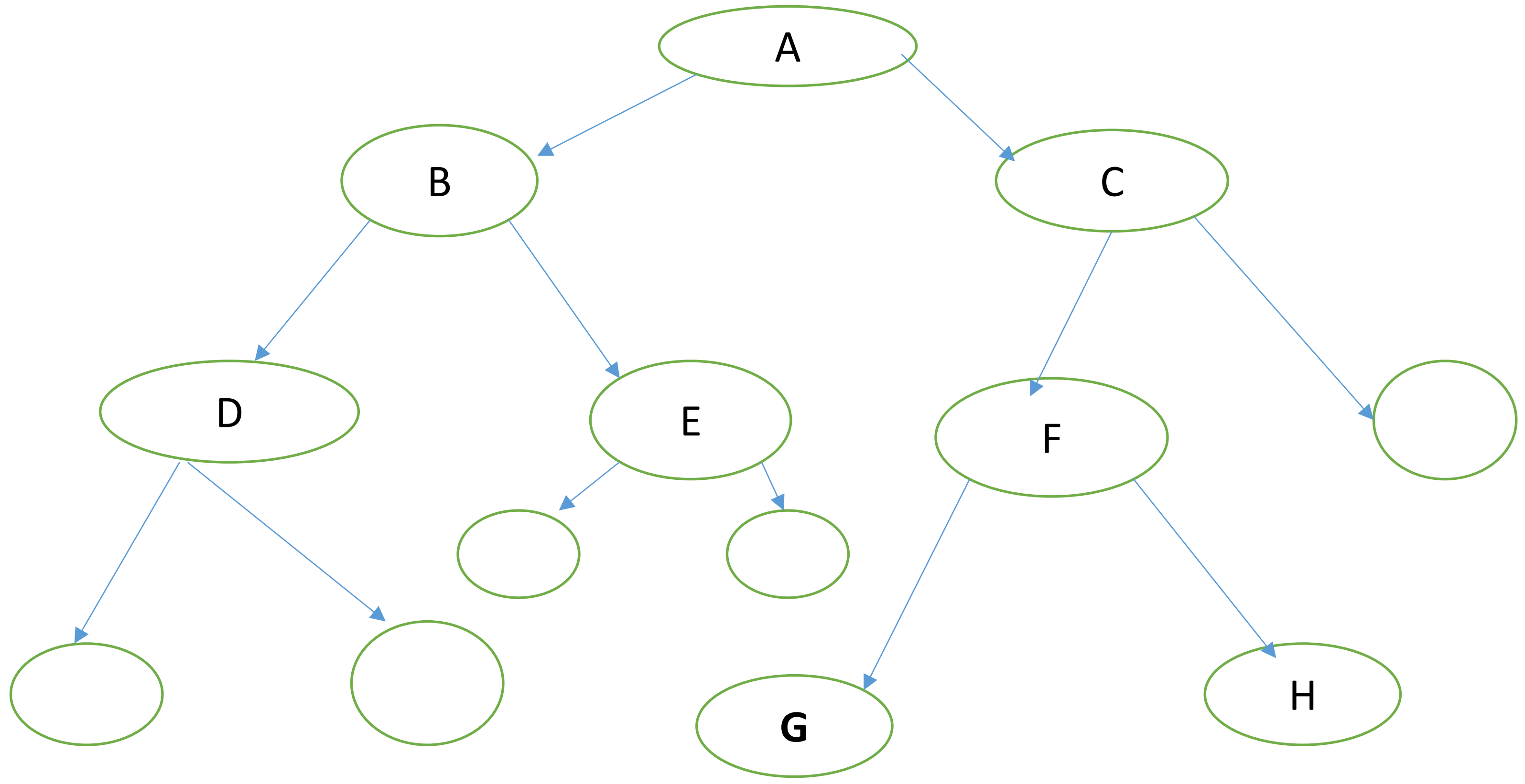
Example(almost complete binary tree)



<u>no</u>

Binary Tree

<ins>How to make a simple binary tree an almost complete binary tree</ins>



Why??????.. Tree structure

- What is the need to have a tree structure?
- What are the applications where this structure can be useful?
- Will this structure help us in thinking more logical and simple algorithms while solving problems?
- Does it reduce complexity?
- Is it easy to implement?
- What are the overall advantages and disadvantages?
- Let us discuss some applications…

- Second lecture

Binary tree representations and operations on binary trees

- We can perceive binary tree as a data structure in which:
- Data is stored in nodes and these nodes are connected to each other like a binary tree or there is some relation ship among the nodes which can be defined as a binary tree
- Address of root node is stored to recognize the tree
- Given address of root node, following operations may be required:
- Address of a particular node
- Contents of a node
- Children of a node (left or right child)
- Father of a node , etc.

Binary tree structure

```
Struct treenode  {int info;
                  struct treenode *left;
                   struct treenode *right;
                  }
OR
Struct treenode  {int info;
                   Struct treenode  *father;
                  struct treenode *left;
                   struct treenode *right;
                  }
```

Can we store binary tree in array?? Or can we implement binary tree using arrays??

Implicit array implementation of binary tree

- If we have **almost complete binary tree** or **complete binary tree**, then n nodes of the tree can be stored in an array of size n.
- Array index has values from 0 to n-1
- If we number the nodes from 0 to n-1, then two children of **node p** will be **2p+1 and 2p+2**
- Given a left child at **position p**, its right child will be **at p+1 position**
- Given a right child at position **p**, its left child will be at **p-1** position
- Father(p) = **p-1/2**
- Note: Here we have not stored any explicit link to children, all the links are implicit

Implicit array implementation of binary tree(Not possible for every tree)

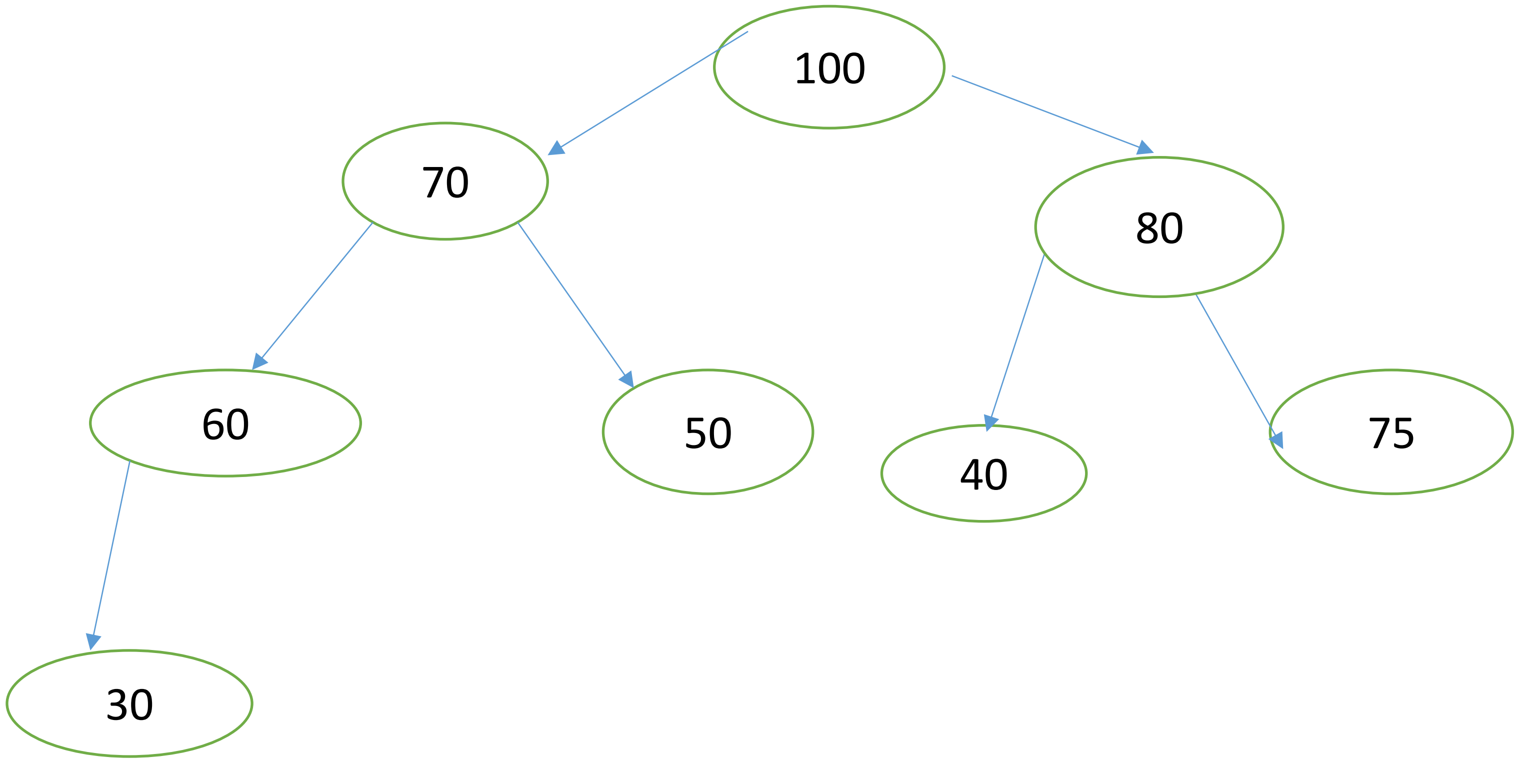
- If we want to implement a binary tree using arrays and if tree is not an almost complete binary tree or a complete binary tree, then
- we have to keep many blank array locations with some flag indicating that this node does not belong to the original tree.
- So, the number of array locations will be more than no. of actual nodes in the tree and many of them may be empty.
- Hence, if the tree is almost complete binary tree or complete binary tree, it is stored in an array otherwise linked representation is used and each node is created and linked dynamically as done in a linked list

Heap and Heap sort

- What is a heap??
- ***Descending heap or max heap***: Max heap of size n is an almost complete binary tree of n nodes such that the contents of each node are less than equal to its father
- So, root contains the maximum or largest element in the heap
- ***Ascending heap or min heap***: Min heap of size n is an almost complete binary tree of n nodes such that contents of each node are greater than equal to its father
- So, root contains the minimum or smallest element in the heap

Example

Suppose data is : 50, 40, 60, 100, 70, 80,75, 30 (Max heap)

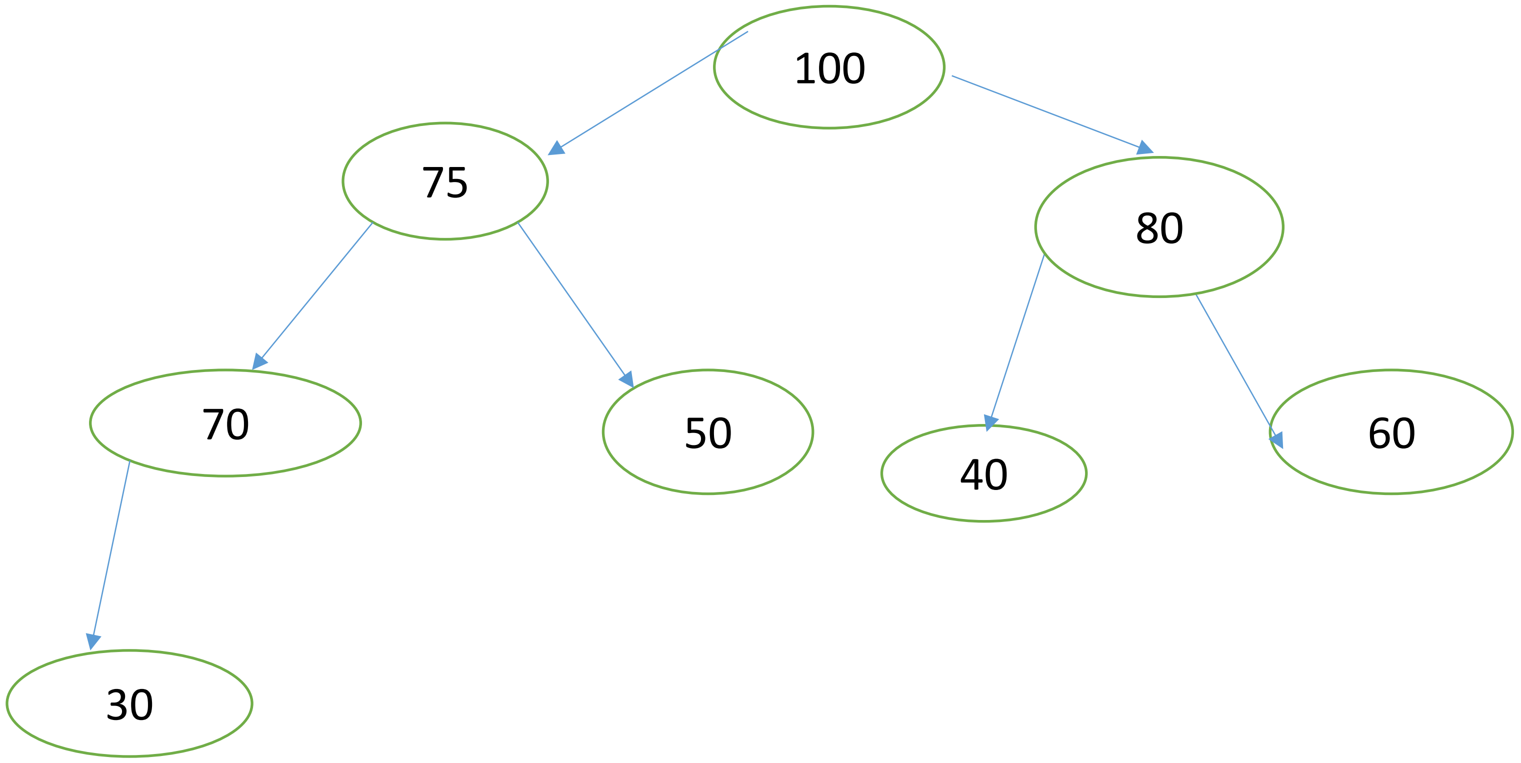


$$\begin{tikzcd}[column sep=small]
 & & & 100 \arrow[dll] \arrow[dr] & & \\
 & 70 \arrow[dl] \arrow[dr] & & & 80 \arrow[dl] \arrow[dr] & \\
60 \arrow[d] & & 50 & 40 & & 75 \\
30 & & & & &
\end{tikzcd}$$

Many such structures i.e. max heaps may be constructed from this data.

Example

Suppose data is : 50, 40, 60, 100, 70, 80,75, 30 (Max heap)



Why the concept of heap??

- For efficient implementation of priority queue (a queue which always deletes highest priority element from the queue)
- Possible implementations of priority queue are
- **Case1**: Store as an unsorted array or linked list

Complexities: insertion : **O(1)** and deletion **O(n)**

- **Case2**: Store as a sorted array or linked list

Complexities: deletion: **O(1)** and insertion **O(n)**

- **Case3**: Store as a descending heap
- Complexities: deletion: **O(logn)** and insertion **O(logn)**

How to construct heap??

- For each element..

1. Insert the element at the last empty position
2. Compare with its father
3. If father is less than child, swap father and child
4. Continue till father becomes greater than child or we have reached the root

Heap Insert function

```
Heap_insert(int dpq[],int k, int addval)
// k = no. of elements already entered in the array
{int i, father;
  i = k;
 father = (i-1)/2;
while(i>0 && DPQ[father] <addvalue)
{ DPQ[i] = DPQ[father];
   i = father;
father = (i-1)/2;
}
DPQ[i] = addval;
}
```

Heap delete function

```
int Heap_delete(int dpq[],int ksize)
/* ksize = no. of elements entered in the array */
{int pos;
pos = DPQ[0];
Heap_adjust(0,ksize-1);
Return(pos);
};
```

Heap Adjust steps

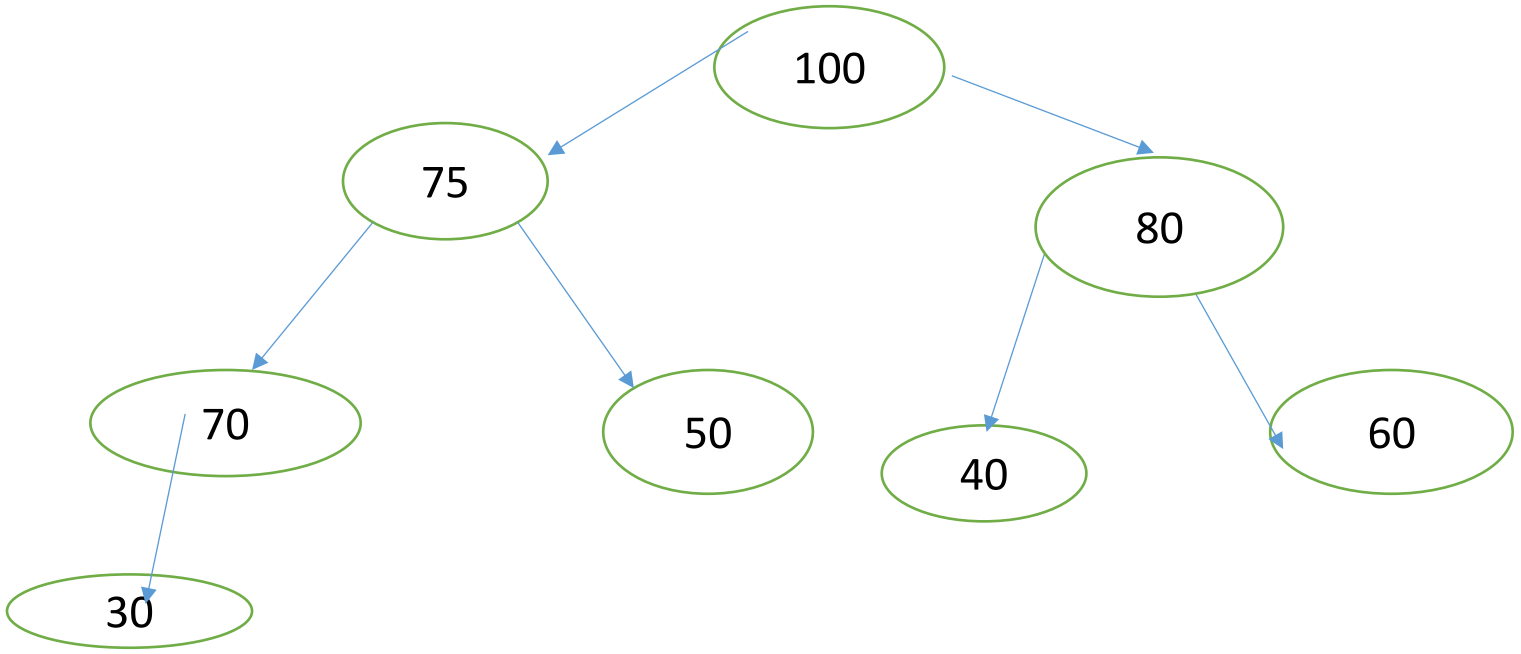
Goal: After deletion, root location is empty, and we need to rebuild a max heap for remaining elements.

Steps:

1. Store the value of last element of array in a temporary variable(***temp***) and reduce the size of array by 1
2. Starting from root, find the index of larger child(***child***)
3. Compare the ***temp*** and ***child*** and put the bigger value in root
4. ***Continue the process till temp gets proper place in the tree***

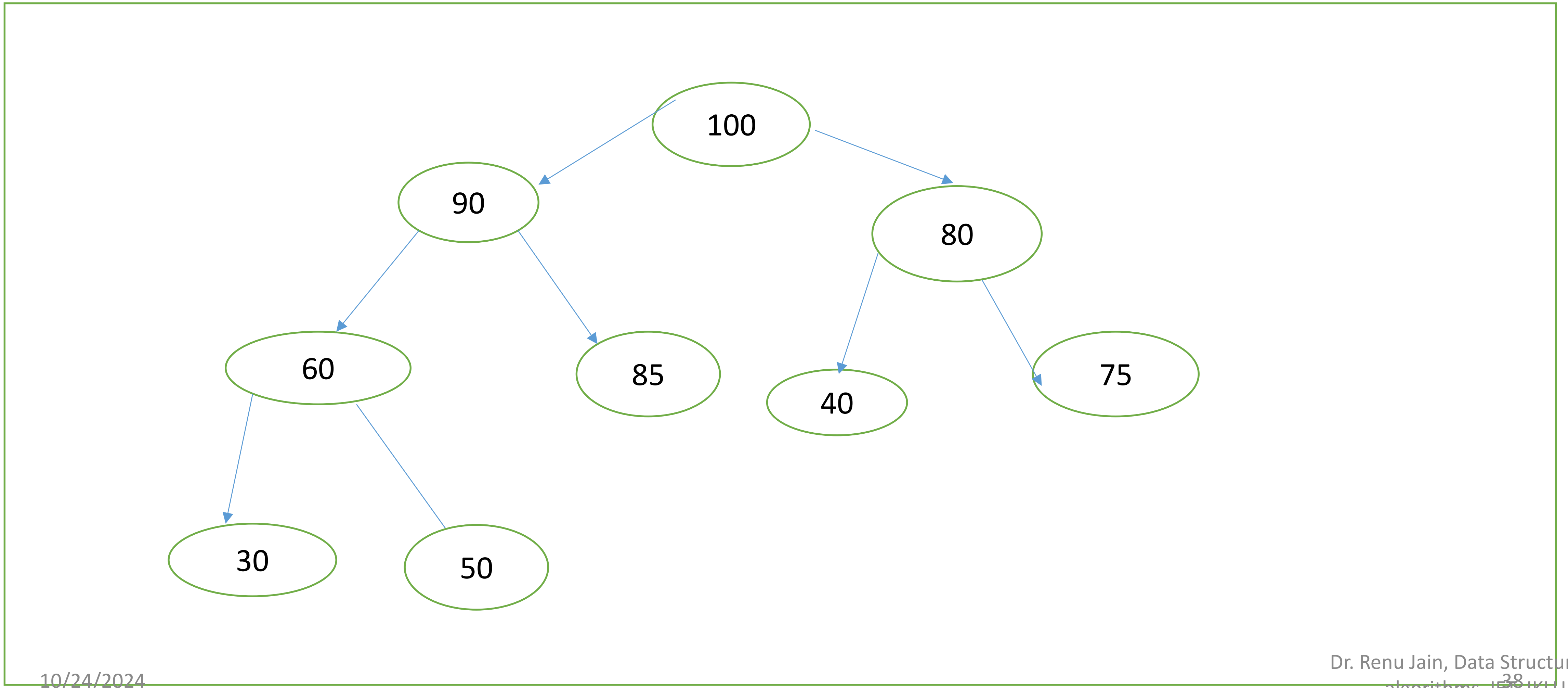
Example

Suppose data is : 50, 40, 60, 100, 70, 80,75, 30 (Max heap)



After deleting 100: 30 will be kept in temp, then 80 will moved to root, then 60 will be moved to position of 80, and 30 will be placed instead of 60

Exercise(deleting root node)



Heap delete function

```
Heap_adjust(int DPQ[], int root, int k)
{int father, child;
father = root; kvalue = DPQ[k];
child = larger_child(father, k-1)
while(child >=0 && kvalue <DPQ[child])
{ DPQ[father] = DPQ[child];
   father=child;
child = larger_child(father,k-1);/* index of larger child of father*, returns -1 if no child/
}
DPQ[father] = kvalue;
}
```

Efficient priority queue implementation

- Implement priority queue as a heap
- This means whenever a new data/client joins the queue, it will be added as if we are inserting the data in a heap
- So, the complexity of insertion is $O(\log_2 n)$
- whenever data is removed from priority queue, element of highest priority is removed
- If data is in a heap, root element is deleted and heap is adjusted
- So, the complexity of deletion is $O(\log_2 n)$

Heap sort

Steps:

1. Build descending heap reading one by one all the data elements of the array to be sorted (basically we are reorganizing the elements of the array)
2. Now delete data one by one and store the deleted data at the end of the array for that heap
3. We will have sorted array after 2n iteration
4. First n iterations of insertion and then n iterations od deletion
5. Total no. of insertions take n X ($\log_2 n$) steps and deletions take n X ($\log_2 n$) steps
6. Complexity of heap sort: O(2n $\log_2 n$)