CS 2124: DATA STRUCTURES
Spring 2024

- 7th Lecture
- Topics: **Introduction to Trees**

Topics

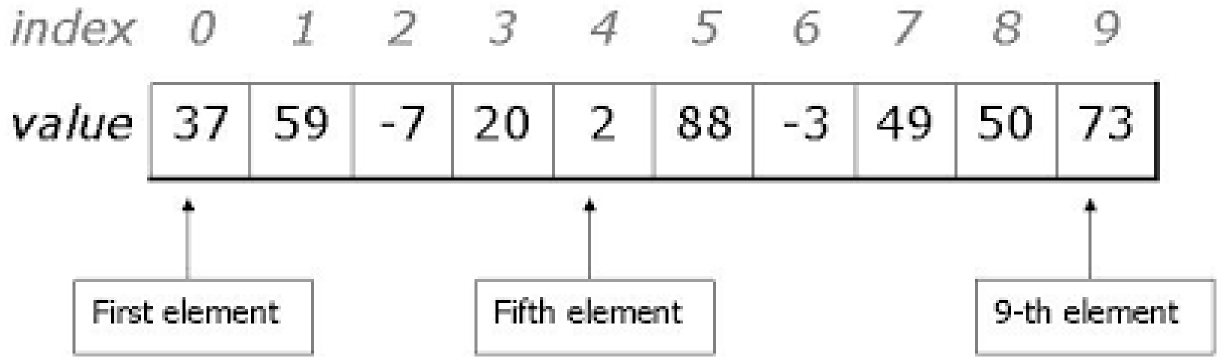
1. Introduction to Trees
   I. Binary Trees
      i. Types of Binary Trees
   II. Building A Binary Search Tree (BST)
      i. Insert into an empty BST
      ii. Duplicate Removal in BST
   III. Binary Tree Traversal
      i. Preorder Traversal
      ii. In order Traversal
      iii. Post order Traversal
2. Expressions as Trees
3. Building Trees
   I. Binary Trees: Dynamic Nodes
4. Traversal Implementation: Recursive
5. Traversal Implementation: Using Stacks
6. Applications

- Assignment:

1. No PDF file
2. A copy paste of output in their PDF file rather then screenshot.
3. Screenshot of entire screen rather then the code and output (like in lectures)
4. EXE file being submitted in zip file on Canvas

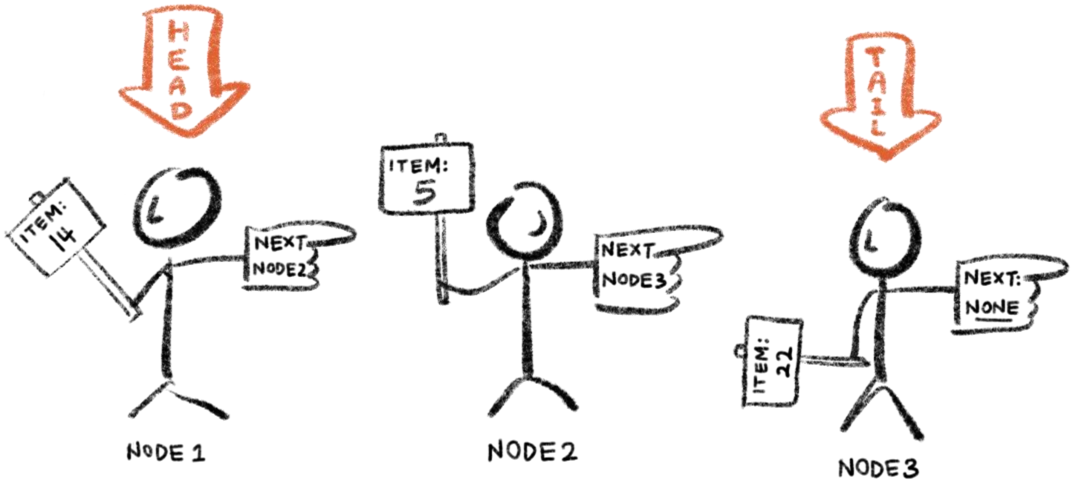
Introduction (What we have covered)

- There are many basic data structures that can be used to solve application problems.
- **Array** is a good static data structure that can be accessed randomly and is fairly easy to implement.
  - Insertion and deletion can be time consuming due to memory management
  - Array are not dynamic (i.e. The size of an array is determined at compile time)



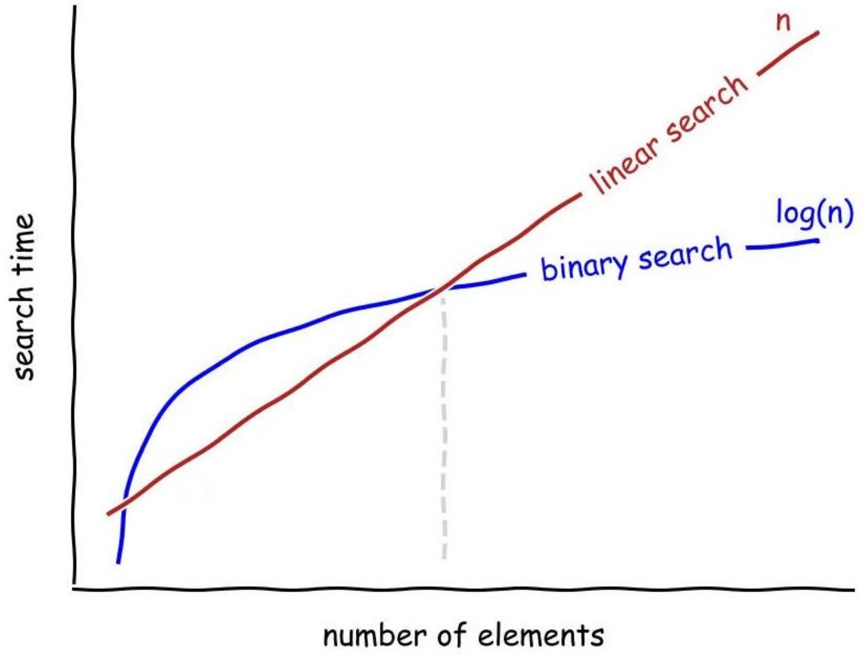
Introduction (What we have covered)

- **Linked Lists** on the other hand is dynamic and is ideal for application that requires frequent operations such as add, delete, and update.
  - One drawback of linked list is that data access is sequential.
- Then there are other specialized data structures like, stacks and queues that allows us to solve complicated problems using these restricted data structures.



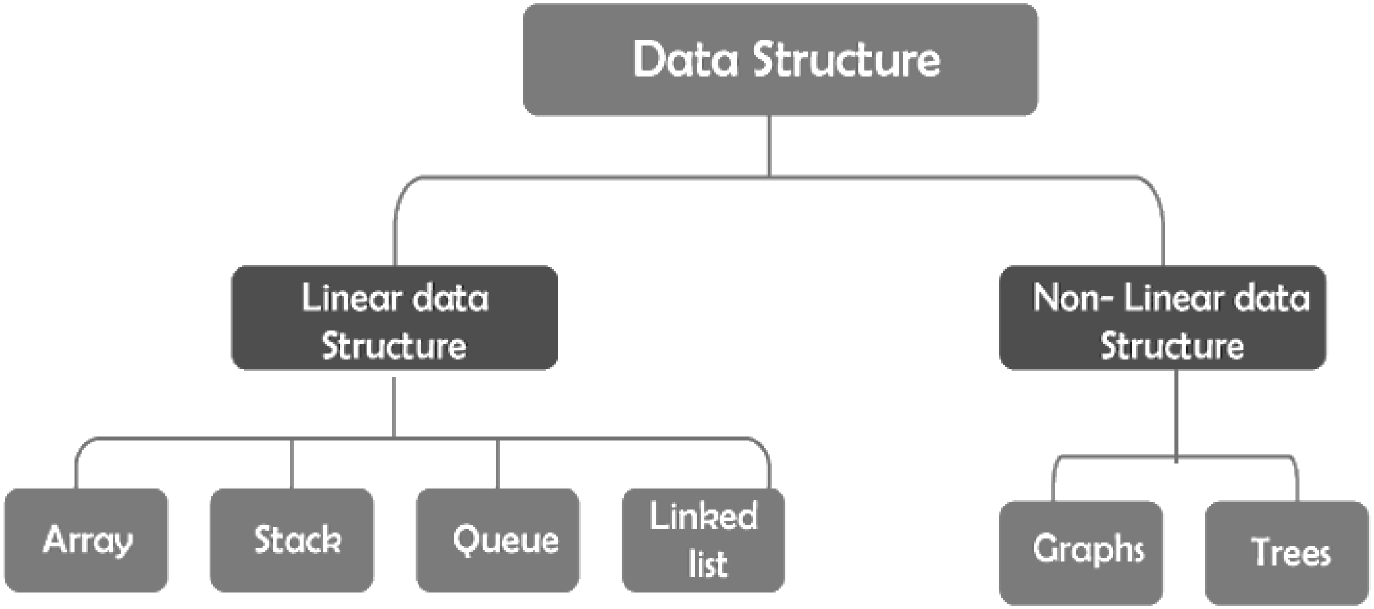
Introduction

- One of the disadvantages of using an array (unsorted) or linked list to store data is the time necessary to **search** for an item.
- Since both the arrays and Linked Lists are **linear structures** the time required to search a "linear" list is proportional to the size of the data set.
    - For example, if the size of the data set is ***n***, then the number of comparisons needed to find (or not find) an item may be as bad as some multiple of ***n***.



Introduction

- In this lecture lets Extend the concept of linked data structure (linked list, stack, queue) to a structure that may have multiple relations among its nodes.
- Such a structure is called a **tree**.
- A **tree** is a collection of nodes connected by directed (or undirected) edges.



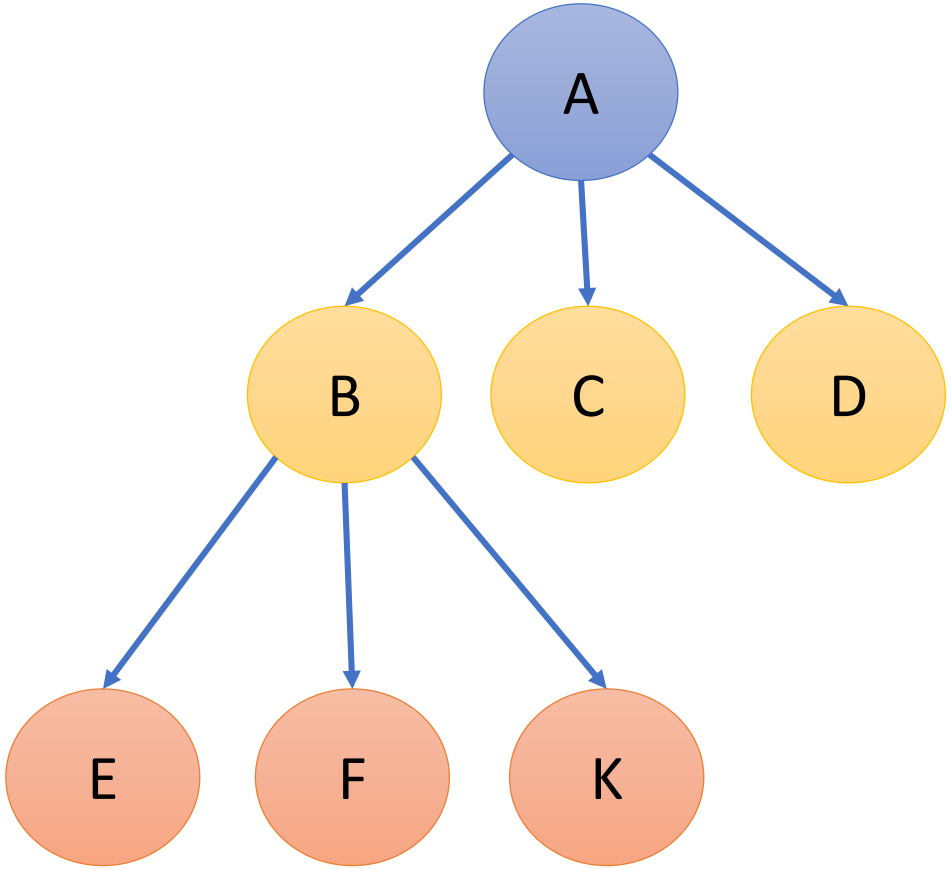
- Applications

1. Storing naturally hierarchical data
2. Database indexing
3. Parsing (Process of breaking down code into its component)
4. Artificial Intelligence
5. Cryptography

Tree

- A tree is a **nonlinear data structure**, compared to arrays, linked lists, stacks and queues which are linear data structures.

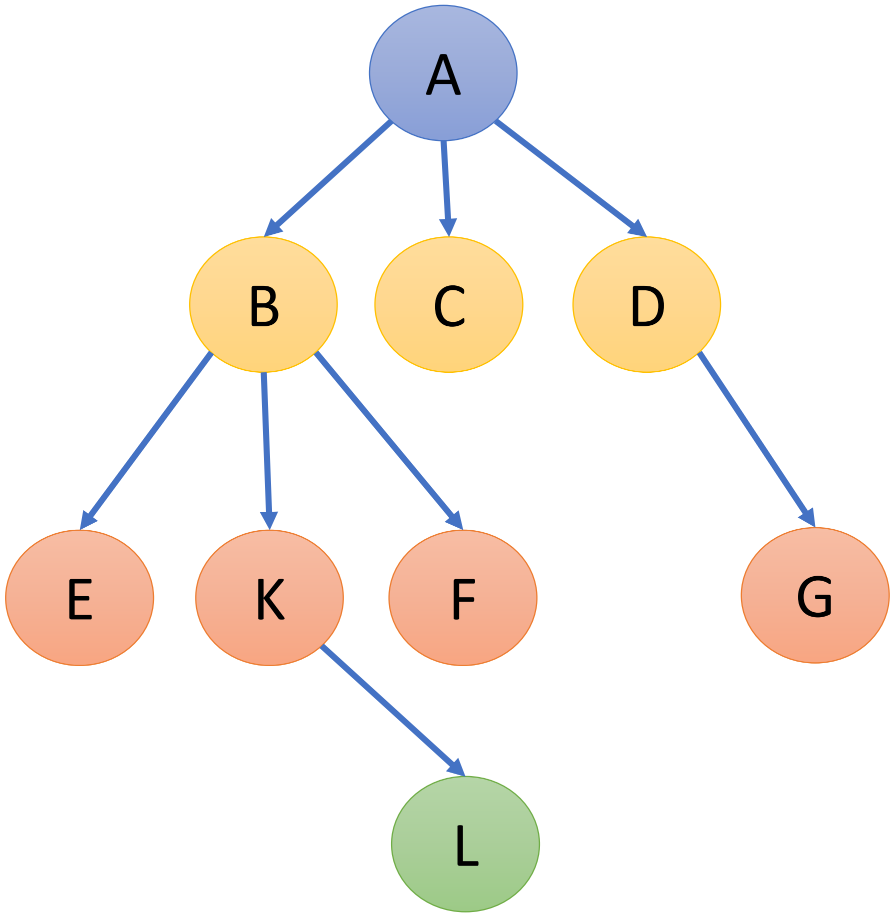
- A tree can be empty with no nodes or a tree is a structure consisting of one node called the root and zero or one or more subtrees.
- A tree has following general properties:
  - One node is distinguished as a root (A)
  - Every node (exclude a root) is connected by a directed edge from exactly one other node
  - A direction is: parent -> children



A is a parent of B, C, D,
B is called a child of A.
B is a parent of E, F, K

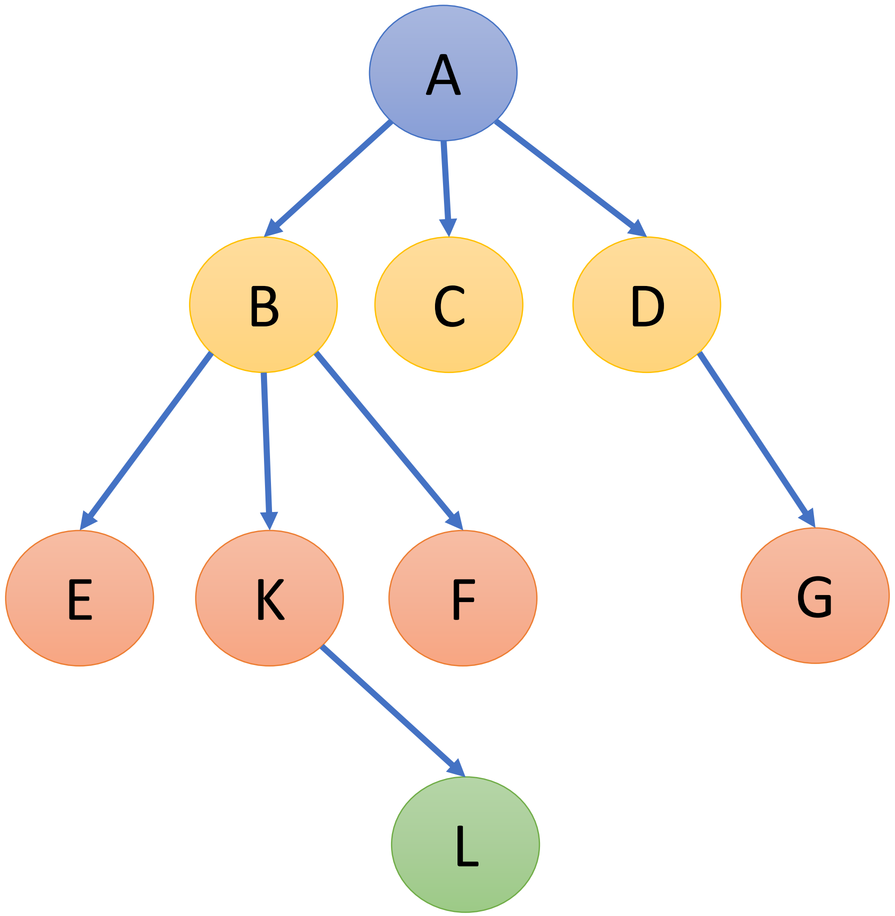
Tree

- In the picture, the root has 3 subtrees.
  - Subtree Root:
    - Node B
    - Node K
    - Node D



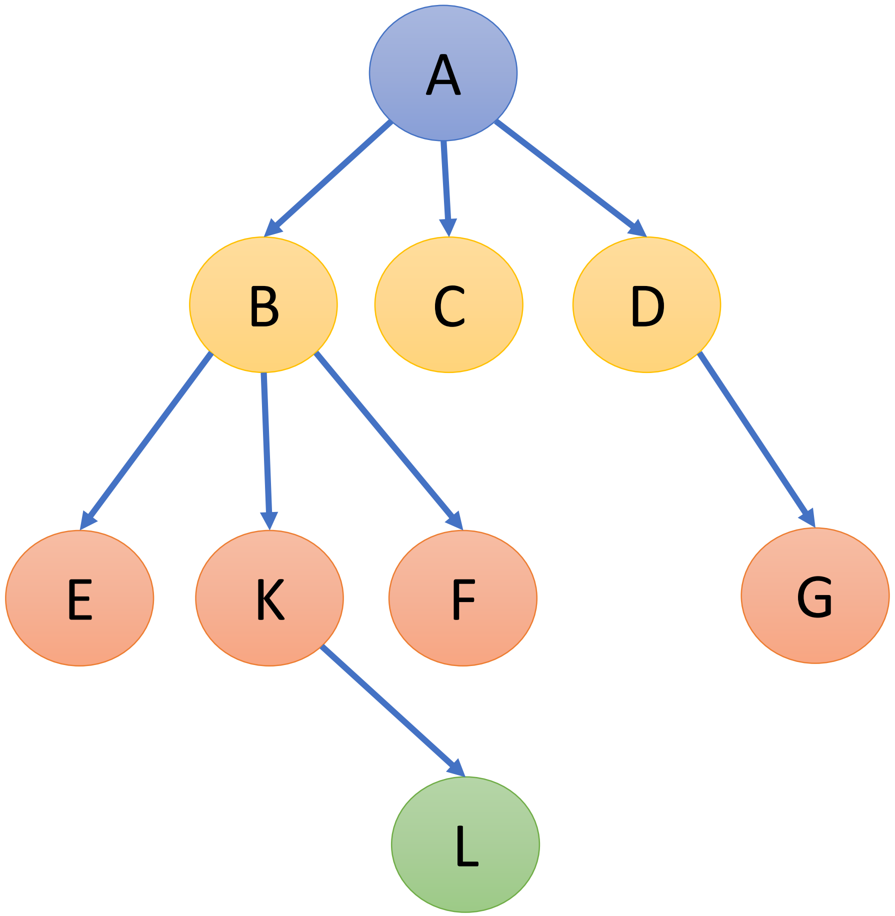
Tree

- In the picture, the root has 3 subtrees (i.e. B, K, D)
- Each node can have arbitrary number of children.
- Nodes with no children are called **leaves**, or **external** nodes.
  - In the picture, **C, E, F, L, G** are **leaves** or **external** nodes.
- Nodes, which are not leaves, are called **internal** nodes.
- Internal nodes have at least one child.



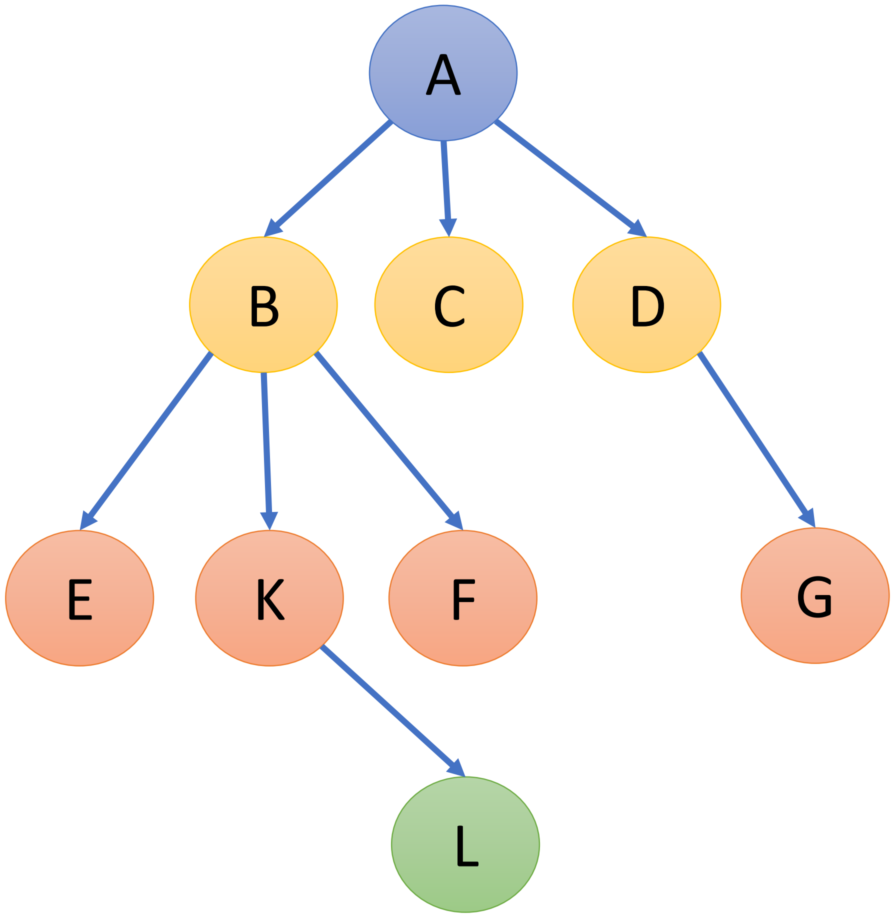
Tree

- In the picture, the root has 3 subtrees (i.e. B, K, D)
- Each node can have arbitrary number of children.
- Nodes with no children are called **leaves**, or **external** nodes.
    - In the picture, **C, E, F, L, G** are **leaves** or **external** nodes.
- Nodes, which are not leaves, are called **internal** nodes.
- Internal nodes have at least one child.
- Nodes with the same parent are called **siblings**.
    - In the picture, **B, C, D** are called **siblings**.

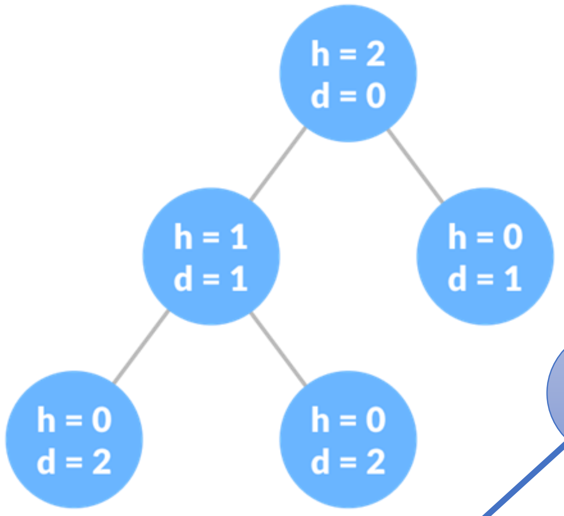


Tree

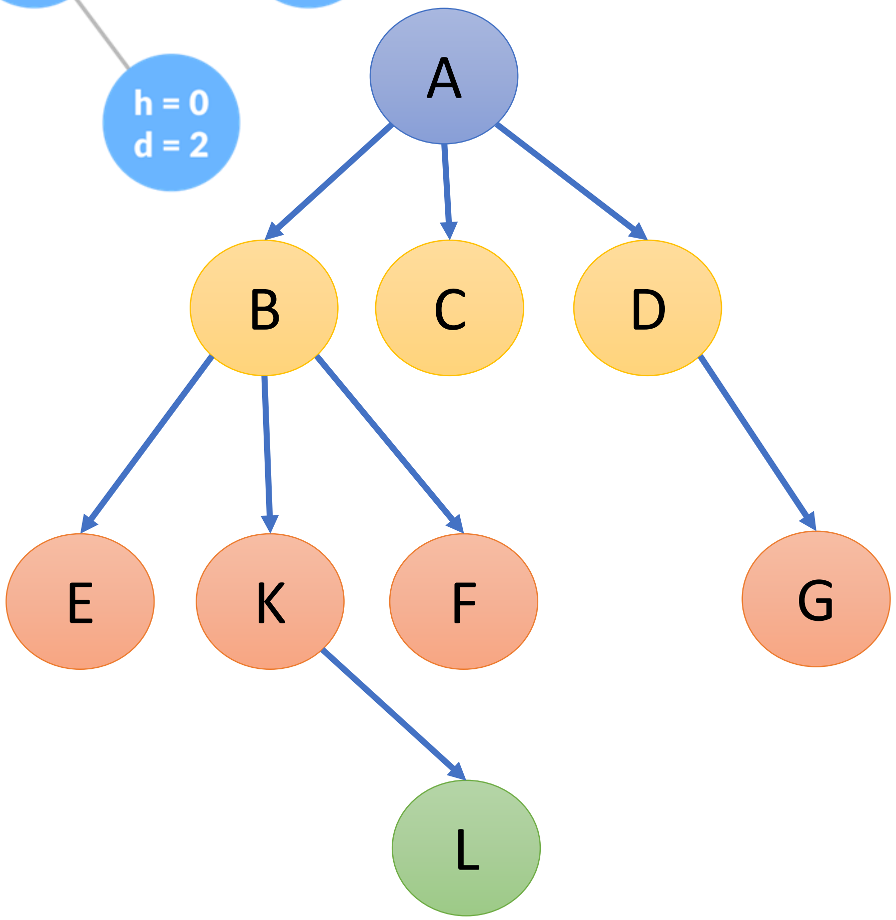
- In the picture, the root has 3 subtrees (i.e. B, K, D)
- Each node can have arbitrary number of children.
- Nodes with no children are called **leaves**, or **external** nodes.
  - In the picture, **C, E, F, L, G** are **leaves** or **external** nodes.
- Nodes, which are not leaves, are called **internal** nodes.
- Internal nodes have at least one child.
- Nodes with the same parent are called **siblings**.
  - In the picture, **B, C, D** are called **siblings**.
- The **depth of a node** is the number of edges from the root to the node.
  - The depth of K is 2.



Tree

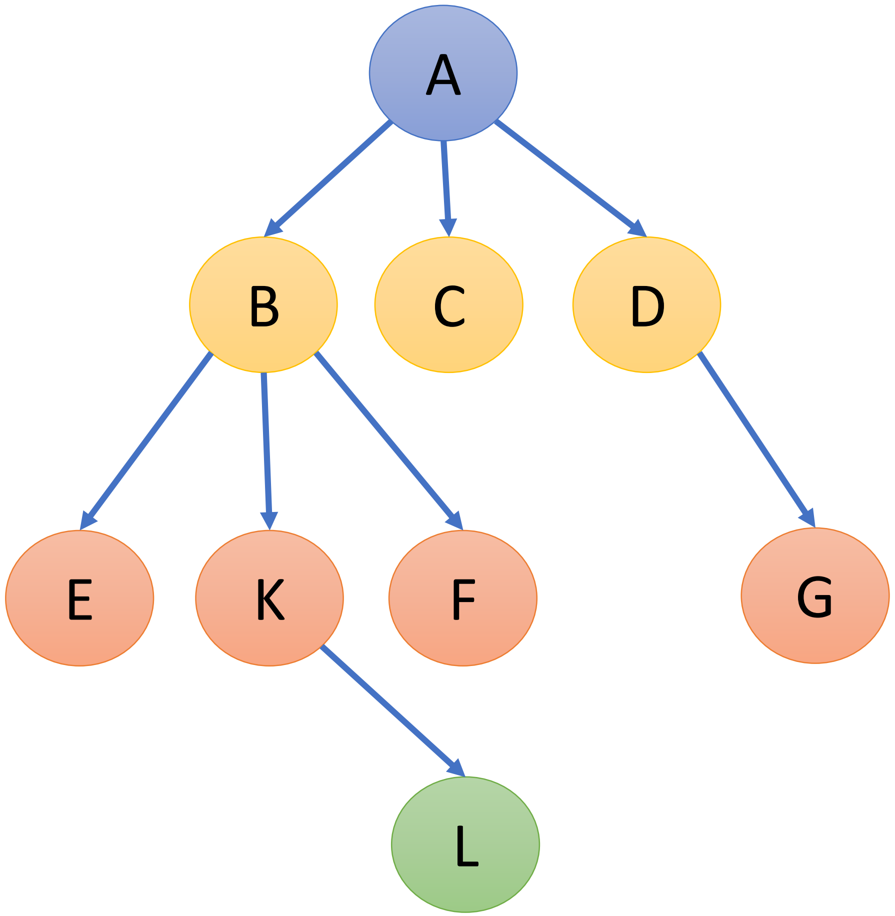


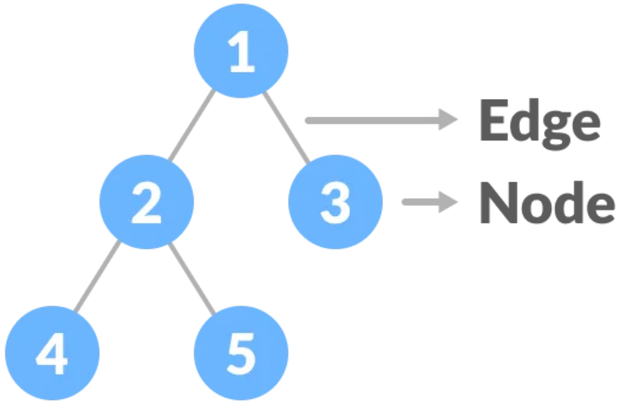
- In the picture, the root has 3 subtrees (i.e. B, K, D)
- Each node can have arbitrary number of children.
- Nodes with no children are called **leaves**, or **external** nodes.
  - In the picture, **C, E, F, L, G** are **leaves** or **external** nodes.
- Nodes, which are not leaves, are called **internal** nodes.
- Internal nodes have at least one child.
- Nodes with the same parent are called **siblings**.
  - In the picture, **B, C, D** are called **siblings**.
- The **depth (d) of a node** is the number of edges from the root to the node.
  - The depth of K is 2.
- The **height (h) of a node** is the number of edges from the node to the deepest leaf.
  - The height of **B** is 2.

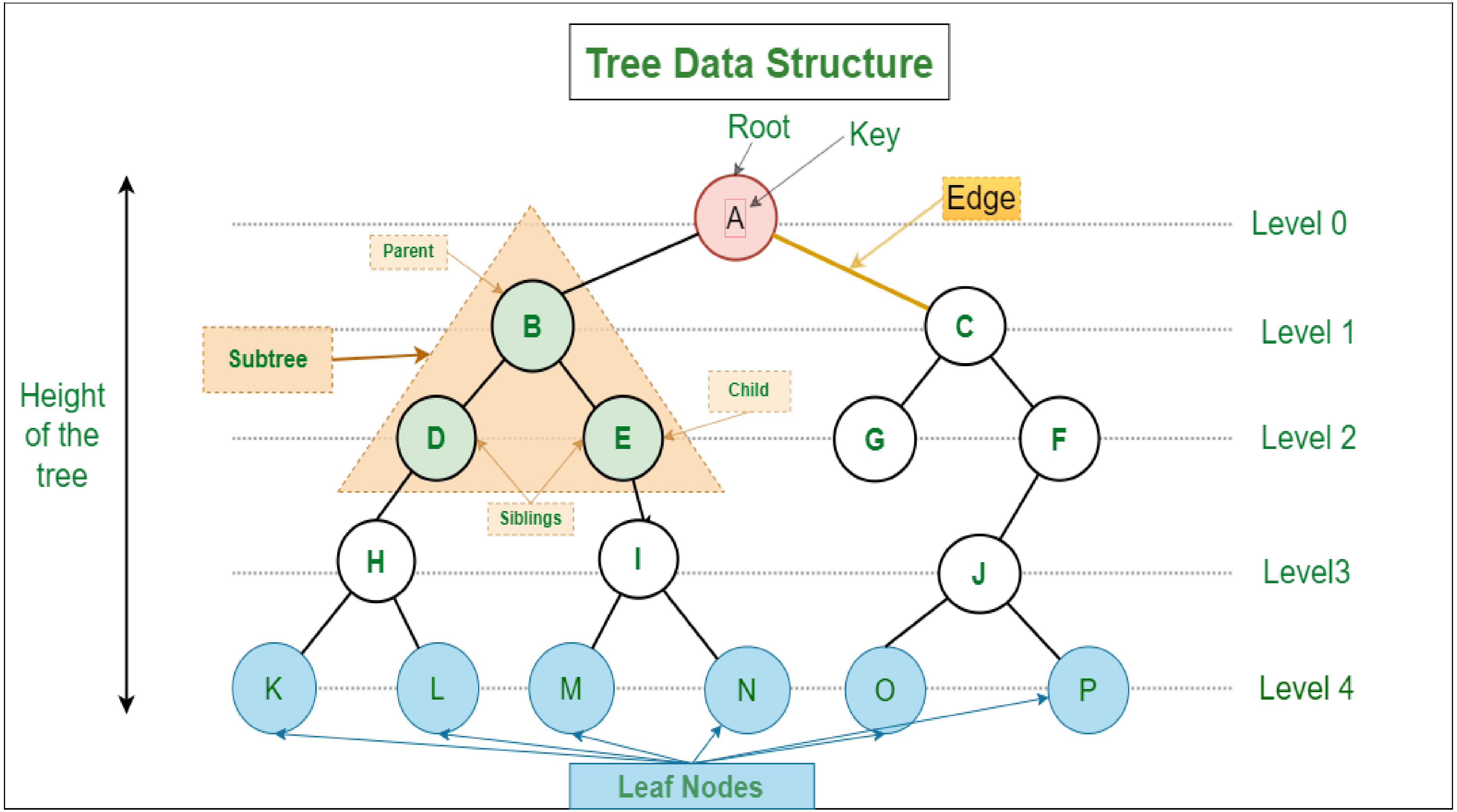


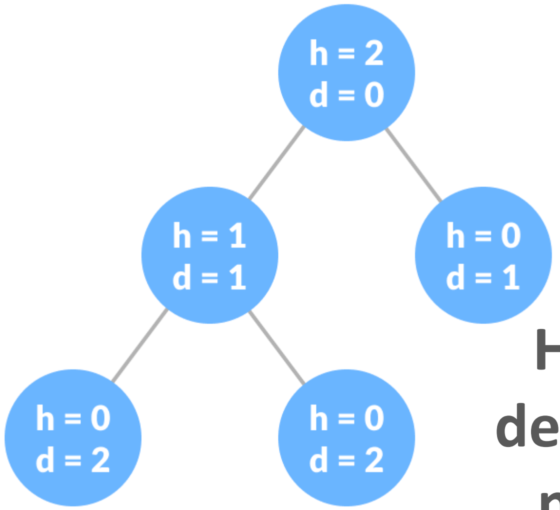
Tree

- In the picture, the root has 3 subtrees (i.e. B, K, D)
- Each node can have arbitrary number of children.
- Nodes with no children are called **leaves**, or **external** nodes.
  - In the picture, **C, E, F, L, G** are **leaves** or **external** nodes.
- Nodes, which are not leaves, are called **internal** nodes.
- Internal nodes have at least one child.
- Nodes with the same parent are called **siblings**.
  - In the picture, **B, C, D** are called **siblings**.
- The **depth of a node** is the number of edges from the root to the node.
  - The depth of K is 2.
- The **height of a node** is the number of edges from the node to the deepest leaf.
  - The height of **B** is 2.
- The **height of a tree** is a height of a root.





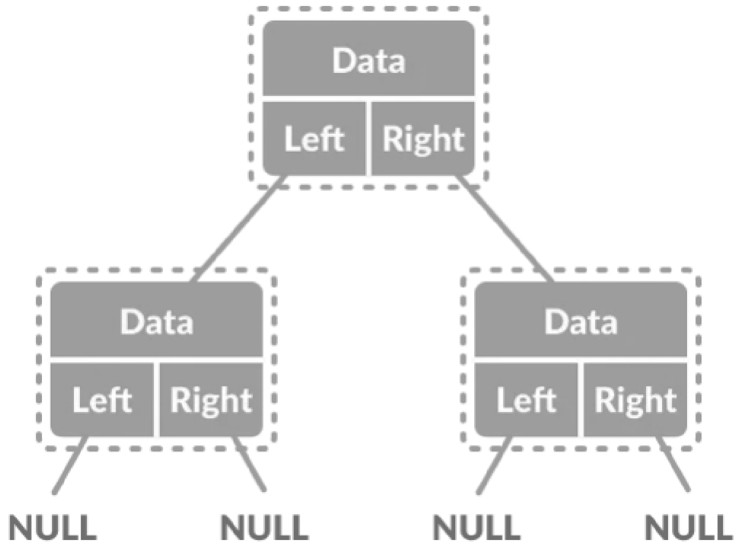


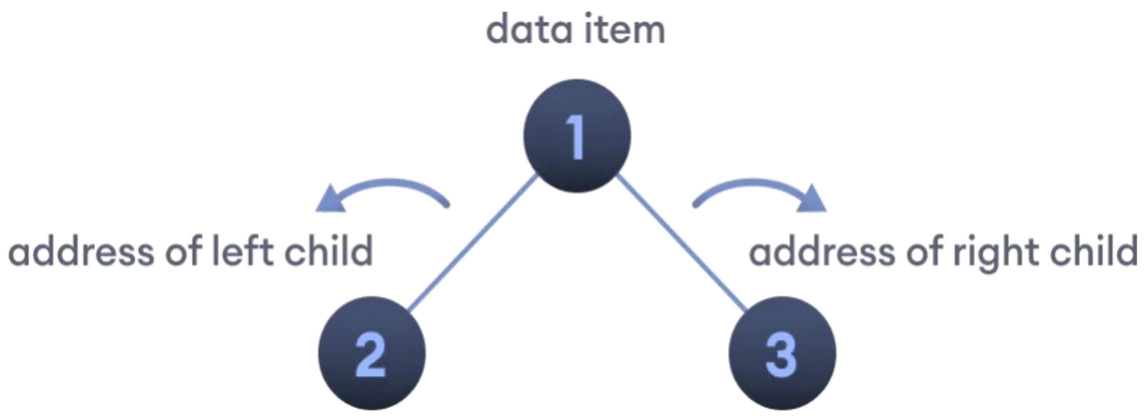


Height (h) and depth (d) of each node in a tree

Image Source

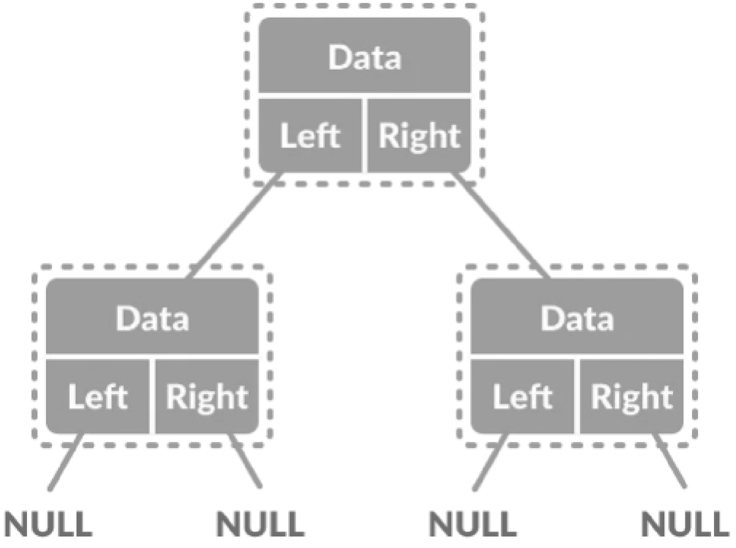
Binary Tree

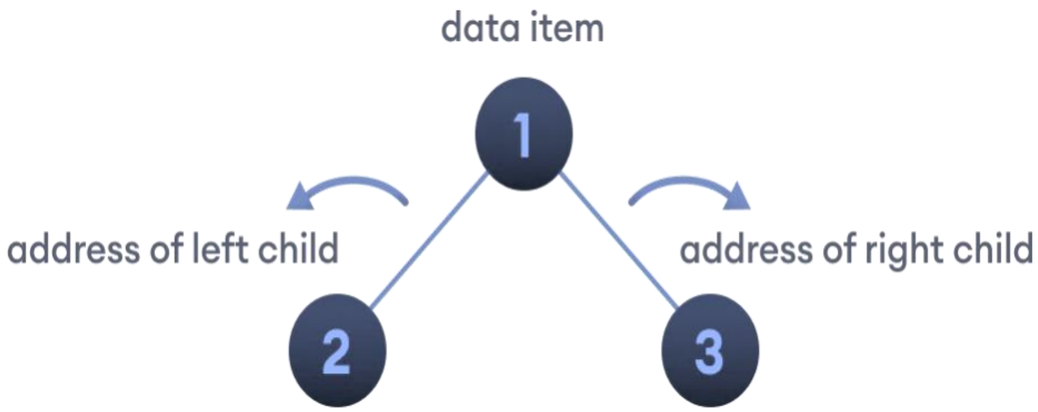




- A **binary tree** is a structurally complete data structure in which each node has at most two children.
- A binary tree usually has two nodes, called the left and right nodes, with the left being less than the right.
- Binary trees are generally used for quick storage and retrieval of data. Because each node can only have two children, it is easy to find a particular data piece without searching through the entire structure.

Binary Tree

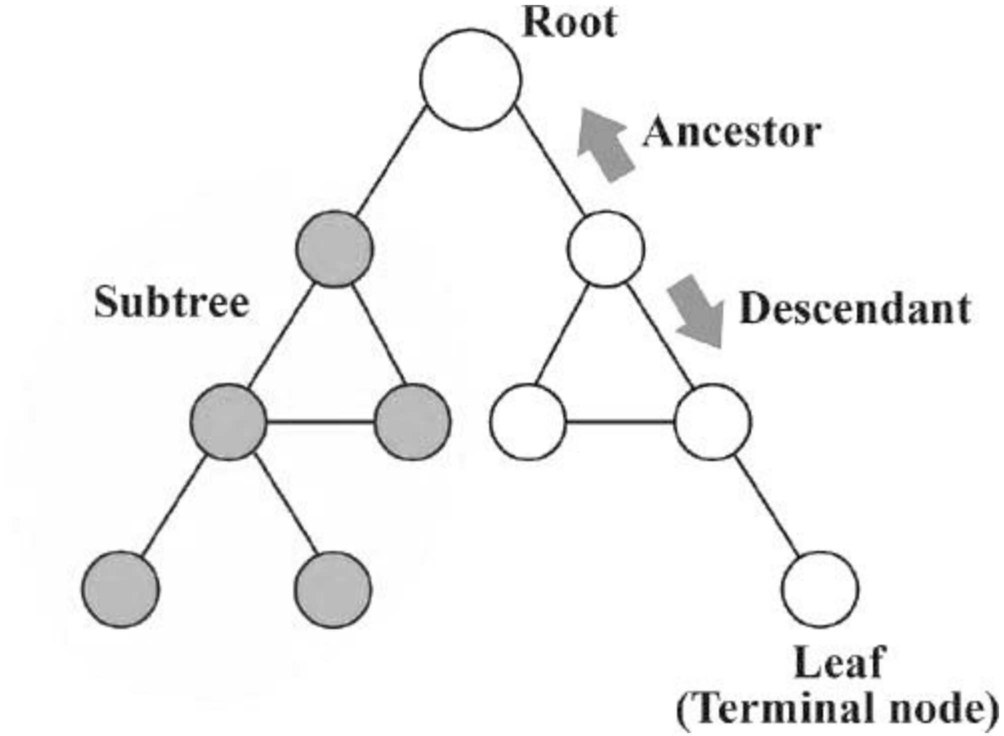




- Additionally, binary trees can be traversed using either a recursive or iterative algorithm.
- As a result, a binary tree in the data structure is often used when performance is critical, such as in real-time applications.
    - Binary search tree (BST): Used to search applications where data is continuously entering and leaving.
    - Binary space partition: Used in 3D video games to determine what objects need to be rendered.
    - Binary trees: Used by high-bandwidth routers for storing router tables, implementing dictionaries, spelling checking etc.

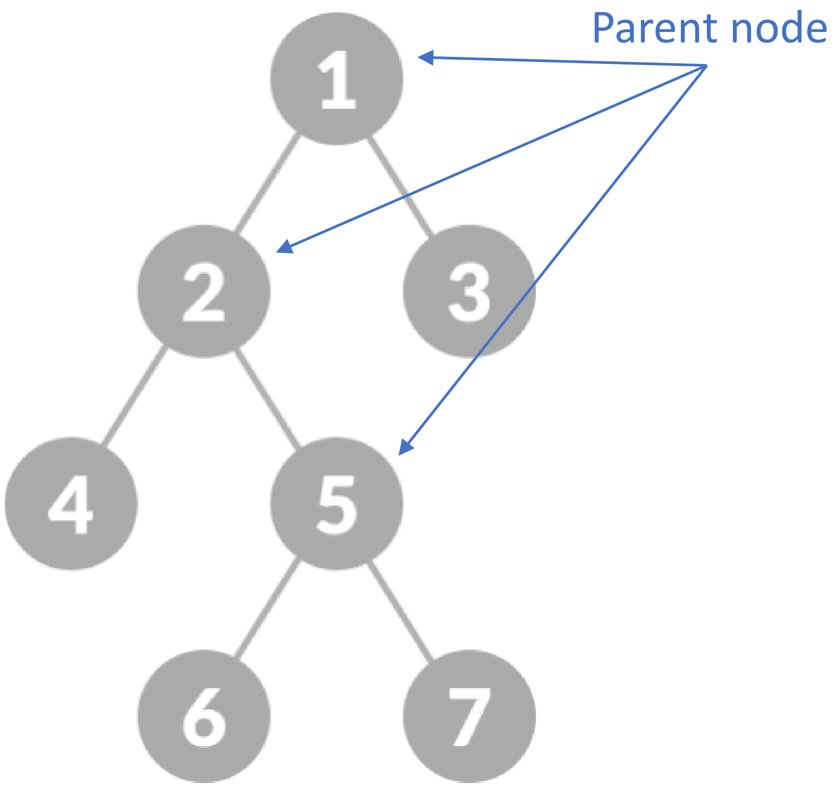
Terminologies Associated with Binary Trees

- **Ancestor Nodes: A**ny node that is higher up in the tree than a given child node.
- **Descendant Nodes: A**ny node that is lower down in the tree than a given parent node.
- **Climbing/Ascending**: Traversing from leaf to root
- **Walking/Descending**: Traversing from root to leaf
- Root Node, Child Node, Sibling Nodes, Leaf Nodes, Internal Nodes, Height, Depth (Already discussed)



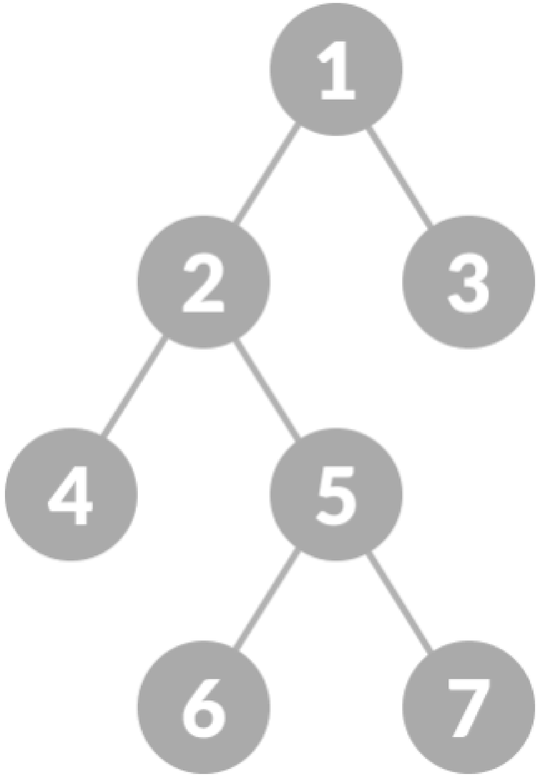
Types of Binary Tree (Completion of levels)

Full Binary Tree: Every parent node/internal node has either two or no children.

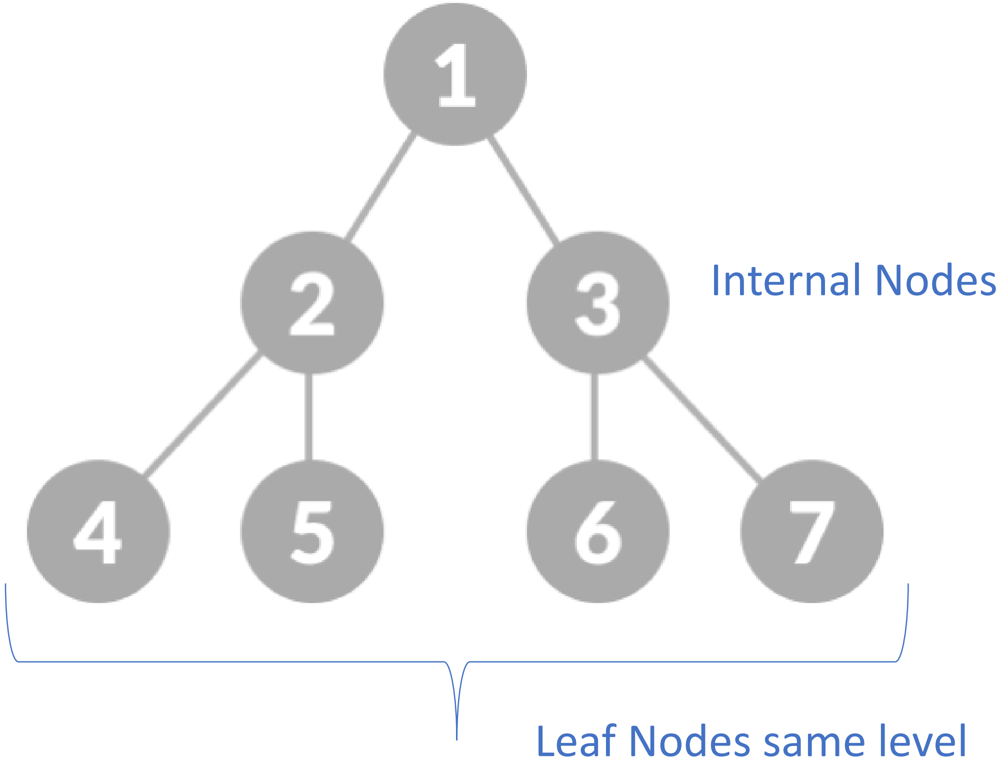


Types of Binary Tree (Completion of levels)

Full Binary Tree: Every parent node/internal node has either two or no children.

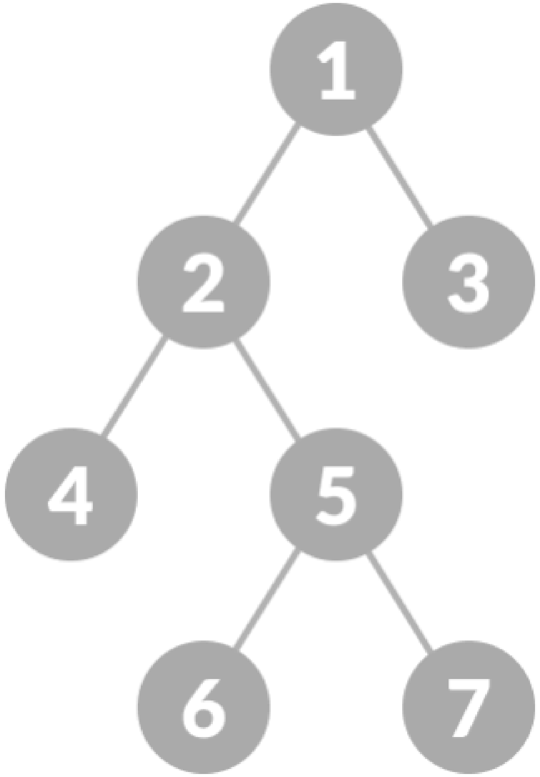


Perfect Binary Tree: Every internal node has exactly two child nodes and all the leaf nodes are at the same level

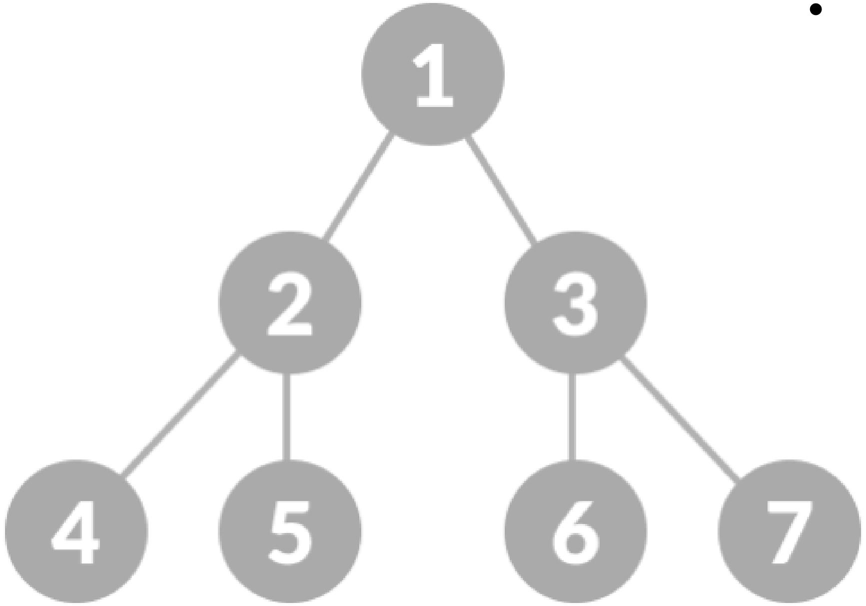


Types of Binary Tree (Completion of levels)

Full Binary Tree: Every parent node/internal node has either two or no children.



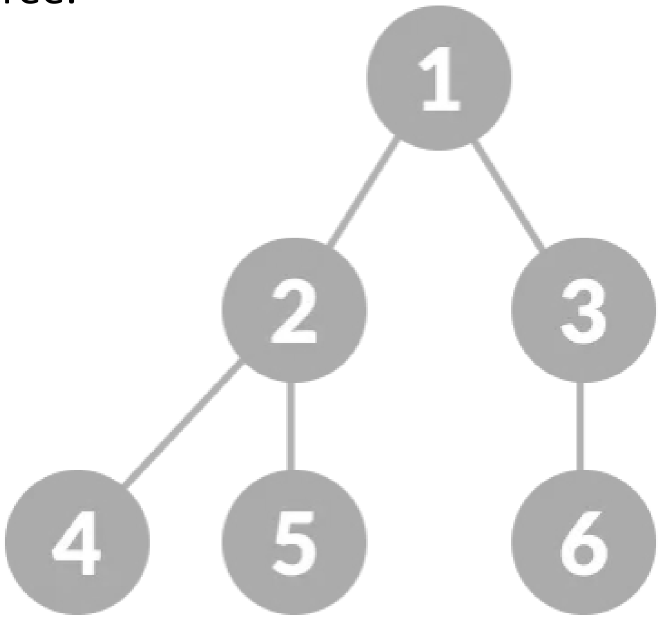
Perfect Binary Tree: Every internal node has exactly two child nodes and all the leaf nodes are at the same level



A **complete binary** tree is just like a full binary tree, but with two major differences

1. Every level must be completely filled, except possibly the last level
2. All the leaf elements must lean towards the left.

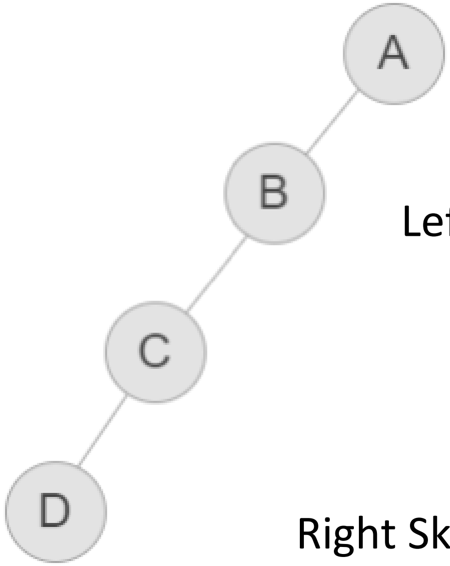
- The last leaf element might not have a right sibling i.e. a complete binary tree doesn't have to be a full binary tree.



Types of Binary Tree (Completion of levels)

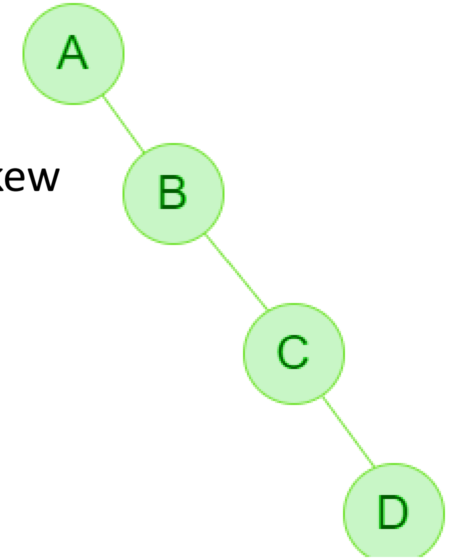
- **Skewed Binary Tree**

A skewed binary tree is a pathological/degenerate tree in which the tree is either dominated by the left nodes or the right nodes. Thus, there are two types of skewed binary tree: left-skewed binary tree and right-skewed binary tree.



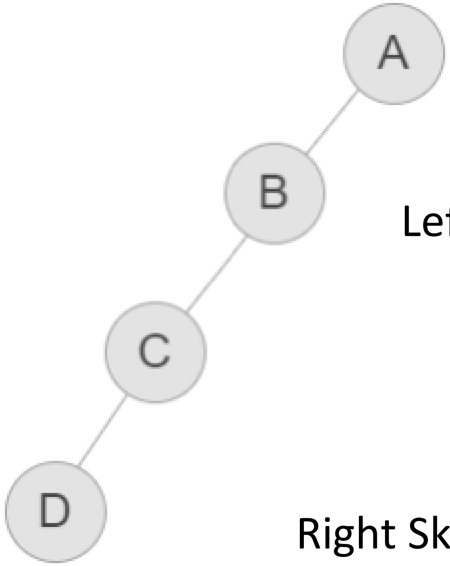
Left Skew

Right Skew



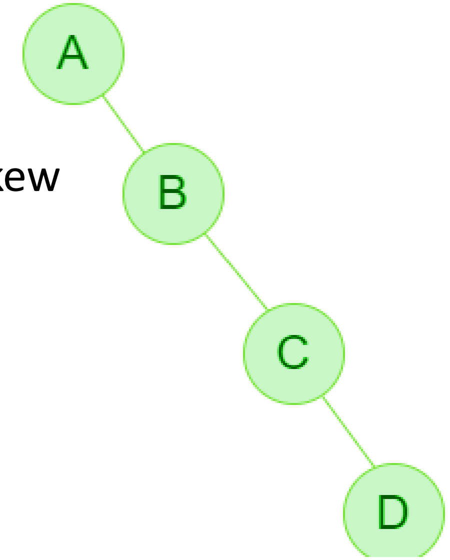
Types of Binary Tree (Completion of levels)

- **Skewed Binary Tree** A skewed binary tree is a pathological/degenerate tree in which the tree is either dominated by the left nodes or the right nodes. Thus, there are two types of skewed binary tree: left-skewed binary tree and right-skewed binary tree.

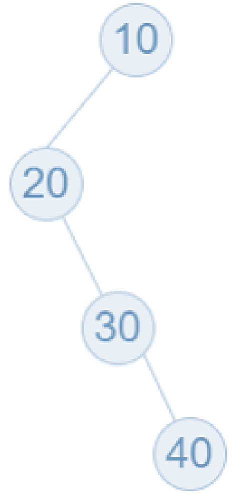


Left Skew

Degenerate (or pathological) tree
A Tree where every internal node has one child. Such trees are performance-wise same as linked list.
A degenerate or pathological tree is a tree having a single child either left or right.

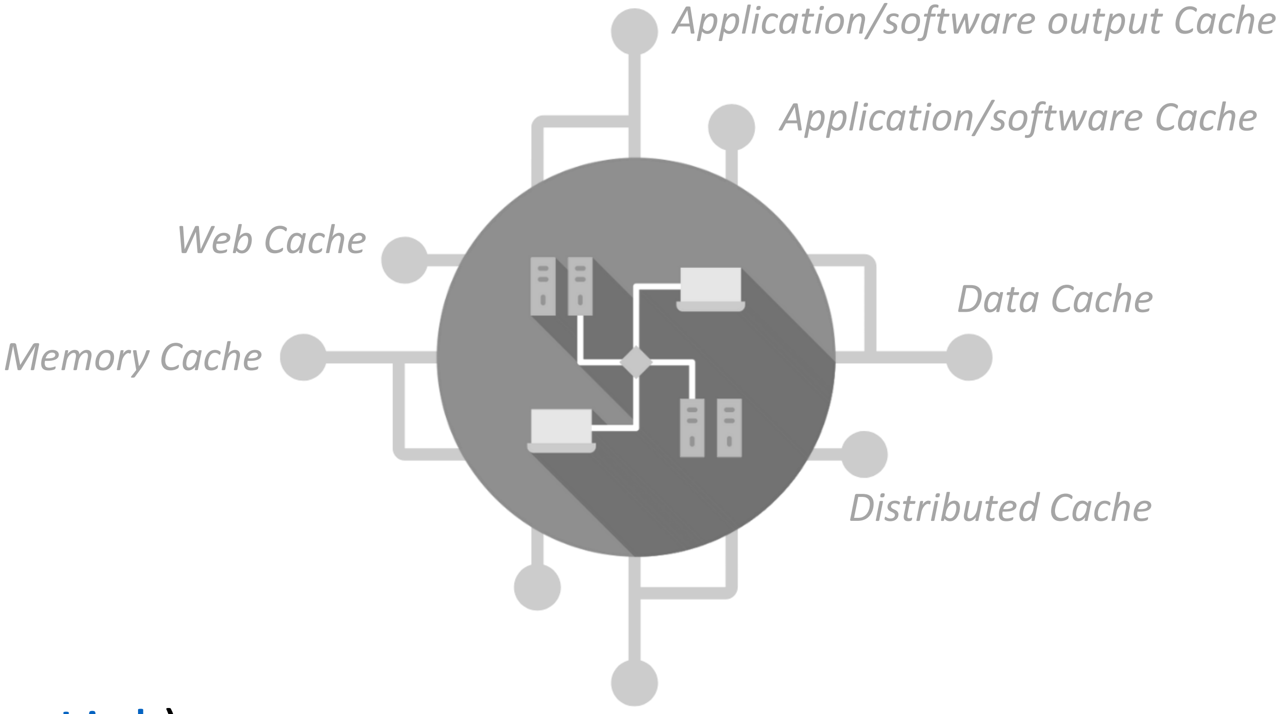


Right Skew



Degenerate (or pathological) tree

Types of Binary Tree

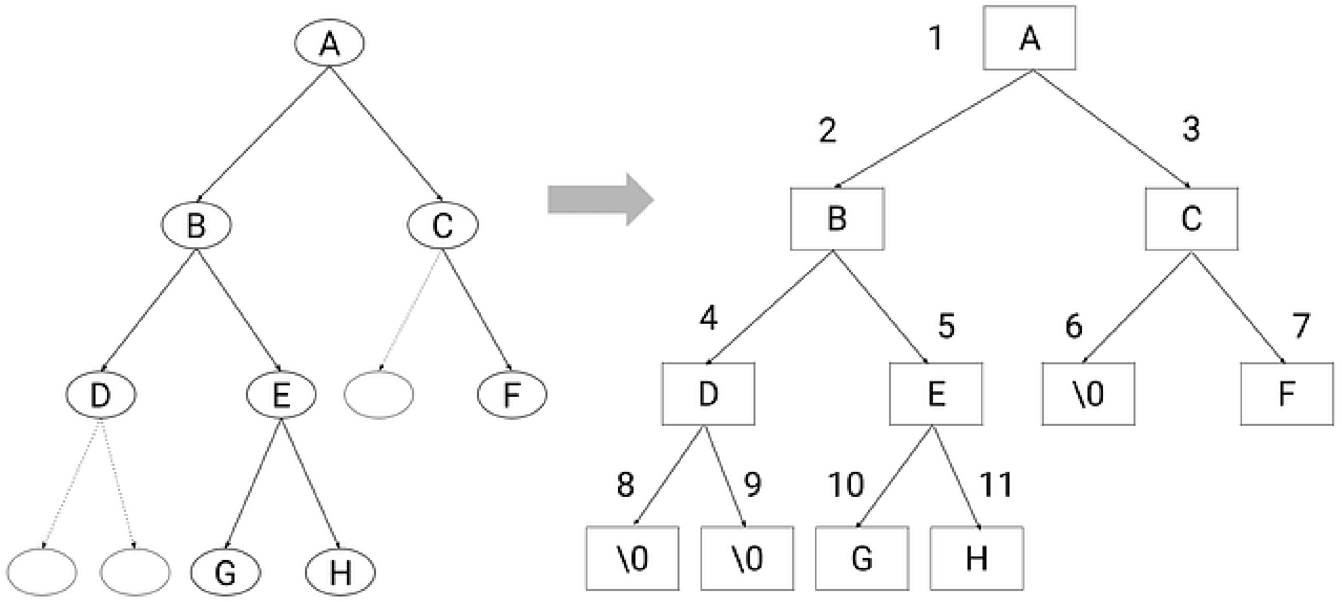


- Paper: Skewed Binary Search Trees (Source: Link)
- In this paper we present an experimental study of various memory layouts of static skewed binary search trees, where each element in the tree is accessed with a uniform probability.
- Our results show that for many of the memory layouts we consider skewed binary search trees can perform better than perfect balanced search trees.
- The improvements in the running time are on the order of 15%.
- Previous work has shown that a dominating factor over the running time for a search is the number of cache faults performed, and that an appropriate memory layout of a binary search tree can reduce the number of cache faults by several hundred percent.

Binary Tree (Array)

Trees can be represented in two ways :

- Dynamic Node Representation (Linked Representation).
- Array Representation (Sequential Representation).

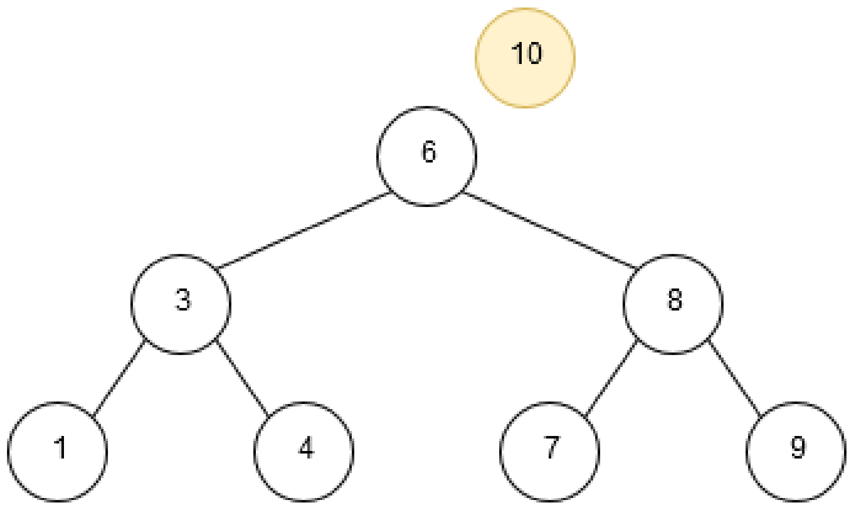


	1	2	3	4	5	6	7	8	9	10	11
Tree Array	A	B	C	D	E	\0	F	\0	\0	G	H

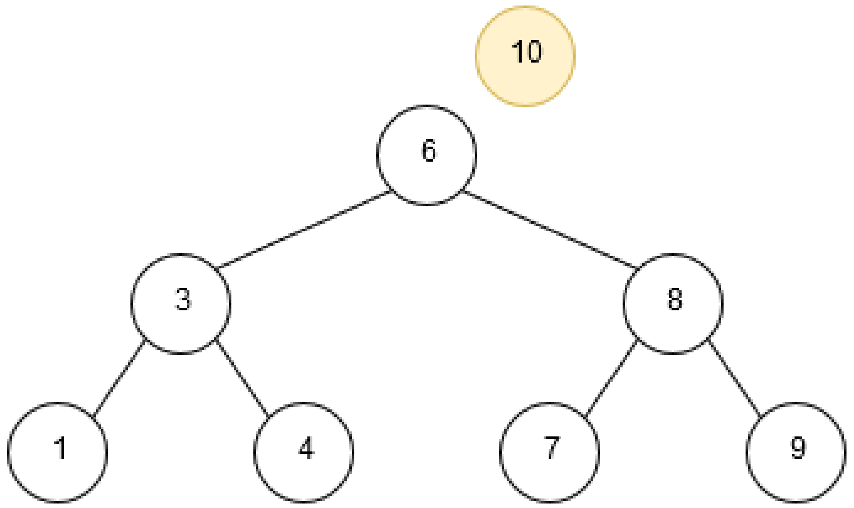
Source: [Link]()

Binary Search Tree (BST)

- Binary search tree is a data structure that quickly allows us to maintain a sorted list of numbers.
  - It is called a binary tree because each tree node has a maximum of two children.
  - It is called a search tree because it can be used to search for the presence of a number in O(log(n)) time.



Binary Search Tree (BST)



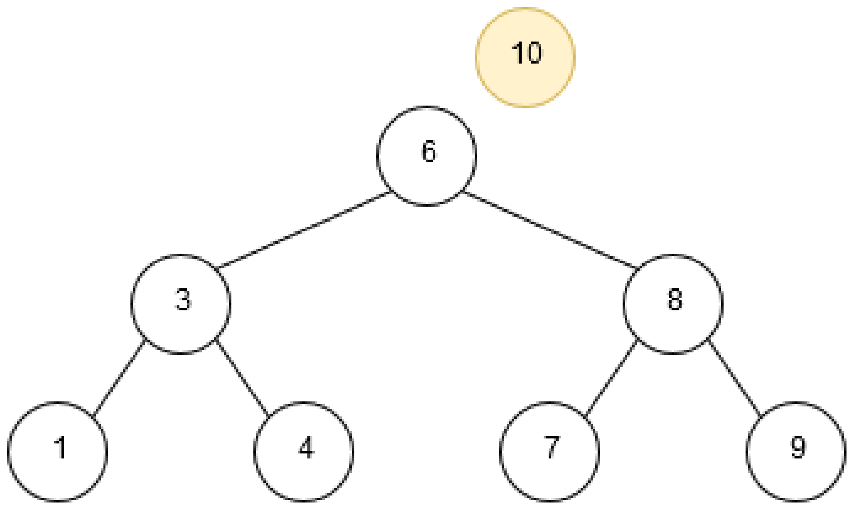
- Binary search tree is a data structure that quickly allows us to maintain a sorted list of numbers.
  - It is called a binary tree because each tree node has a maximum of two children.
  - It is called a search tree because it can be used to search for the presence of a number in O(log(n)) time.

```c
struct node {
  int data; //node will store some data
  struct node *right_child; // right child
  struct node *left_child; // left child
};

//function to create a node
struct node* new_node(int x) {
  struct node *temp;
  temp = malloc(sizeof(struct node));
  temp -> data = x;
  temp -> left_child = NULL;
  temp -> right_child = NULL;

  return temp;
}
```

Binary Search Tree (BST)



- Binary search tree is a data structure that quickly allows us to maintain a sorted list of numbers.
  - It is called a binary tree because each tree node has a maximum of two children.
  - It is called a search tree because it can be used to search for the presence of a number in O(log(n)) time.

- The properties that separate a binary search tree from a regular binary tree is
  1. All nodes of **left subtree** are **less** than the root node
  2. All nodes of **right subtree** are **more** than the root node
  3. Both subtrees of each node are also BSTs i.e. they have the above two properties

```
struct node {
  int data; //node will store some data
  struct node *right_child; // right child
  struct node *left_child; // left child
};

//function to create a node
struct node* new_node(int x) {
  struct node *temp;
  temp = malloc(sizeof(struct node));
  temp -> data = x;
  temp -> left_child = NULL;
  temp -> right_child = NULL;

  return temp;
}
```

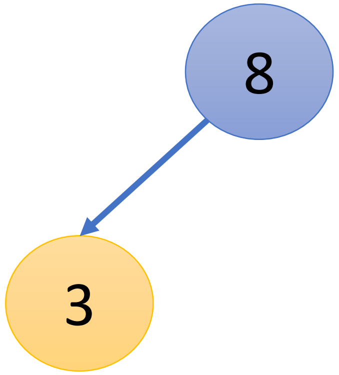
Binary Search Tree(BST)

Elements: 8,3,10,1,6,4

- Insert a new node starting at the root (set current node to root)
  - If new node is < current, move left
  - If new node is >= current, move right
  - Repeat this until current is null. Insert it here.
- This is similar to binary search of an array

Binary Search Tree(BST)

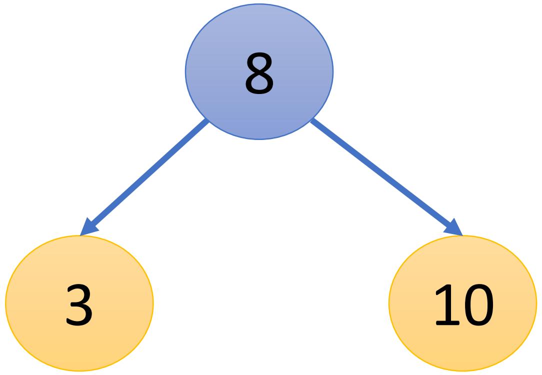
Elements: 8,3,10,1,6,4



- Insert a new node starting at the root (set current node to root)
  - If new node is < current, move left
  - If new node is >= current, move right
  - Repeat this until current is null. Insert it here.
- This is similar to binary search of an array

Binary Search Tree(BST)

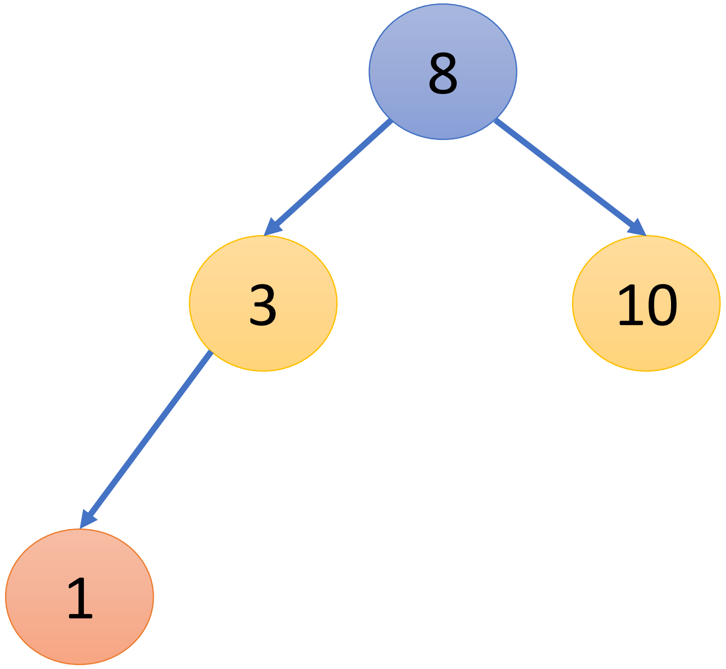
Elements: 8,3,10,1,6,4



- Insert a new node starting at the root (set current node to root)
  - If new node is < current, move left
  - If new node is >= current, move right
  - Repeat this until current is null. Insert it here.
- This is similar to binary search of an array

Binary Search Tree(BST)

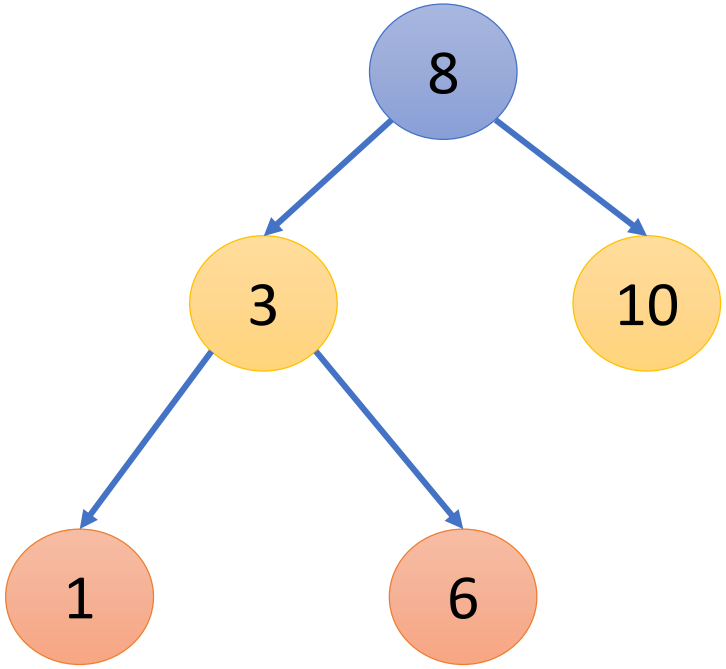
Elements: 8,3,10,1,6,4



- Insert a new node starting at the root (set current node to root)
  - If new node is < current, move left
  - If new node is >= current, move right
  - Repeat this until current is null. Insert it here.
- This is similar to binary search of an array

Binary Search Tree(BST)

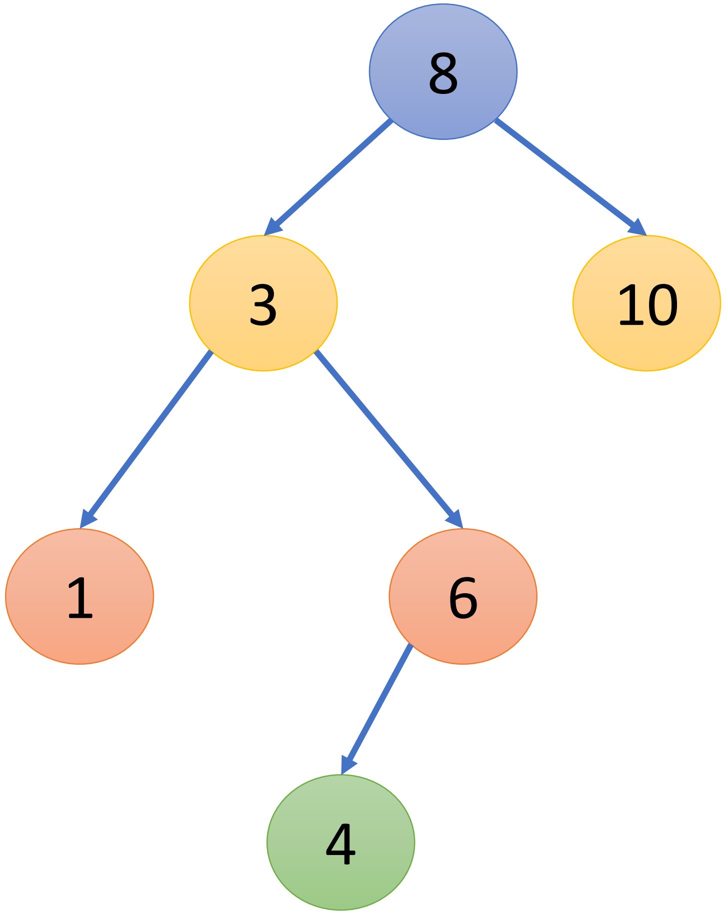
Elements: 8,3,10,1,6,4



- Insert a new node starting at the root (set current node to root)
  - If new node is < current, move left
  - If new node is >= current, move right
  - Repeat this until current is null. Insert it here.
- This is similar to binary search of an array

Binary Search Tree(BST)

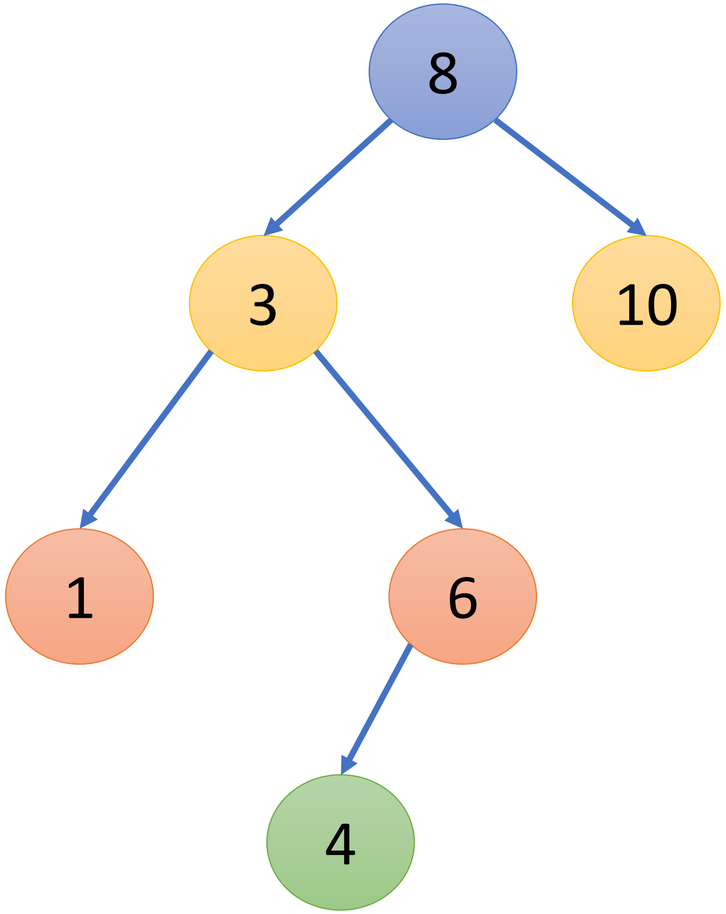
Elements: 8,3,10,1,6,4



- Insert a new node starting at the root (set current node to root)
  - If new node is < current, move left
  - If new node is >= current, move right
  - Repeat this until current is null. Insert it here.
- This is similar to binary search of an array

Binary Search Tree(BST)

Elements: 8,3,10,1,6,4

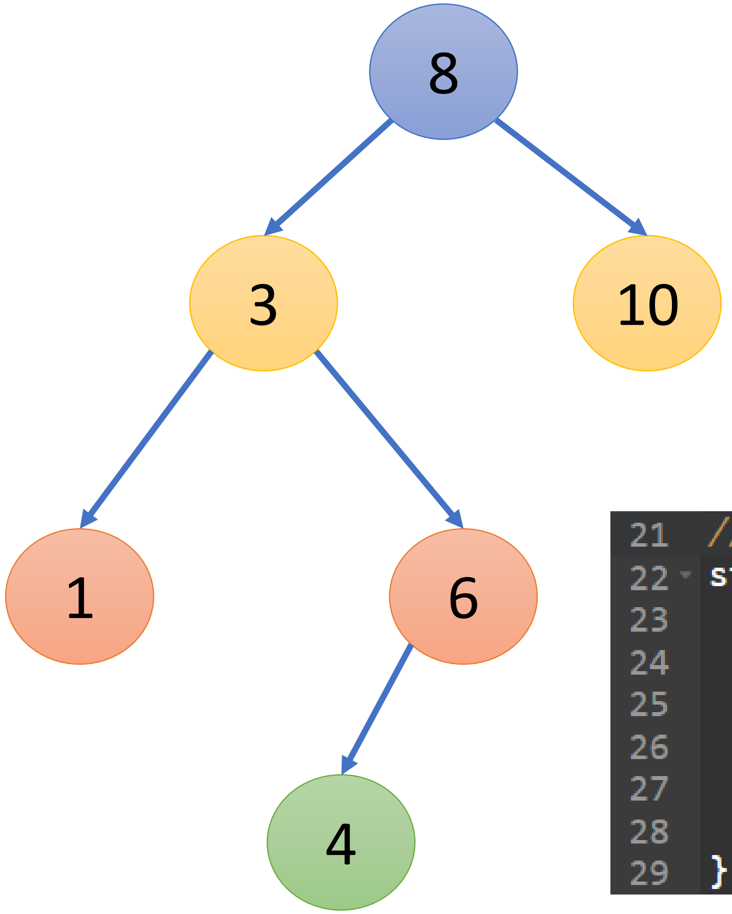


- Insert a new node starting at the root (set current node to root)
  - If new node is < current, move left
  - If new node is >= current, move right
  - Repeat this until current is null. Insert it here.
- This is similar to binary search of an array

```
// insertion
struct node* insert(struct node * root, int x) {
  //searching for the place to insert
  if (root == NULL)
    return new_node(x);
  else if (x > root -> data) // x is greater. Should be inserted to the right
    root -> right_child = insert(root -> right_child, x);
  else // x is smaller and should be inserted to left
    root -> left_child = insert(root -> left_child, x);
  return root;
}
```

Binary Search Tree(BST)

Elements: 8,3,10,1,6,4



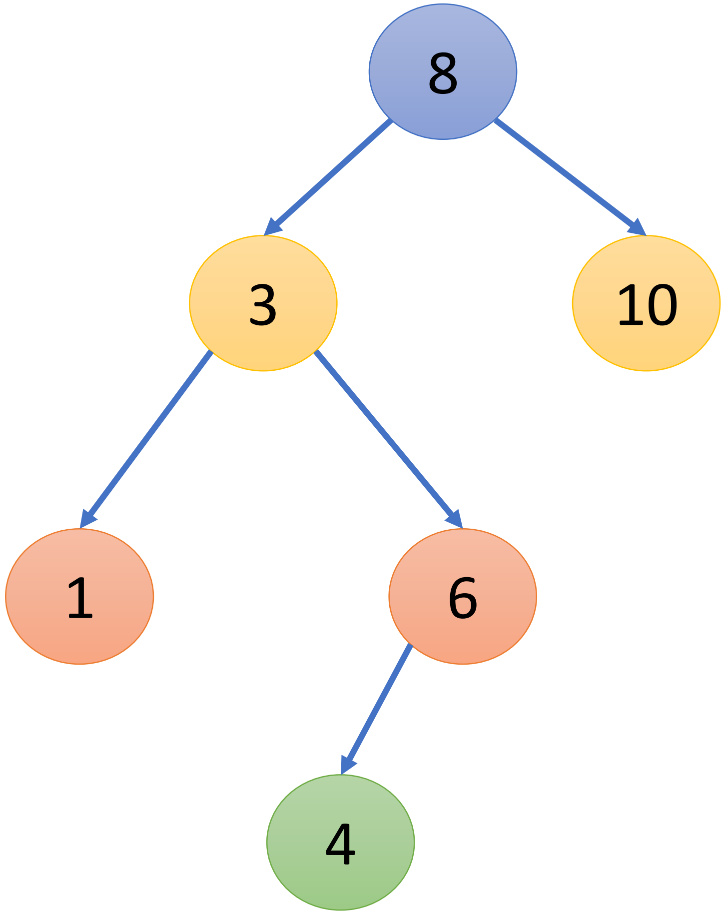
- Insert a new node starting at the root (set current node to root)
    - If new node is < current, move left
    - If new node is >= current, move right
    - Repeat this until current is null. Insert it here.
- This is similar to binary search of an array

Searching ?

```
// searching operation
struct node* search(struct node * root, int x) {
  if (root == NULL || root -> data == x) //if root->data is x then the element is found
    return root;
  else if (x > root -> data) // x is greater, so we will search the right subtree
    return search(root -> right_child, x);
  else //x is smaller than the data, so we will search the left subtree
    return search(root -> left_child, x);
}
```

Binary Search Tree(BST)

Elements: 8,3,10,1,6,4



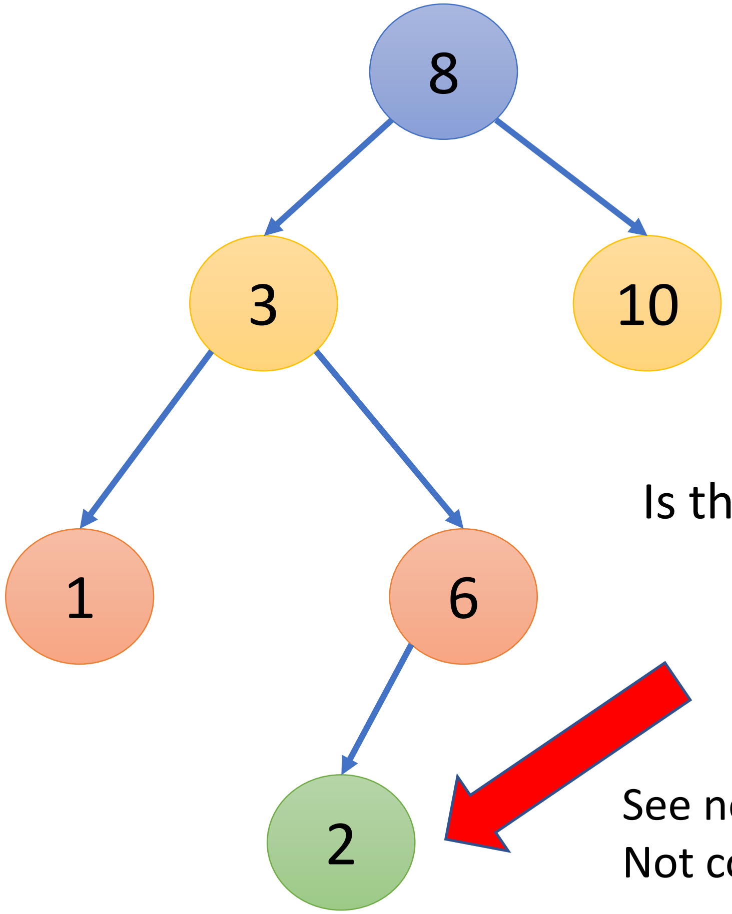
- Insert a new node starting at the root (set current node to root)
  - If new node is < current, move left
  - If new node is >= current, move right
  - Repeat this until current is null. Insert it here.
- This is similar to binary search of an array

```
// searching operation
struct node* search(struct node * root, int x) {
  if (root == NULL || root -> data == x) //if root->data is x then the element is found
    return root;
  else if (x > root -> data) // x is greater, so we will search the right subtree
    return search(root -> right_child, x);
  else //x is smaller than the data, so we will search the left subtree
    return search(root -> left_child, x);
}
```

```
// insertion
struct node* insert(struct node * root, int x) {
  //searching for the place to insert
  if (root == NULL)
    return new_node(x);
  else if (x > root -> data) // x is greater. Should be inserted to the right
    root -> right_child = insert(root -> right_child, x);
  else // x is smaller and should be inserted to left
    root -> left_child = insert(root -> left_child, x);
  return root;
}
```

Binary Search Tree(BST - Example)

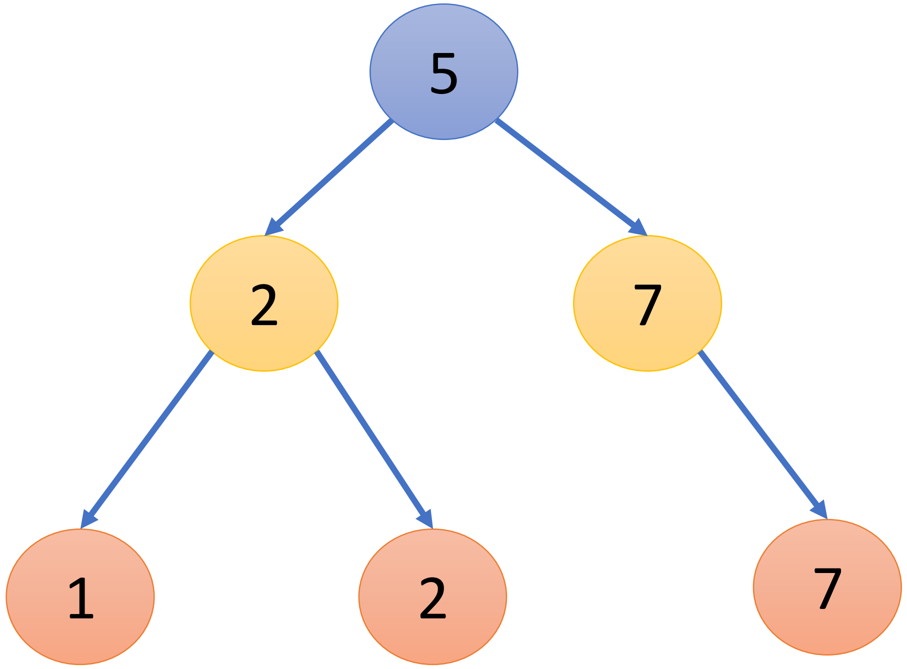
Elements: 8,3,10,1,6,2



Is this a BST?

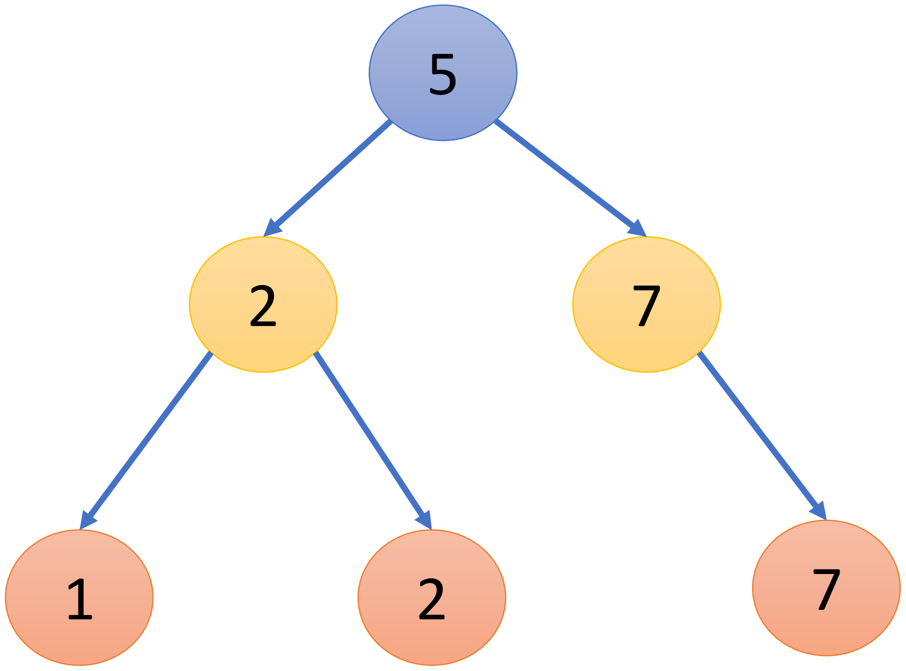
See node 2!!
Not correctly places in the tree

Elements: 5,2,7,1,7,2

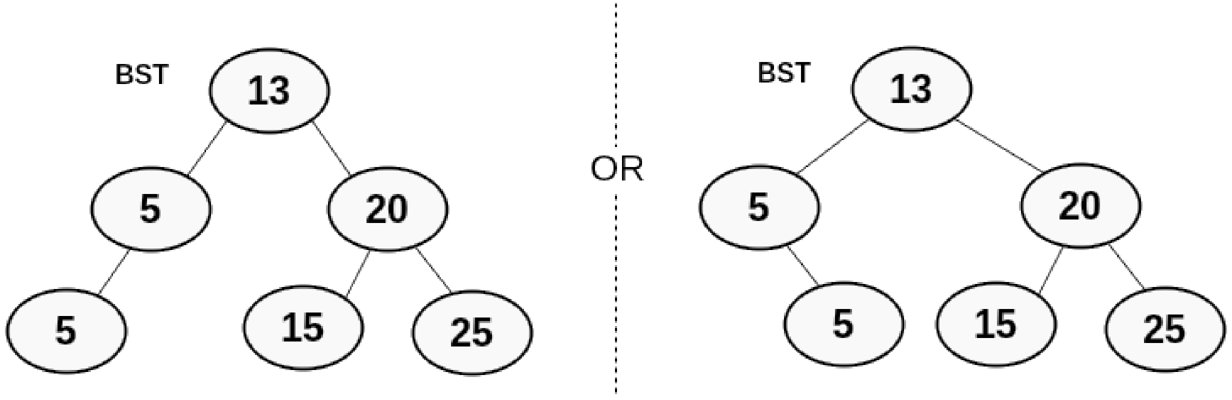


Binary Search Tree(BST - Example)

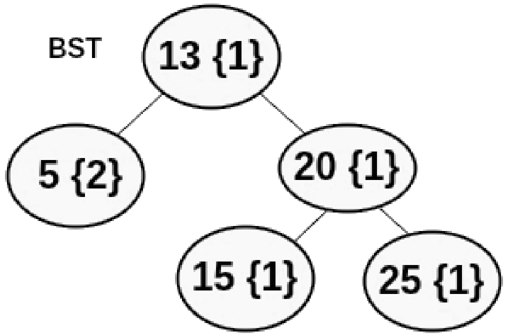
Elements: 5,2,7,1,7,2



1. Store the duplicate element in the left or right subtree



2. Stores the count of the node. So the count of Node 5 will be 2



Source: [Link]