Today's topics

- Trees
- Binary search trees

References

- Read chapter-4 from Mark Allen Weiss
- Read chapter-12 from Cormen et al.

Linear data structures

- Array and Linked Lists are linear structures that contain a sequence of elements

Linear data structures

- Array and Linked Lists are linear structures that contain a sequence of elements
  - What is the time complexity of an insert operation in an unordered array? $O(1)$
  - What is the time complexity of a delete operation in an unordered array? $O(1)$
  - What is the time complexity of a search operation in an unordered array? $O(n)$
  - What is the time complexity of an insert operation in a sorted array? $O(n)$
  - What is the time complexity of a delete operation in a sorted array? $O(n)$
  - What is the time complexity of a search operation in a sorted array? $O(\log n)$

O(n) vs O(log(n))

n	log(n)
1024	10
2048	11
4096	12
8192	13
16384	14
32768	15
65536	16
131072	17
262144	18
524288	19
1048576	20

From this table, it’s evident that O(log(n)) is much better than O(n). For most practical purposes, log(n) is somewhat close to a constant time operation. For a problem size of 1M, an O(log(n)) algorithm takes merely 20 operations. In 100 operations, we can solve a problem size of 2^100. If the space complexity of the problem is also O(n), we can not even store the input for the problem size 2^100 (10^9 Billion TB) on probably any machine.

It would be great if we could do all search, insert, and delete operations in O(log(n)).

Search and update

- Why is search important?
    - We do search all the time
        - Web search engines (the input is the list of all web pages and their contents)
        - Searching for a file on your system (the input is the list of all files and directories)
        - YouTube (input is the list of the captions and descriptions of all videos)
        - Twitter (searching for a particular hashtag)
    - In these applications, new contents are added or removed all the time
        - search, delete, and insert usually go hand in hand

Search and update

- We can’t do search and update (insert or delete) operations simultaneously in O(log n) time using linear data structures ☹

- A non-linear (hierarchical) data structure called a tree is used to perform all these operations in O(log n) ☺

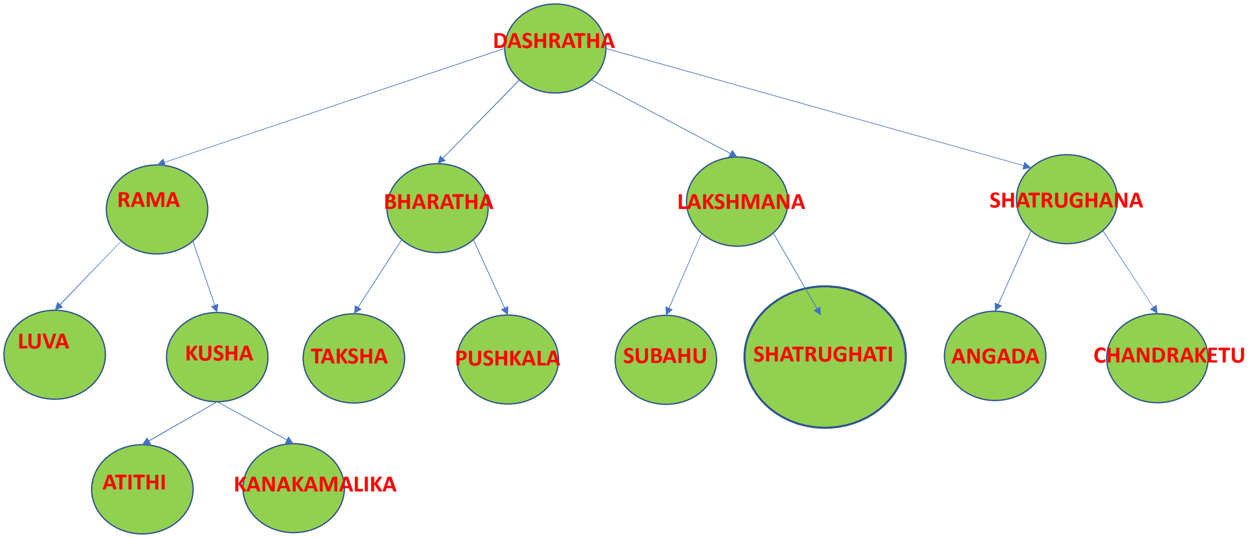
Tree

Tree structure

- The top element (node) of a non-empty tree is called the root node
- Each node (other than root) in the tree has exactly one parent node
- Each node in the tree has zero or more children nodes
- Nodes with the same parents are siblings
- Trees are drawn upside down

Family tree (Ramayana)

ROOT
PARENT
CHILDREN
SIBLING



An outgoing edge in every node connects a node to its children. The source of the edge is called the parent node. Every node has exactly one parent node except the root node, which has no parents. In this case, DASHRATHA is the root node. Two nodes with the same parents are called siblings, e.g., LUVA and KUSHA are siblings of each other.

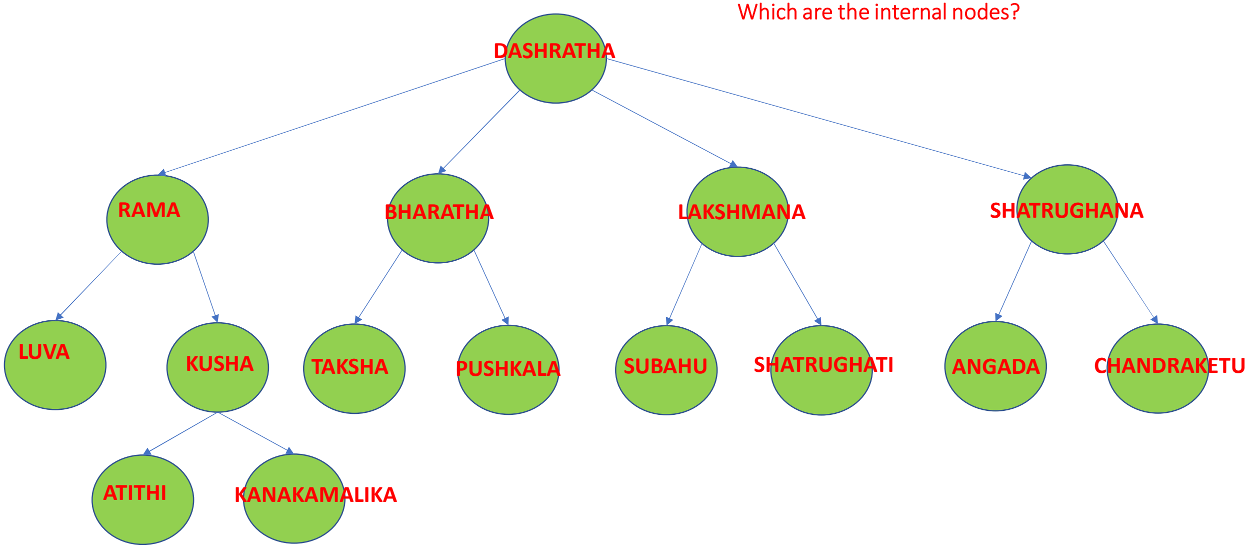
Terminology

- A node with no children is called a leaf node
- A node that is not a leaf is called an internal node

Family tree (Ramayana)

INTERNAL NODE
LEAF

Which are the leaf nodes?
Which are the internal nodes?



LUVA, ATITHI, KANAKAMALIKA, etc., are the leaf nodes because they don't have any children. RAMA, KUSHA, BHARATHA, etc., are internal nodes because they have at least one child.

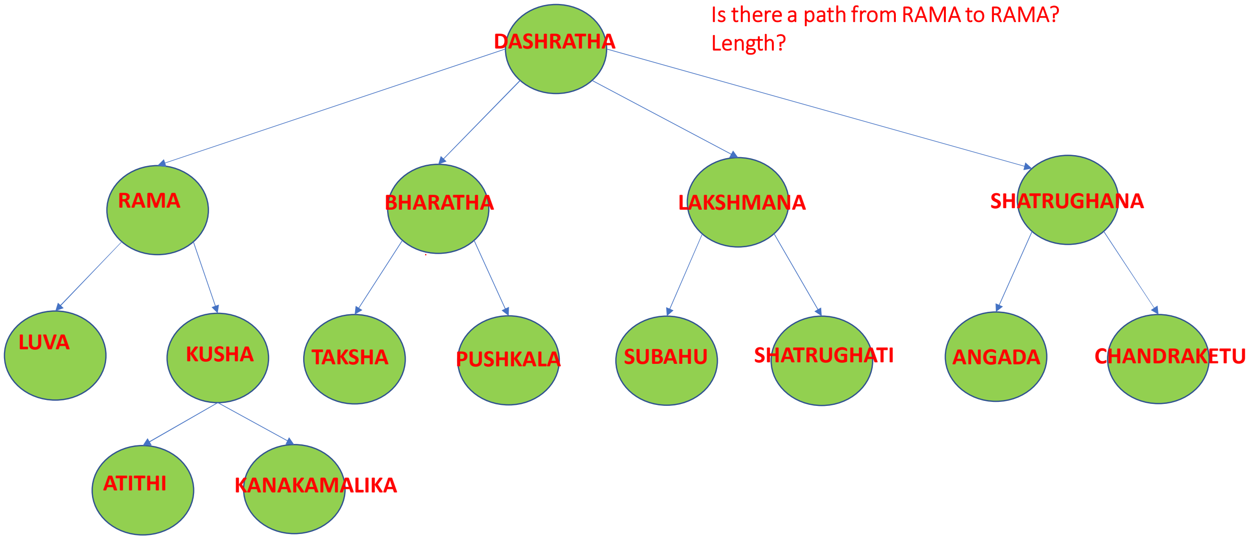
Terminology

- A path from node n_1 to n_k is defined as a sequence of nodes n_1, n_2, …, n_k such that n_i is the parent of n_{i+1} for all $1 <= i < k$
- The length of the path is the number of edges on the path
- There is a path of length zero from every node to itself
- There is at most one path between any pairs of nodes in a tree
- There is a path from the root to every other node in a tree

Family tree (Ramayana)

PATH

Is there a path from DASRATHA to ANGADA?
Is there a path from TAKSHA to PUSHKALA?
Is there a path from RAMA to RAMA?
Length?



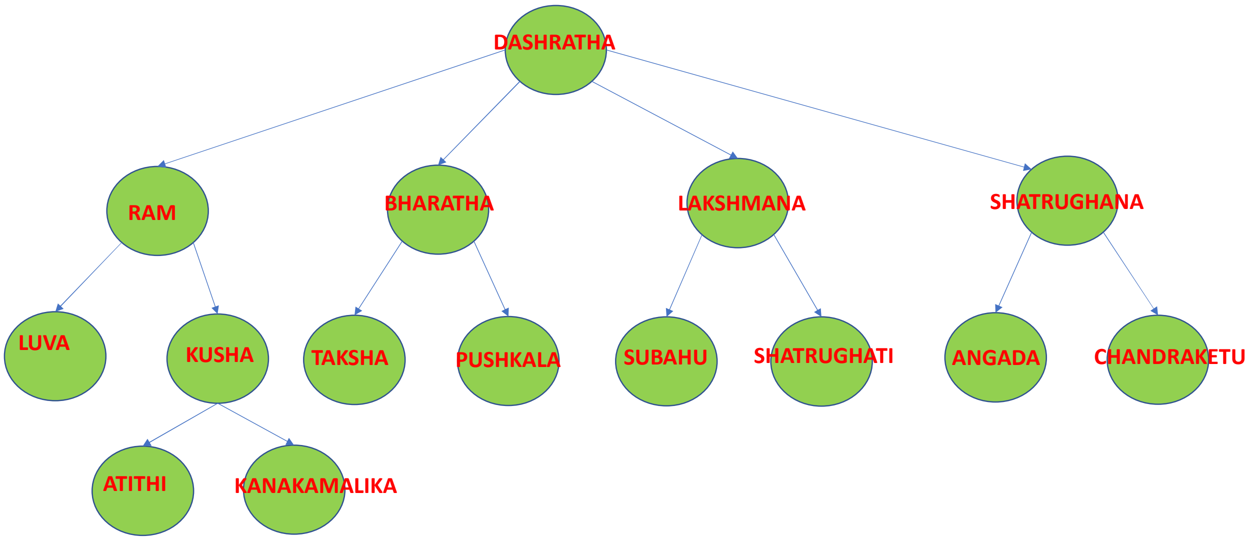
There is a path between DASRATHA and ANGADA, i.e., DASRATHA-SATRUGHANA-ANGADA. This is because DASRATHA is the parent of SATRUGHANA, and STARUGHANA is the parent of ANGADA. The length of this path is two, i.e., the number of edges in the path. There is no path between TAKSHA and PUSHKALA. TAKSHA-BHARATHA-PUSHKALA is not a path because TAKSHA is not a parent of BHARATHA. There is a path from RAMA to RAMA of length zero. There is a path from the root (DASHRATHA) to every other node in the tree.

Terminology

- If there is a path from n_1 to n_2, then n_1 is an ancestor of n_2 and n_2 is a descendant of n_1
    - If $n_1 \neq n_2$, n_1 is a proper ancestor of n_2 and n_2 is a proper descendant of n_1
    - Every node is an ancestor and descendant of itself

Family tree (Ramayana)

ANCESTOR
DESCENDANT
PROPER ANCESTOR
PROPER DESCENDANT



RAMA is an ancestor of ATITHI because there is a path from RAMA to ATITHI. ATITHI is an ancestor of himself; however, ATITHI is not a proper ancestor of himself.

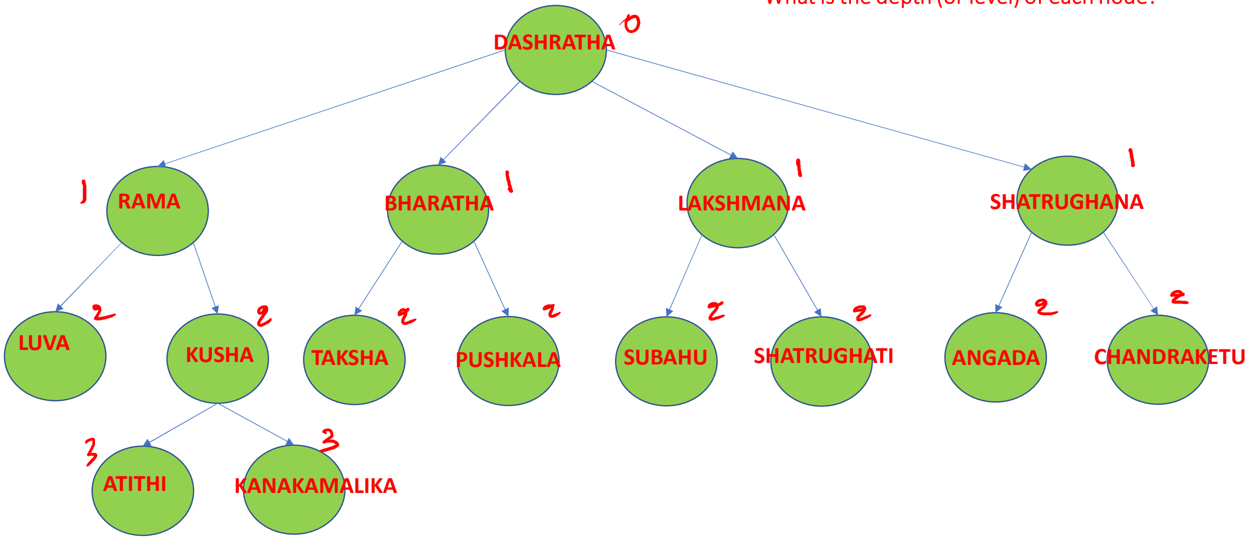
Terminology

- There is a path from the root to every other node in a tree
- The depth (or level) of a node is the length of the path from the root to itself
- The depth of the tree is the maximum of the depths of all nodes

Family tree (Ramayana)

PATH

What is the depth (or level) of each node?



The depth of ATITHI is 3, i.e., the length of the path from the root (DASHRATHA). The depths of the other nodes are also shown on this slide.

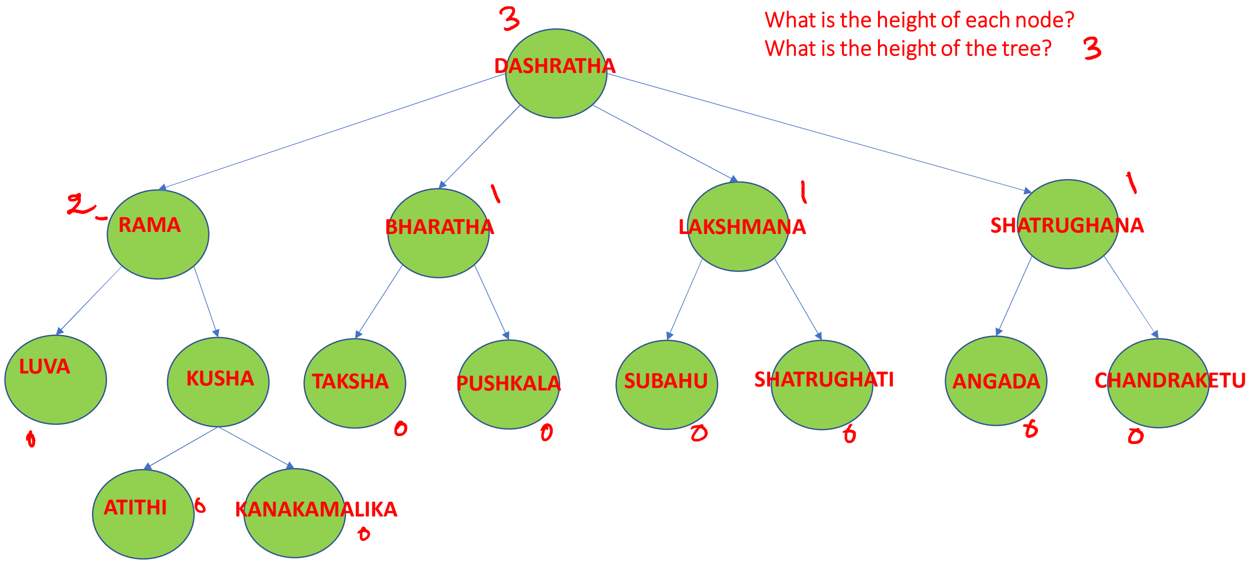
Terminology

- There is a path from every node to a leaf node
- The height of a node n is the length of the longest path from the n to a leaf
- All leaves are at the height of zero
- The height of the tree is equal to the height of the root
    - The height of an empty tree is -1

Family tree (Ramayana)

PATH

What is the height of each node?
What is the height of the tree? 3



The height of RAMA is two because there is a path from RAMA to leaf nodes LUVA, ATHITHI, and KANKAMALIKA. Out of these paths, the maximum length is two, and therefore two is the height of the node corresponding to RAMA. The height of the tree is the height of DASHRATHA, i.e., three.

Depth vs. Height

- The depth of a tree is equal to the height of the tree

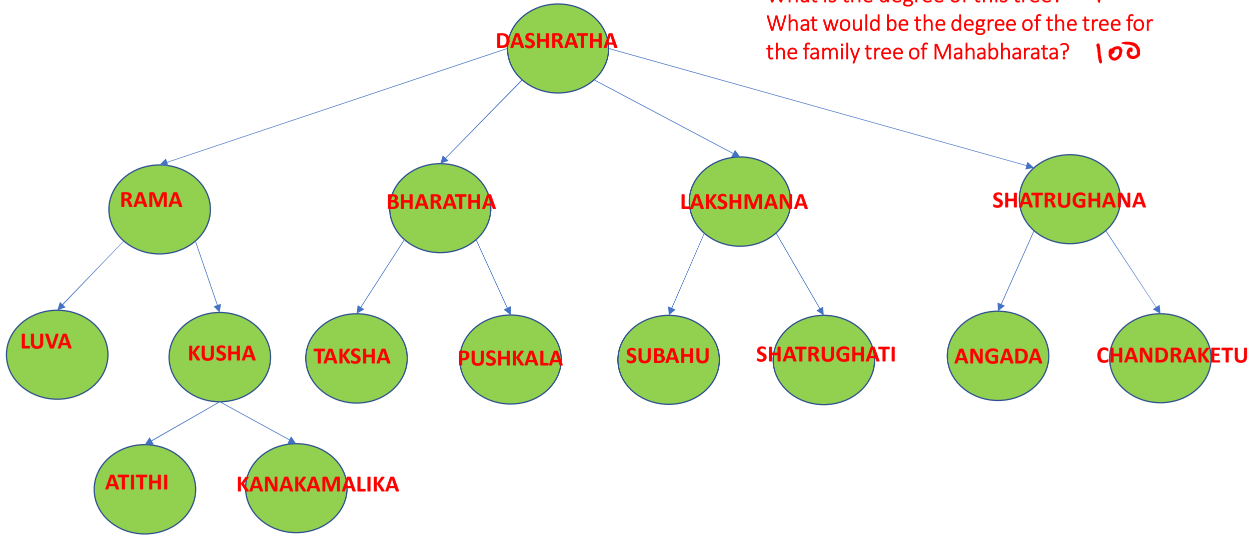
Terminology

- The maximum children in any node is called the degree of a tree
- Trees of degree two are called binary trees

Family tree (Ramayana)

PATH

What is the degree of this tree? 4
What would be the degree of the tree for the family tree of Mahabharata? 100



The degree of this tree is four because DASRATHA has the maximum number of children, i.e., four. If we draw the family tree of Mahabharata, the degree would be 100 because DHRITRASTRA had 100 children.

Binary tree

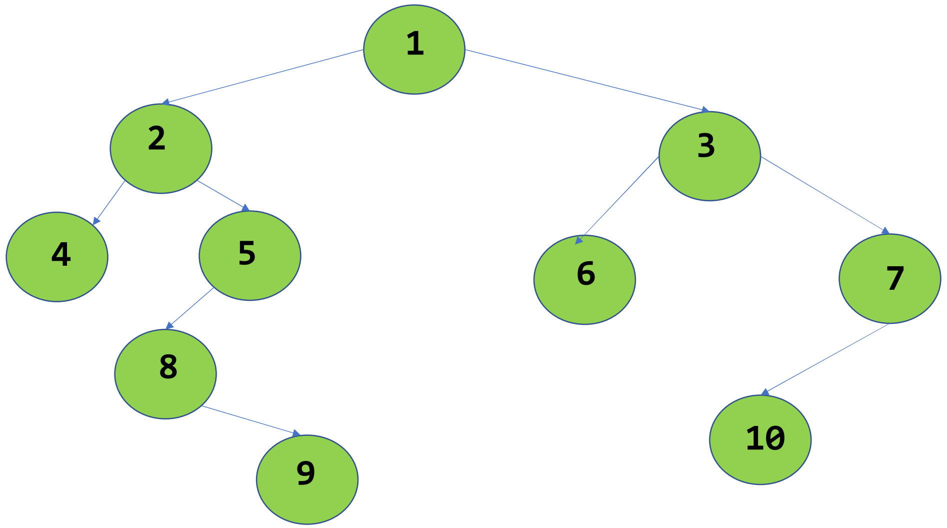
Binary tree

- The degree of a binary tree is two

- Recursive definition
  - A binary tree is either empty or consists of
    - a root node
    - two distinct binary trees called the left and right subtrees
    - root contains references to (or addresses of) left and right subtrees

Binary tree

DEGREE?



This is a binary tree because the maximum number of children in any node is two.

Type (Binary Tree)

```
struct node {
    int val;
    struct node * left;
    struct node * right;

};
```

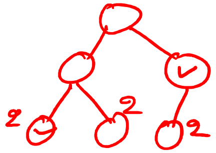
Complete binary tree

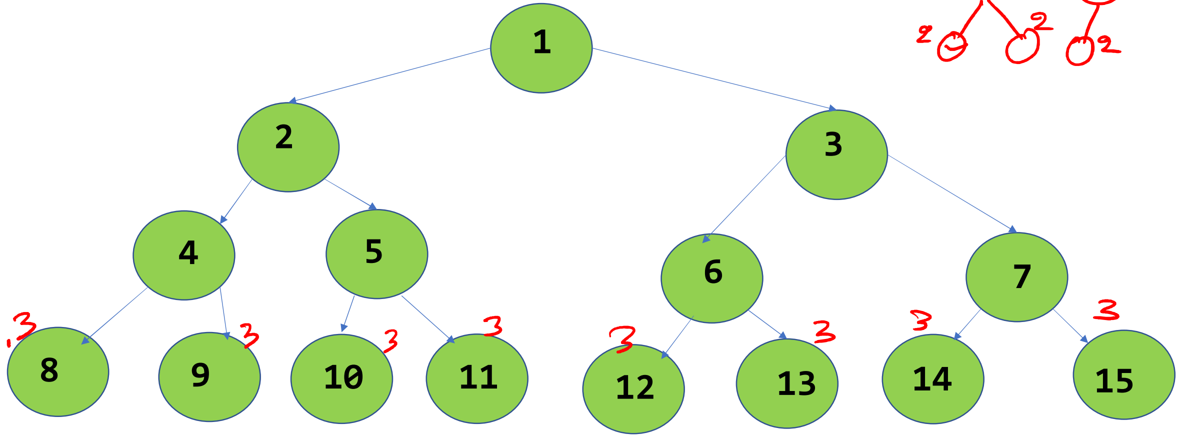
- In a complete binary tree, all the leaves have the same depth

- All the internal nodes have degree two

Complete binary tree

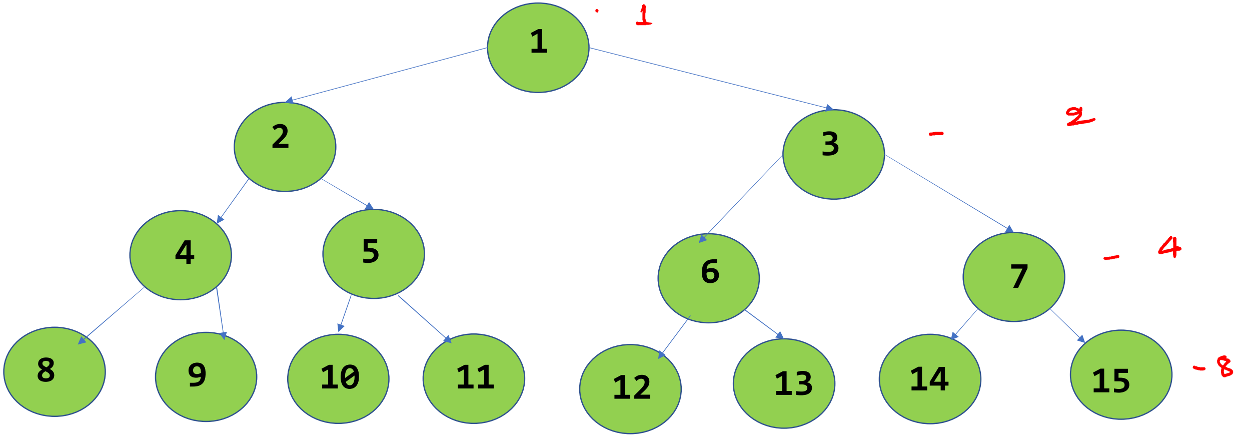
Why is this a complete binary tree?





This is a complete binary tree because all leaves are at the same depth (or level). All internal nodes have two children.

Height = f(number of nodes)



Height = f(number of nodes)

Max depth is h .

At depth 0 what are the number of nodes 1

At depth 1 2

At depth 2 2^2

At depth 3 2^3

- - -

At depth h 2^h

$$1 + 2 + 2^2 + \dots + 2^h = n$$

$$2^{h+1} - 1 = n$$

$$h = (\log_2(n+1)) - 1$$

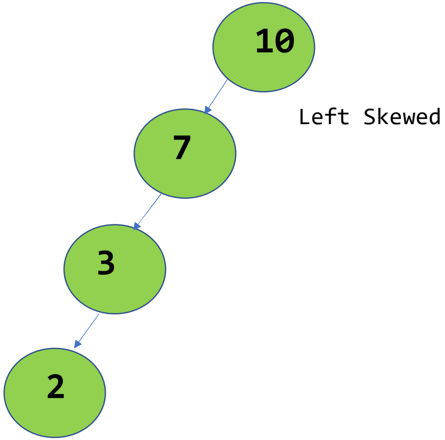
Height = f(number of nodes)

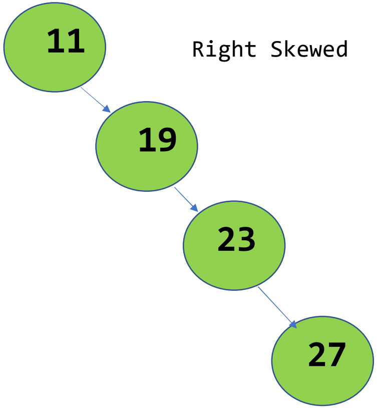
Let’s say the maximum depth (of height) of the tree is h.
Number of nodes at depth 0: 1
Number of nodes at depth 1: 2
Number of nodes at depth 2: 2^2
….
Number of nodes at depth h: 2^h
If the total number of nodes in the tree is n

$$1 + 2 + 2^2 + \cdots + 2^h = n$$
$$2^{h+1} - 1 = n$$
$$h = (\log_2(n + 1)) - 1$$

Skewed tree

- In a skewed binary tree, all the nodes have at most one child





Tree traversal

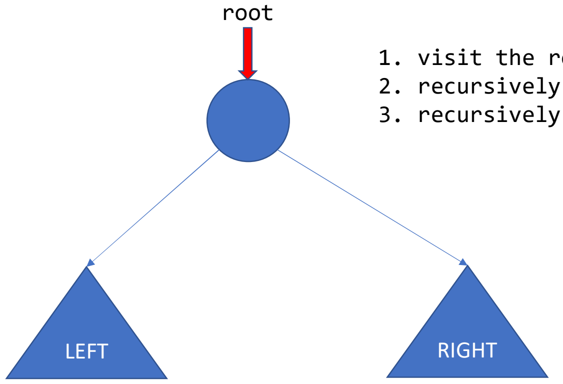
Tree traversal

- The aim of the traversal is to visit each node in the tree exactly once

- Generally, we traverse the tree in the following orders
  - Preorder traversal
  - Inorder traversal
  - Postorder traversal

- The above algorithms are recursive in nature

Preorder traversal

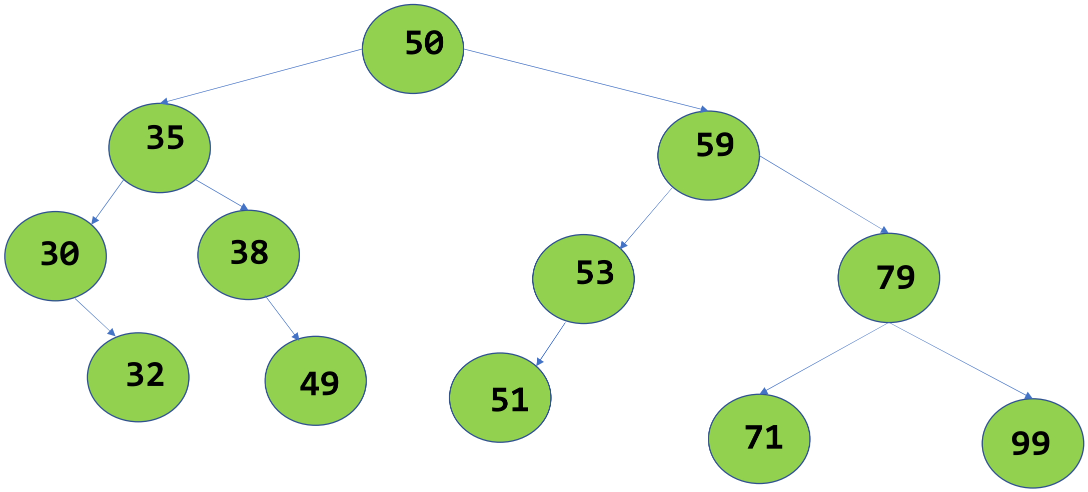


1. visit the root node
2. recursively traverse the left subtree
3. recursively traverse the right subtree

Preorder

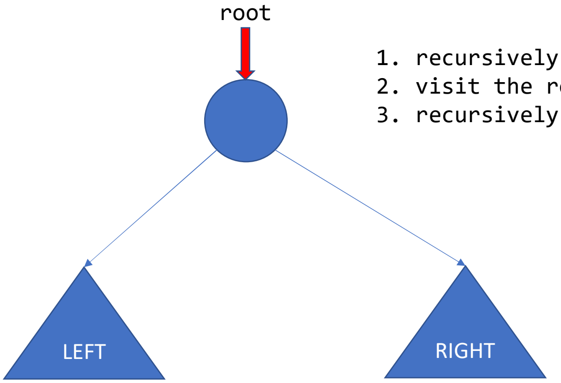
PREORDER: ROOT LEFT RIGHT

50 35 30 32 38 49 59
53 51 79 71 99



In preorder traversal, we can first print (or traverse) the node and then recursively traverse the left subtree followed by the right subtree. To traverse this tree, we first print 50 and then traverse the left subtree rooted at 35. To traverse the tree rooted at 35, we first print 35 and then traverse the tree rooted at 30. To traverse the tree rooted at 30, we first print 30 and then traverse the empty left subtree. After we are done with the left subtree, we traverse the right subtree of 30 to traverse the tree rooted at 30 completely. The right subtree of 30 is 32. Traversing the subtree rooted at 32 prints 32. We continue this until we have completely traversed the root of the original tree, i.e., 50. The result of the preorder traversal is shown on this slide.

Inorder traversal

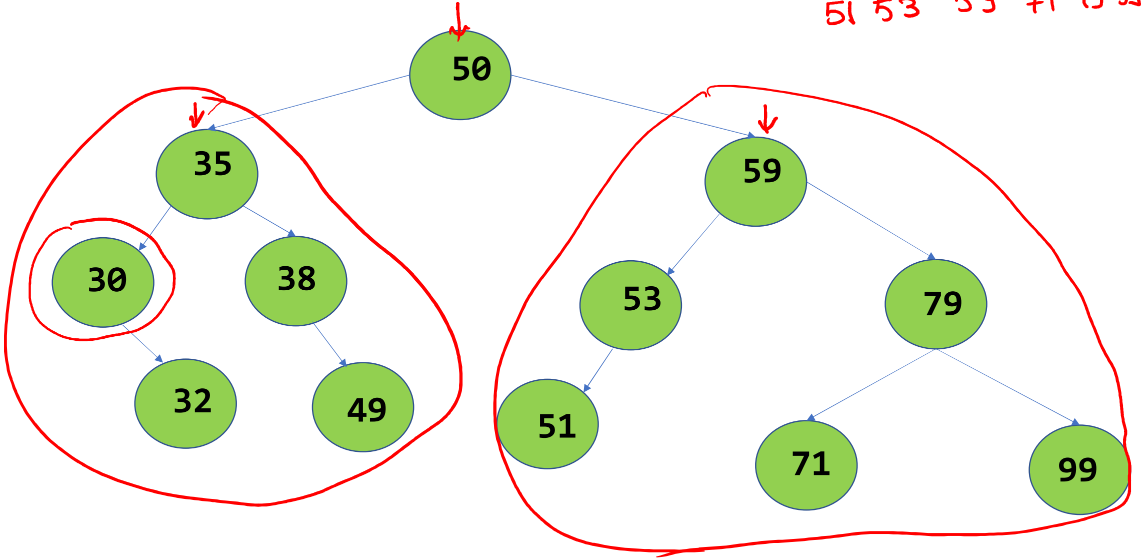


```
1. recursively traverse the left subtree
2. visit the root node
3. recursively traverse the right subtree
```

Inorder

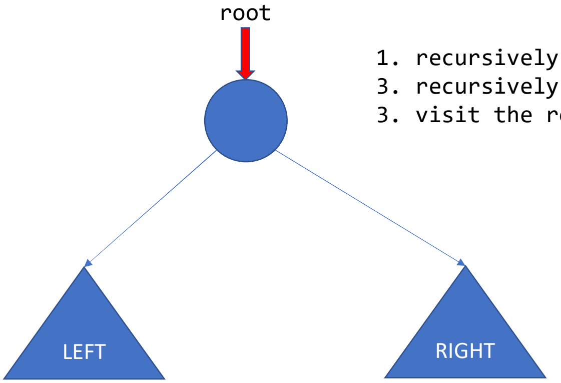
INORDER: LEFT ROOT RIGHT

30 32 35 38 49 50
51 53 59 71 79 99



In inorder traversal, we first print (or traverse) the left subtree before printing the value of the root node. After that, we traverse the right subtree. The output of the inorder traversal is shown on this slide.

Postorder traversal

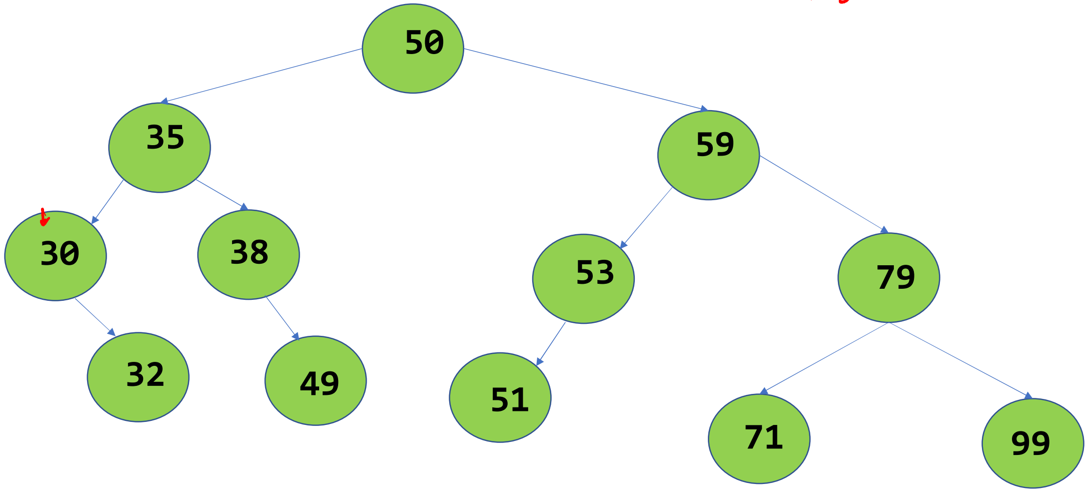


```
1. recursively traverse the left subtree
3. recursively traverse the right subtree
3. visit the root node
```

Postorder

POSTORDER: LEFT RIGHT ROOT

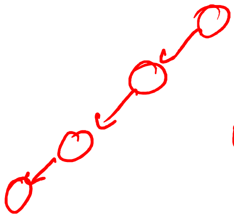
32 30 49 38 35 51 53
71 99 79 59 50



In postorder traversal, we first print (or traverse) the left subtree followed by the right subtree before finally printing the value of the root. The result of the postorder traversal is shown on this slide.

Preorder traversal

```
void preorder(struct node *root) {
  if (root == NULL) {
    return;
  }
  printf("%d ", root->val);
  preorder(root->left);
  preorder(root->right);
}
```



20   n^2

10   10

100   100

=200

1   19

1 + $(19)^2$

Time complexity:

$T(n) = 2T(\frac{n}{2}) + C$ $O(n)$

$T(n) = T(n-1) + C$ $O(n)$

This is the recursive algorithm for doing the postorder traversal. The time complexity can't be better than O(n) because we need to visit all the nodes at least once. If both the left and right subtrees always contain roughly the same number of nodes (i.e., n/2), the time complexity would be T(n) = 2T(n/2) + c, which is O(n). In the worst case, we will always encounter a skewed tree, i.e., the number of nodes in the left-subtree is n-1, and the right subtree is empty, or vice-versa; in this case, the time complexity T(n) = T(n-1) + c, which is again O(n).

Inorder traversal

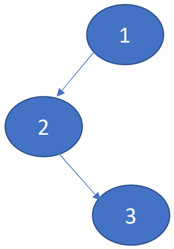
```
void inorder(struct node *root) {
  if (root == NULL) {
    return;
  }
  inorder(root->left);
  printf("%d ", root->val);
  inorder(root->right);
}
```

Postorder traversal

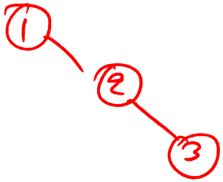
```
void postorder(struct node *root) {
  if (root == NULL) {
    return;
  }
  postorder(root->left);
  postorder(root->right);
  printf("%d ", root->val);
}
```

Reconstructing binary tree

- Given preorder traversal, can we reconstruct the original binary tree



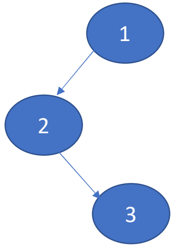
1 2 3



We can not reconstruct the original tree from a given preorder traversal because multiple trees can have the same preorder traversal.

Reconstructing binary tree

- Given inorder traversal, can we reconstruct the original binary tree



2 3 1

We can not reconstruct the original tree from a given inorder traversal because multiple trees can have the same inorder traversal.

Exercise

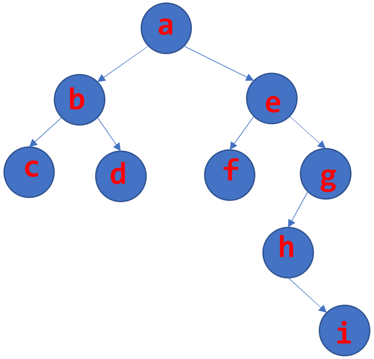
- Write an algorithm to reconstruct the unique tree from given inorder and preorder traversals (assuming no duplicates)
- Write an algorithm to reconstruct the unique tree from given inorder and postorder traversals (assuming no duplicates)
- Give an example to demonstrate that we can't reconstruct a unique tree from given preorder and postorder traversals (assuming no duplicates)

Tree algorithms

Height of a binary tree

```
1 + max( height(root->left), height (root->right))

if (root == NULL)
    return -1;
```



Let's say the heights of the left subtree and the right subtree are lh and rh; the height of the tree would be one more than the maximum of lh and rh.

Height of a binary tree

- Recursive step
  - `h_left` = Recursively calculate the height of the left subtree
  - `h_right` = Recursively calculate the height of the right subtree
  - the height of the tree is `1 + max(h_left, h_right)`
- Base case
  - if the tree is empty, `return -1`

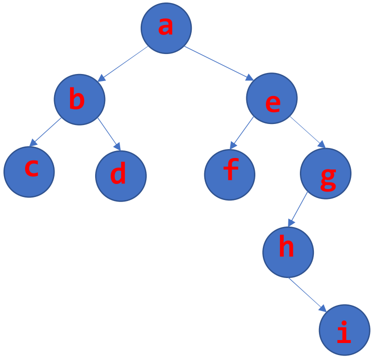
Height of a binary tree

```
int height(struct node *root)
{
  if (root == NULL) {
    return -1;
  }
  return 1 + max(height(root->left), height(root->right));
}
```

Count the number of nodes

```
1+ count(root->left) + count(root->right)

if (root == NULL)
    return 0;
```



The total number of nodes in a tree is one more than the sum of the total number of nodes in the left subtree and the total number of nodes in the right subtree plus one.

Count the number of nodes

- Recursive step
  - `l_count` = recursively count the number of nodes in the left subtree
  - `r_count` = recursively count the number of nodes in the right subtree
  - The total number of nodes is `l_count + rcount + 1`

- Base case
  - If the tree is empty, `return 0`

Count the number of nodes

```
int count(struct node *root) {
  if (root == NULL) {
    return 0;
  }
  return 1 + count(root->left) + count(root->right);
}
```

Binary search tree

Binary search tree (BST)

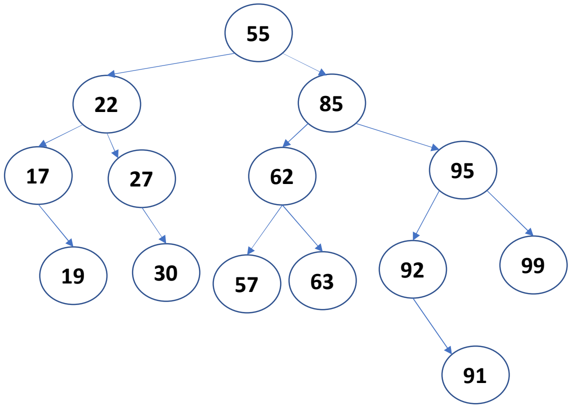
- BST is a binary tree and has the following properties

- The values of all the keys in the left subtree are less than the value of the key stored at root

- The values of all the keys in the right subtree are more than the value of the key stored at root

- The left and right subtrees in a BST are BSTs

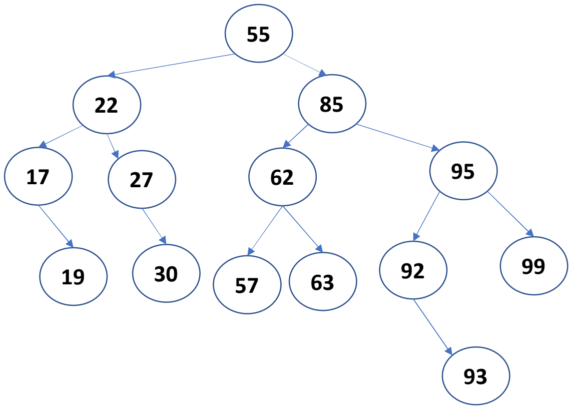
BST



Is this a BST? No

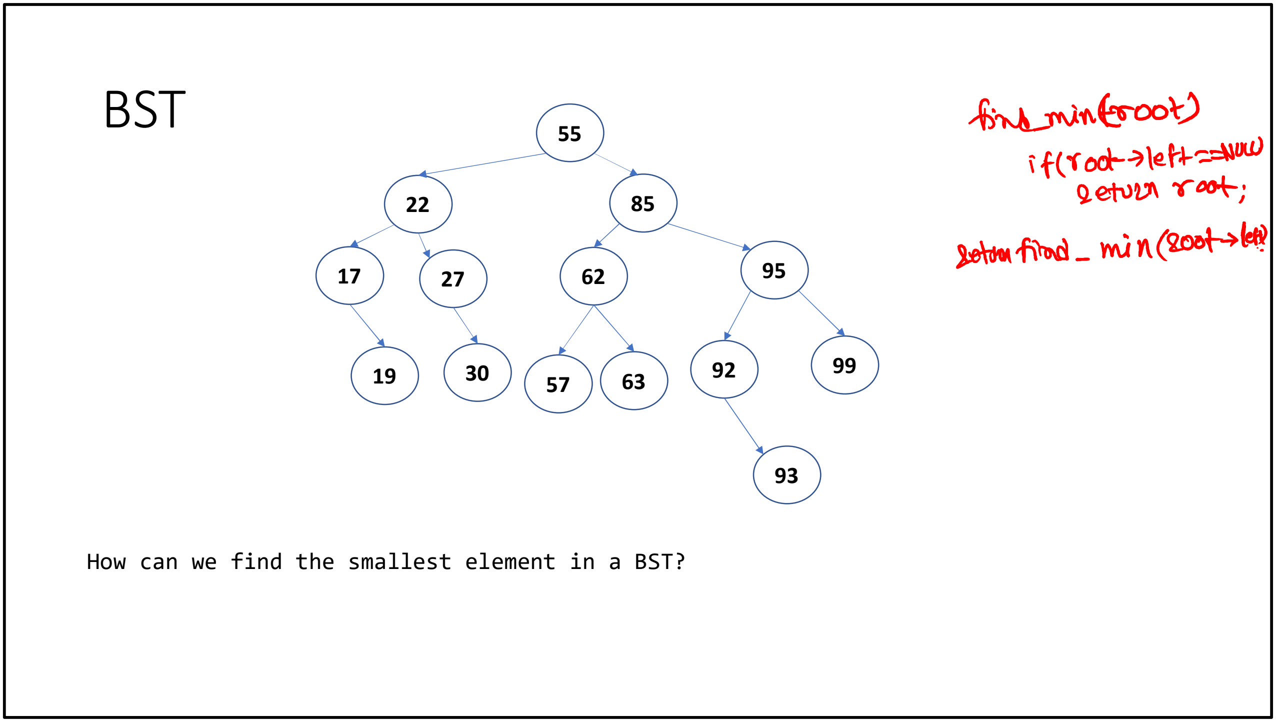
All the keys on the right of 55 are more than 55, and all the keys on the left of 55 are less than 55; therefore, this tree is BST if both the left and right subtrees are also BST. Let’s look at the right subtree. All the nodes on the right of 85 contain larger keys, and all the nodes on the left of 85 contain smaller keys; therefore, the tree rooted at 85 is a BST if both the left and right subtrees are also BST. Now look at 95, all the keys on the right are greater, and all the nodes on the left are lesser; therefore, it could be a BST if the left and right subtrees are also BSTs. Let’s look at 92; the node on the right is less than 92; therefore, 92 is not a BST. This also means that the trees rooted at 95, 85, and 55 are not BSTs.

BST

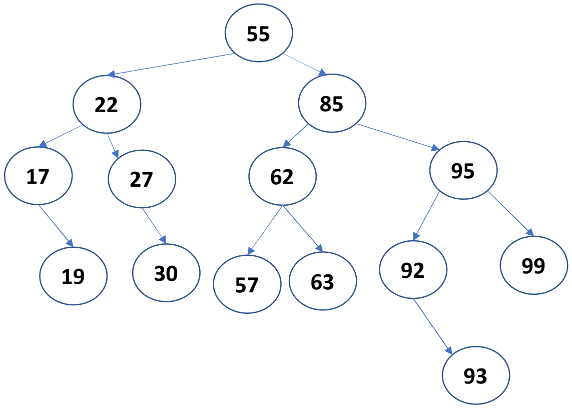


Is this a BST? Yes

This is the BST because it satisfies all properties of a BST.



BST

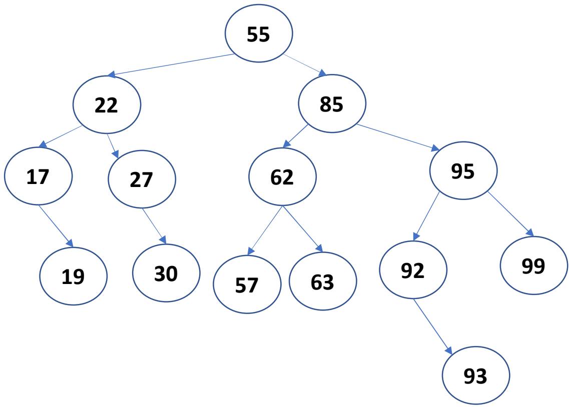


```
find_min(root)
    if(root->left==NULL)
        return root;
return find_min(root->left)
```

How can we find the smallest element in a BST?

If the left subtree is empty, root must be the smallest element because of the property of a BST. Otherwise, the smallest element must be the smallest in the left subtree because all the keys in the left subtrees are smaller. Similarly, the smallest element in the left subtree is the smallest element in the left subtree of the left subtree. If we go this way, the smallest element would be the leftmost element. To find the smallest element, we need to go from the root of the tree to a node that is the leftmost leaf node or the parent of leaf nodes. Therefore, the time complexity would be the height of the tree. Because, in the worst case, the height can be n-1, where n is the number of nodes in the tree, therefore the time complexity is O(n).

BST



```
How can we find the largest element in a BST?
```

The largest element is the rightmost element.

BST ADT

- Search(T, X): finds a node with key X in the BST rooted at T
- Insert(T, X): Inserts a new node with value X in the BST rooted at T
- Delete(T, X): Deletes a node with key X in the BST rooted at T
- FindMin(T): Returns a node with the minimum key value
- FindMax(T): Returns a node with the maximum key value

BST

- In our future discussions, we are assuming that each node in the BST contains an integer value that is also the key
- In general, a BST may contain any user-defined type with a combination of one or more fields used as the key
- In addition to defining the key, we also need to provide a way to compare two keys

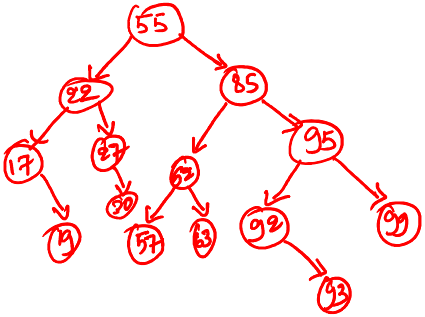
Insertion

Insertion

- The insert routine inserts a new node in the BST with a given input value

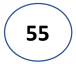
BST insertion

55 22 85 95 92 17 99 62 93 19 27 57 63 30



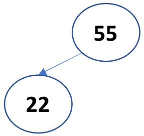
We next show how to insert nodes one by one in a BST. If the key we are inserting is larger than the root node, then we insert it in the right subtree; otherwise, we insert it in the left subtree. If the tree is empty, we create a new node with the given value and return the address of that node.

BST insertion



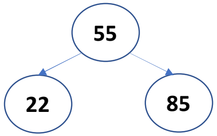
55 22 85 95 92 17 99 62 93 19 27 57 63 30

BST insertion



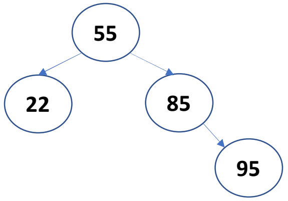
55 22 85 95 92 17 99 62 93 19 27 57 63 30

BST insertion



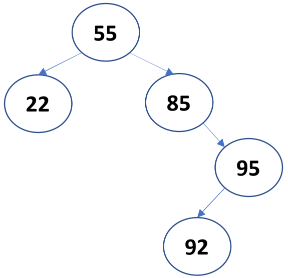
55 22 85 95 92 17 99 62 93 19 27 57 63 30

BST insertion



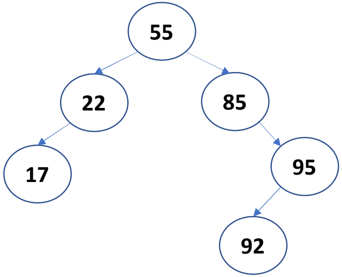
55 22 85 95 92 17 99 62 93 19 27 57 63 30

BST insertion



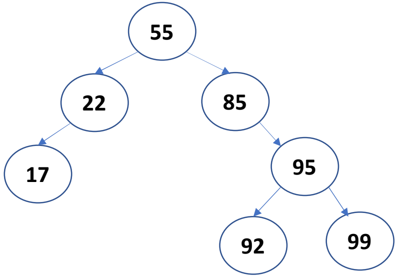
55 22 85 95 92 17 99 62 93 19 27 57 63 30

BST insertion



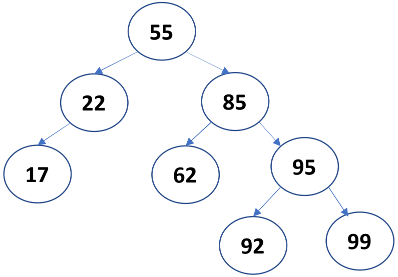
55 22 85 95 92 17 99 62 93 19 27 57 63 30

BST insertion



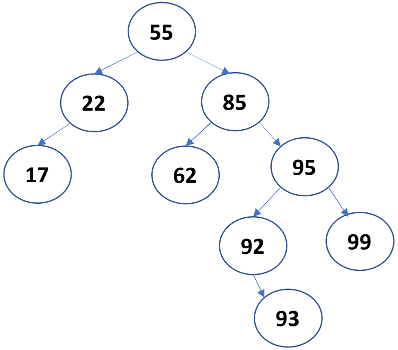
55 22 85 95 92 17 99 62 93 19 27 57 63 30

BST insertion



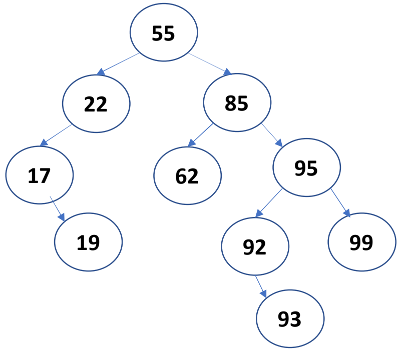
55 22 85 95 92 17 99 62 93 19 27 57 63 30

BST insertion



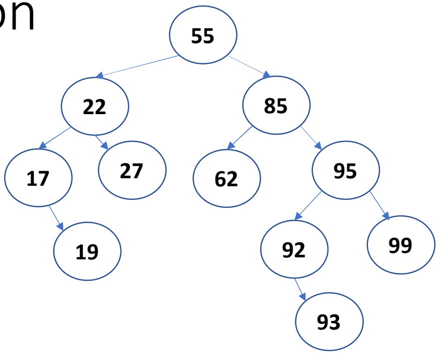
55 22 85 95 92 17 99 62 93 19 27 57 63 30

BST insertion



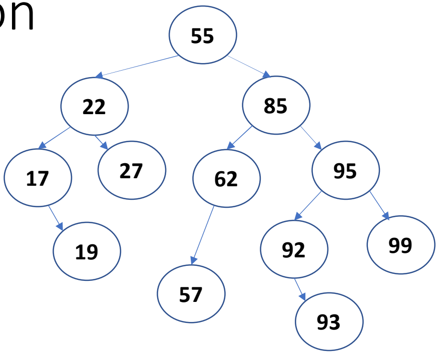
55 22 85 95 92 17 99 62 93 19 27 57 63 30

BST insertion



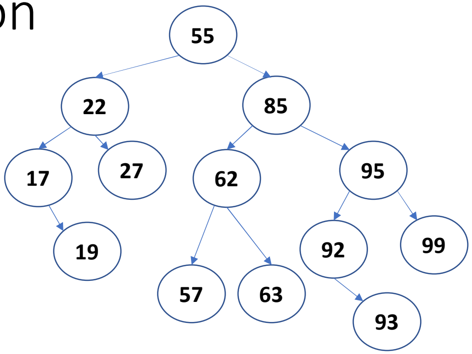
55 22 85 95 92 17 99 62 93 19 27 57 63 30

BST insertion



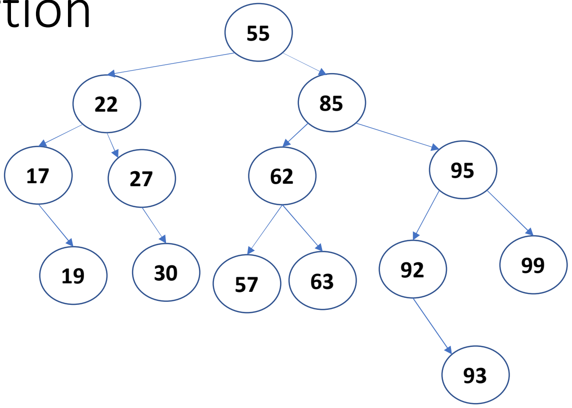
55 22 85 95 92 17 99 62 93 19 27 57 63 30

BST insertion



55 22 85 95 92 17 99 62 93 19 27 57 63 30

BST insertion



55 22 85 95 92 17 99 62 93 19 27 57 63 30

BST insertion (recursive algorithm)

- insert returns the root (possibly new) of the tree after inserting a node with a given value (val) (we are assuming val is also the key)
- Recursive step

```
if (val < root->val) {
    root->left = insert(root->left, val);
else
    root->right = insert(root->right, val);
```

- Base step

```
if (root == NULL)
    return allocate_node(val);
```

This is the recursive algorithm for the insertion. If the tree is empty, we create a new node with the given value and return its address; otherwise, if the key is less than root, we insert the node in the left subtree and store the possibly updated root of the left subtree (because of the insertion) in root->left. If the key is greater, we insert it in the right subtree and store the possibly updated root in root->right.