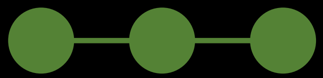
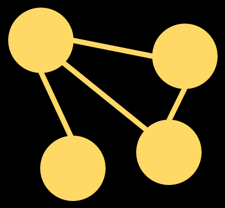
Trees

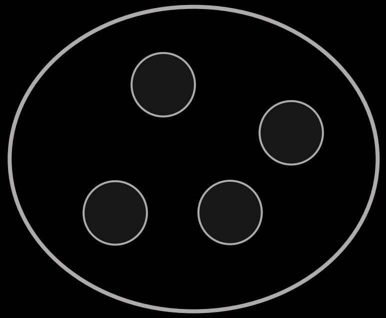
Data Structures & Algorithm

Categories of Data Structures

Linear Ordered	Non-linear Ordered	Not Ordered
Lists	Trees	Sets
Stacks	Graphs	Tables/Maps
Queues		

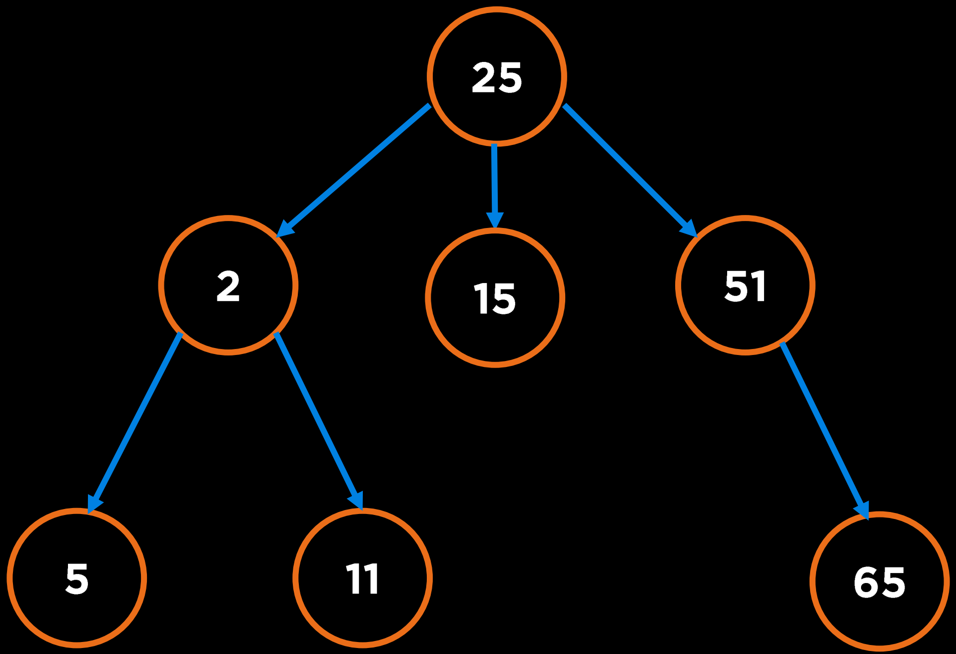






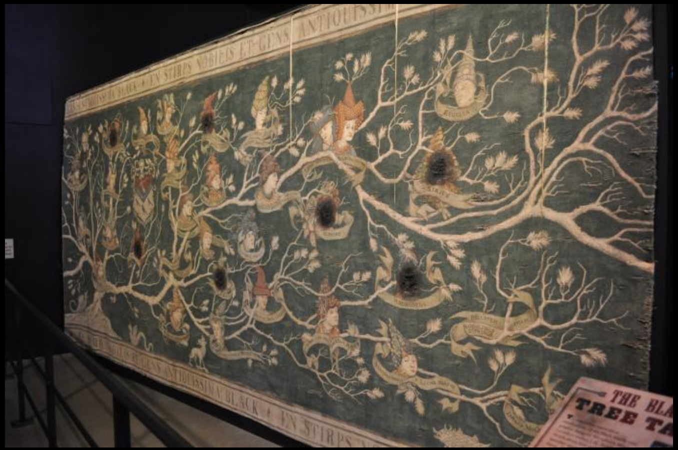
Trees

A tree is a rooted, directed, acyclic structure. It has three properties: one root; each node has one parent; and no cycles.

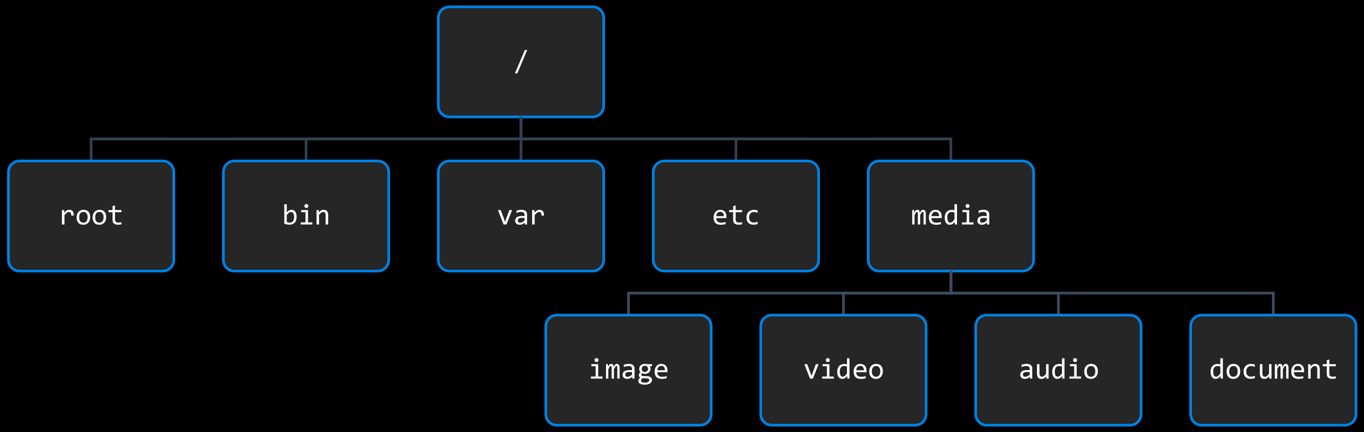


Use Cases

- Family Trees
- Decision Trees
- File Systems
- Expression Trees
- Search Trees



https://commons.wikimedia.org/wiki/File:Black_family_tapestry_as_seen_at_Harry_Potter_Experience.jpg

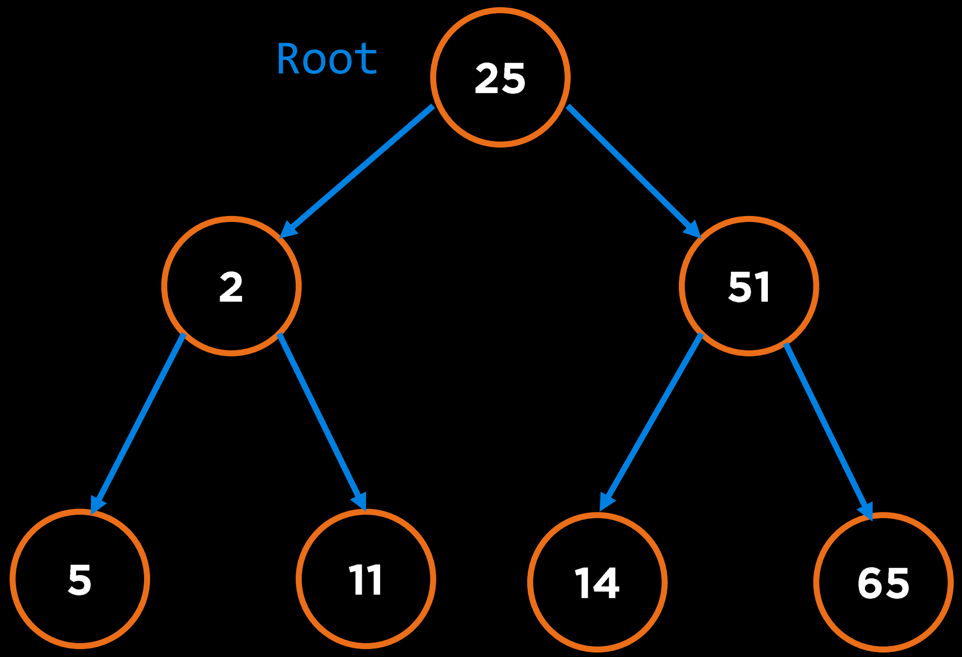


Tree Terminology

Trees: Terminology

Root

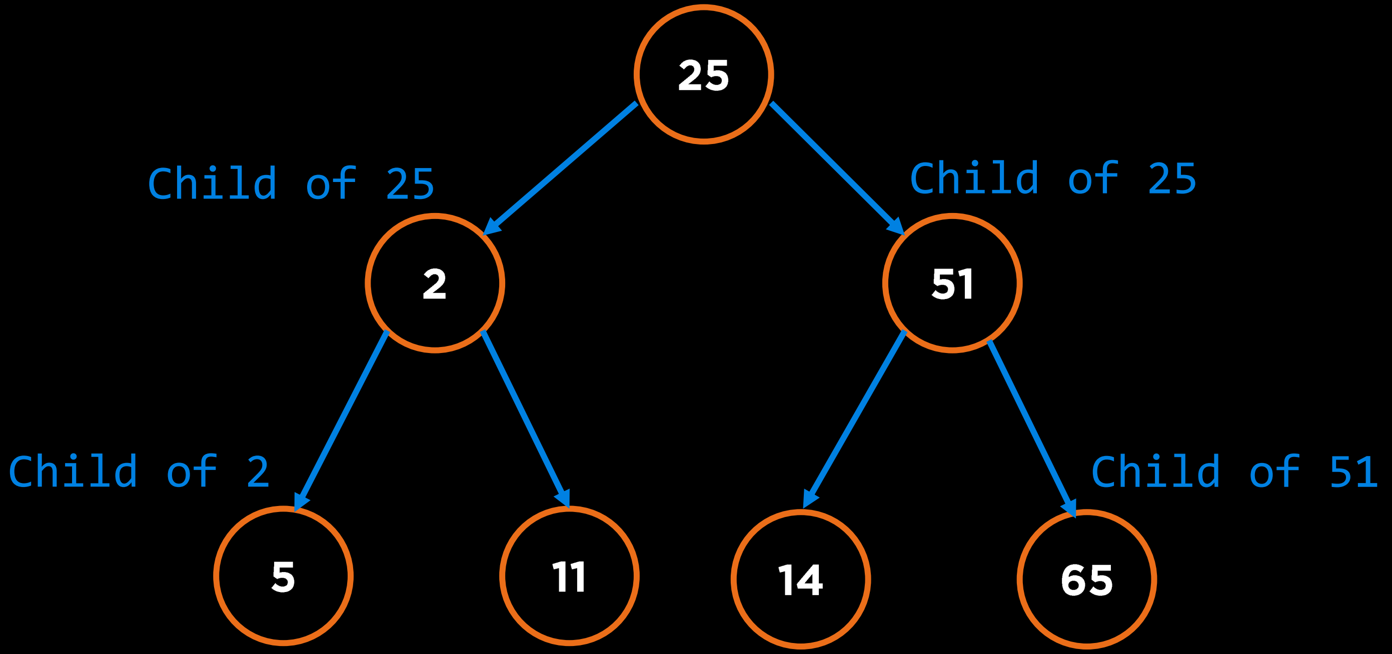
The node at the top is called the root.



Trees: Terminology

Children

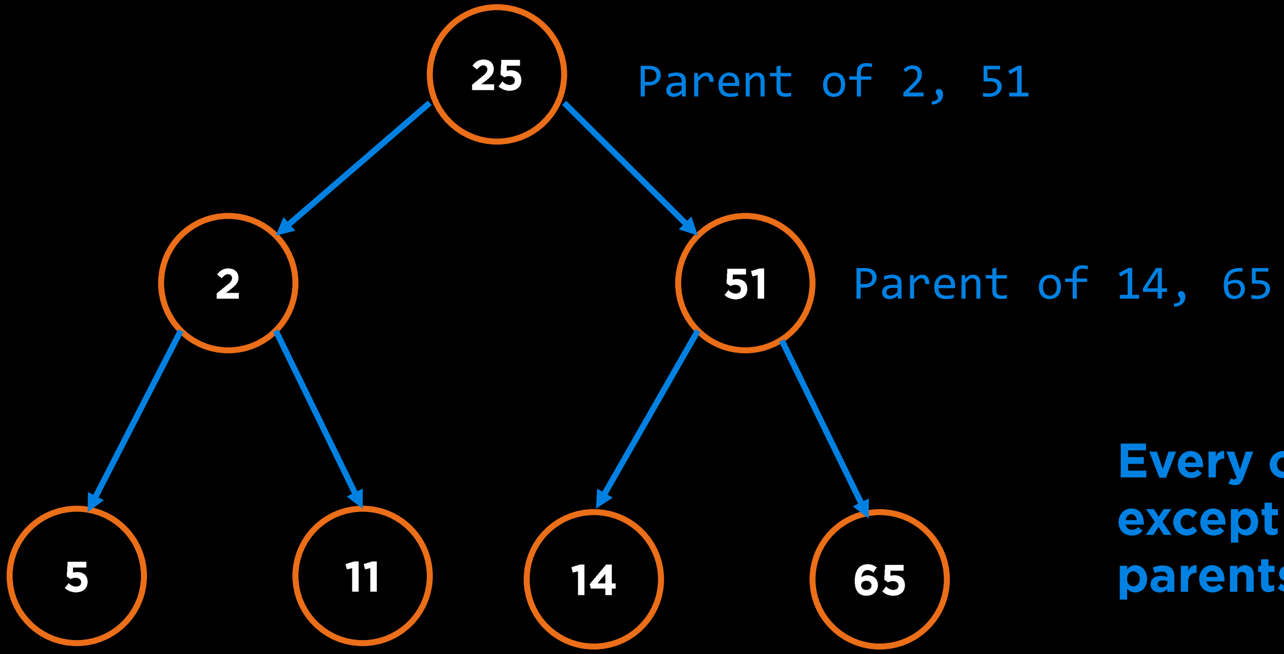
Successors of a node are called children.



Trees: Terminology

Parent

Predecessors of a node are called parent.

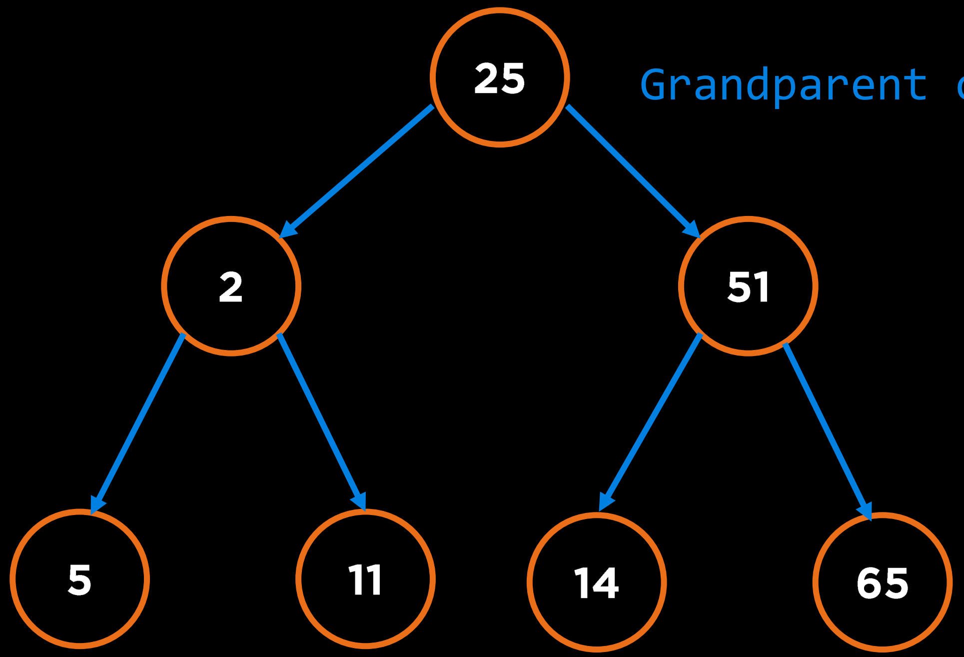


Every child has one parent except the root. Root has no parents.

Trees: Terminology

Grandparent

Predecessors of a node's parent are called grandparent.

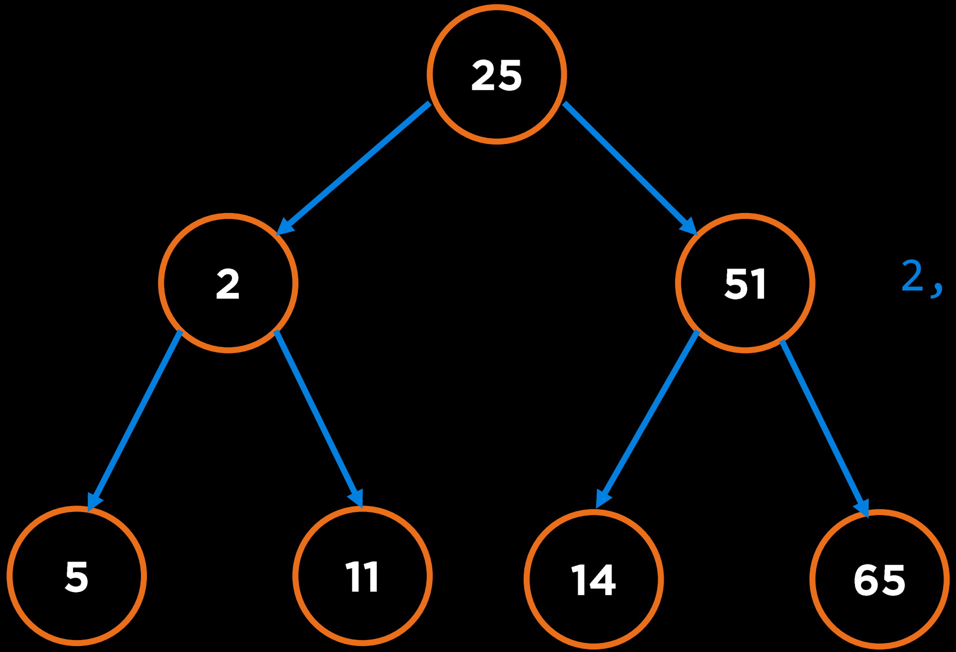


`Grandparent of 14, 65, 5, 11`

Trees: Terminology

Sibling

All nodes that have the same parent node are siblings.

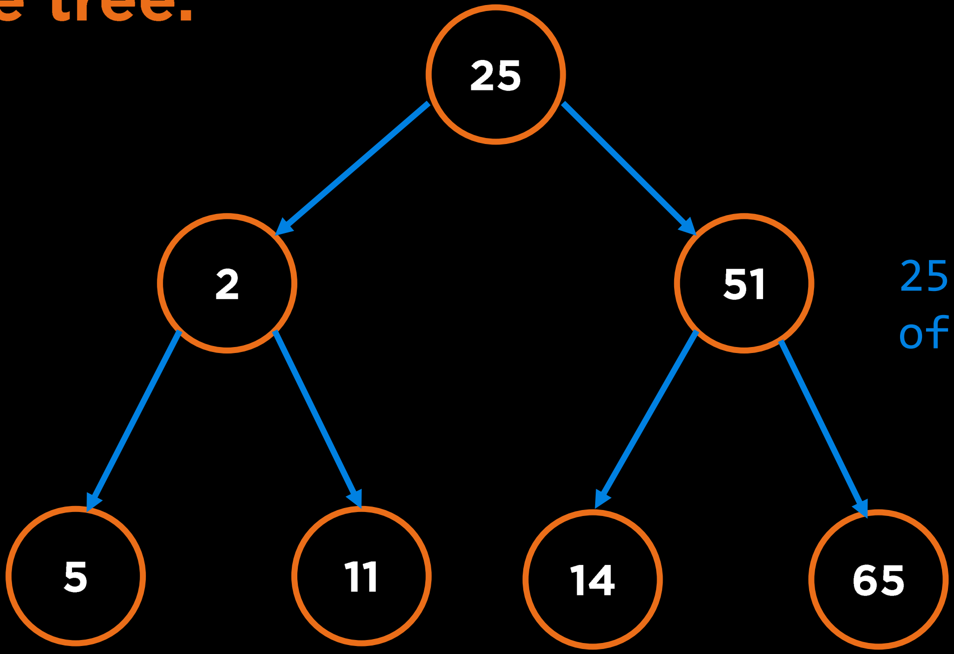


2, 51 are siblings

Trees: Terminology

Ancestor

All nodes that can be reached by moving only in an upward direction in the tree.

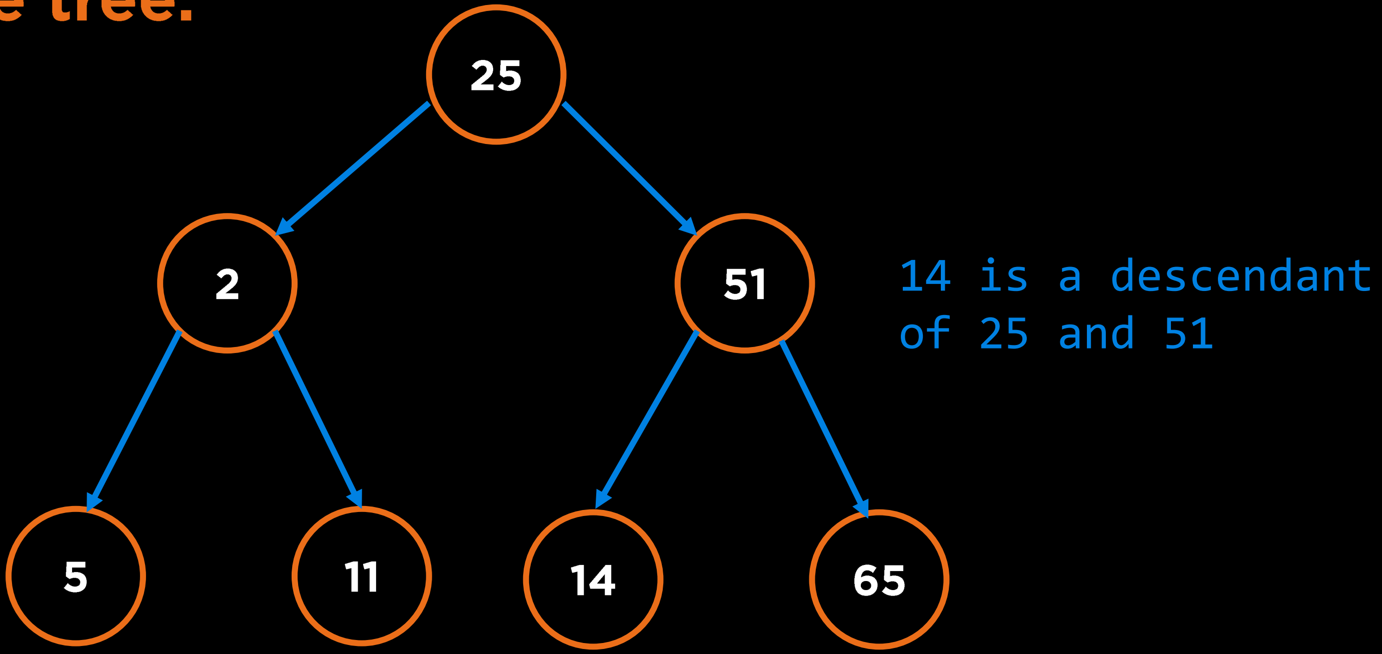


25, 51 are ancestors of 14

Trees: Terminology

Descendent

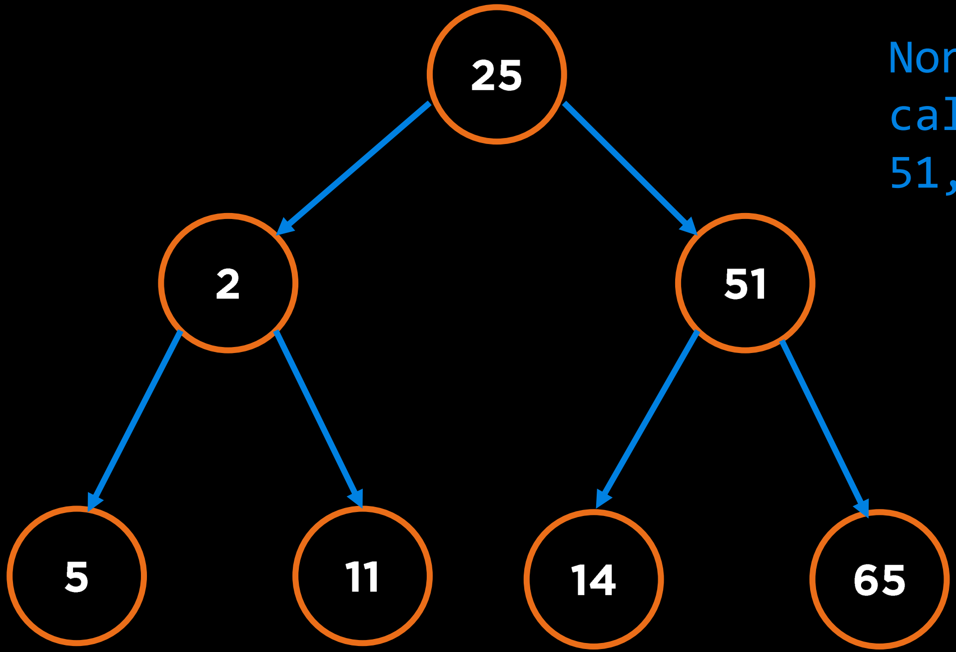
Nodes that can be reached by moving only in a downward direction in the tree.



Trees: Terminology

Leaf

Nodes with no children are called leaf nodes or external nodes.



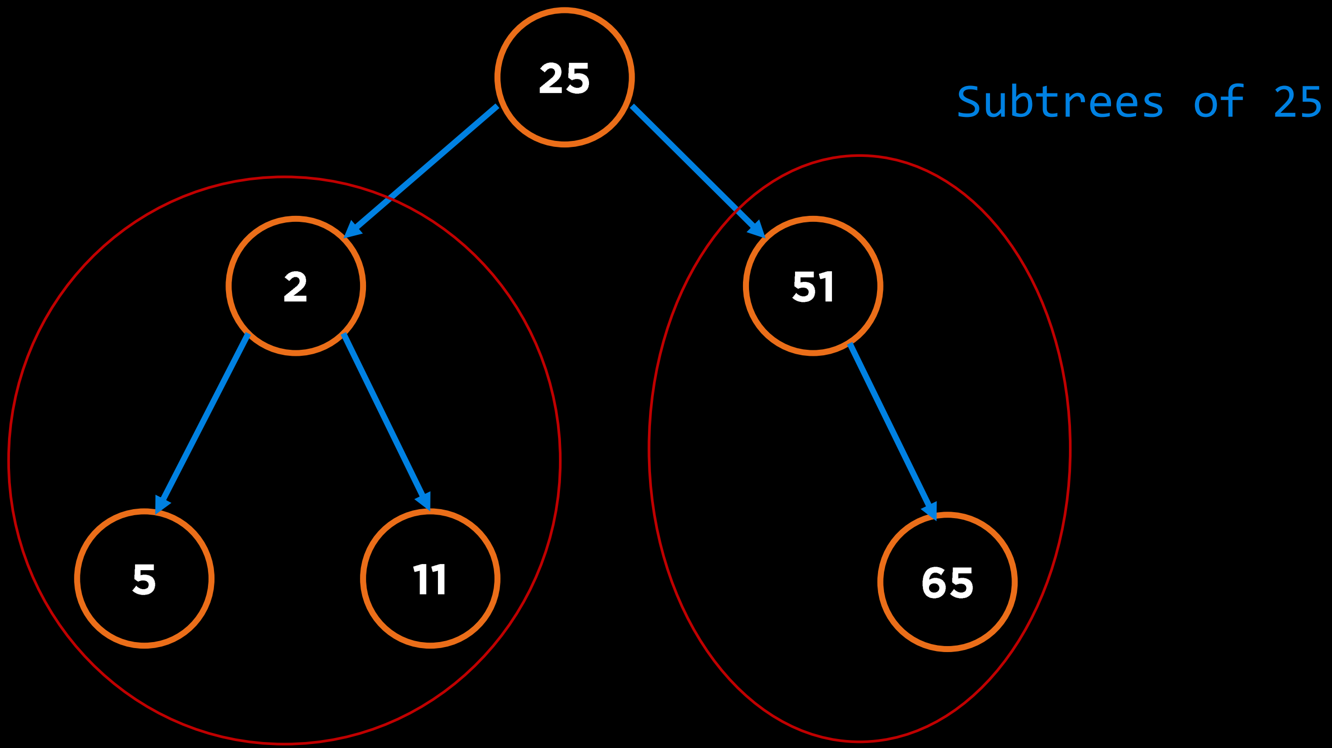
5, 11, 14, 65 are Leaf Nodes

Non-leaf nodes are also called internal nodes. 25, 51, 2 are internal nodes.

Trees: Terminology

Subtree

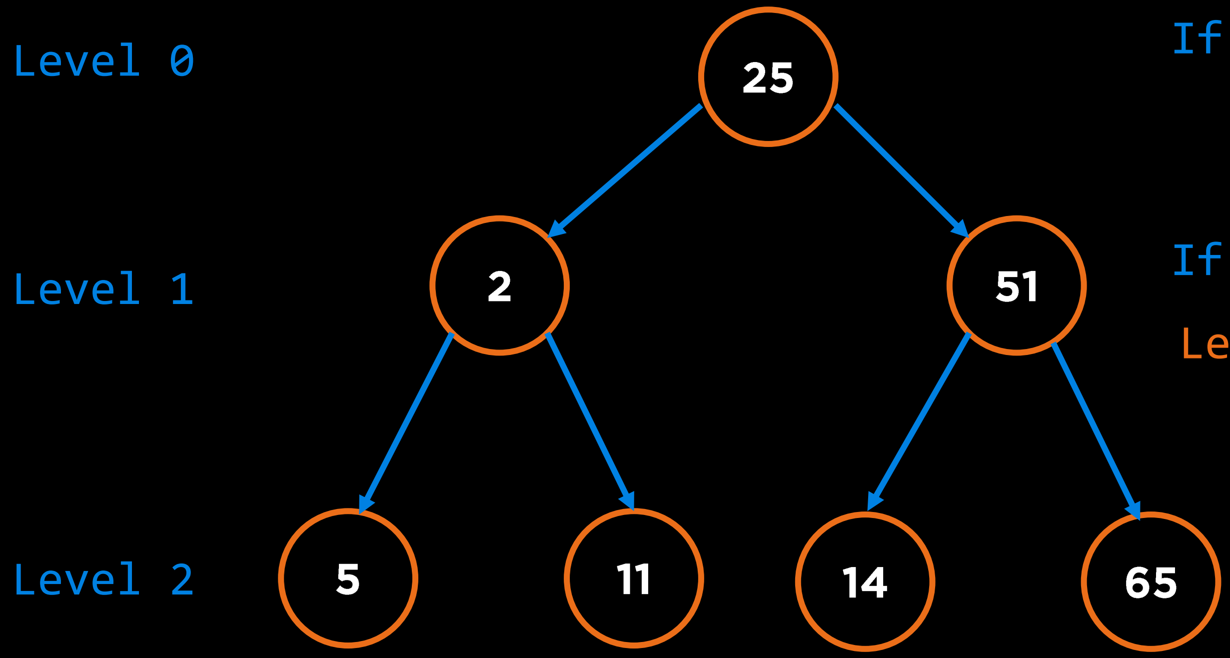
A subtree of a node is a tree whose root is a child of that node.



Trees: Terminology

Level (Depth)

The level of a node is the distance of that node from the root.



`If node n is root,`

`Level(n) = 0`

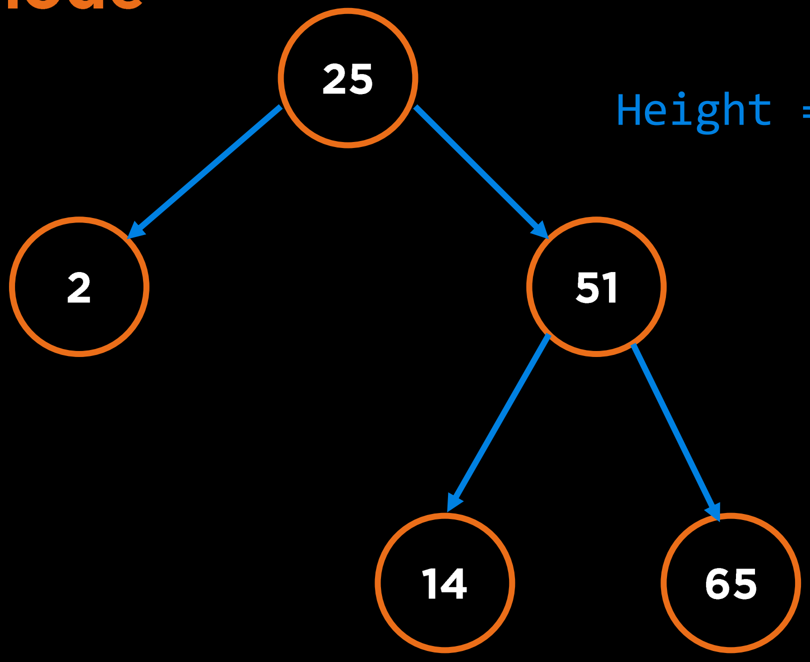
`If node n is not a root,`

`Level(n) = level(parent) + 1`

Trees: Terminology

Height – 0 based height (based on edges)

The height of a tree is the number of nodes in the longest path from the root node to a leaf node



Height = 2

If tree has just the root,

Height = 0

If tree has more than the root,

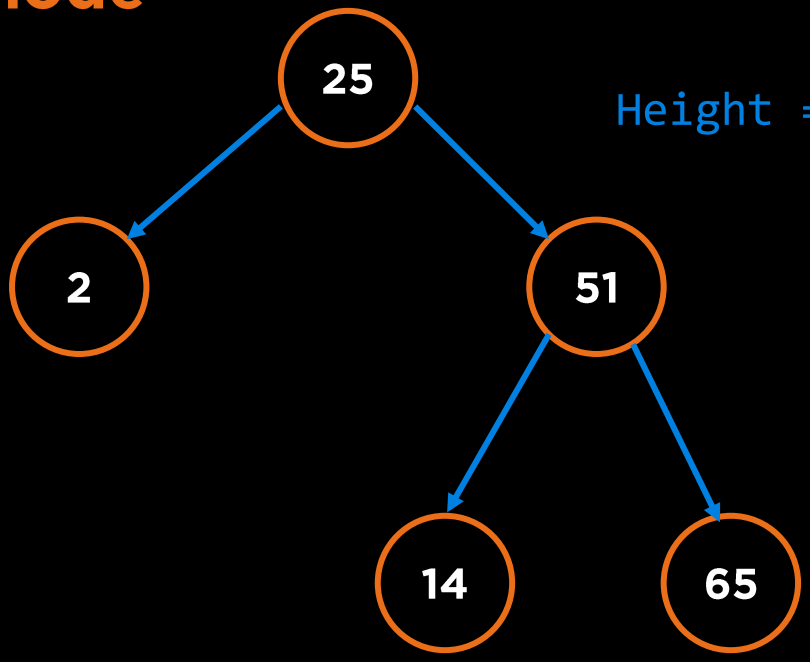
Height = 1 + max(Height(children))

https://stackoverflow.com/questions/4065439/height-of-a-tree-with-only-one-node

Trees: Terminology

Height – 1 based height (based on nodes)

The height of a tree is the number of nodes in the longest path from the root node to a leaf node



Height = 3

```
If tree has just the root,
        Height = 1

If tree has more than the root,
 Height = 1 + max(Height(children))
```

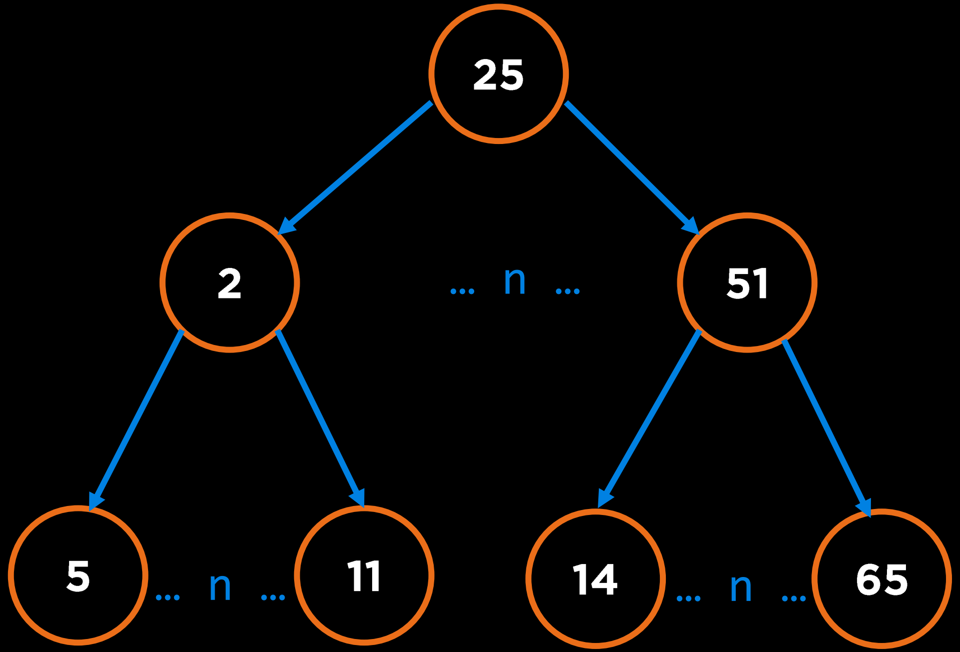
https://stackoverflow.com/questions/4065439/height-of-a-tree-with-only-one-node

Tree Types

Trees: Type

N-Ary Tree

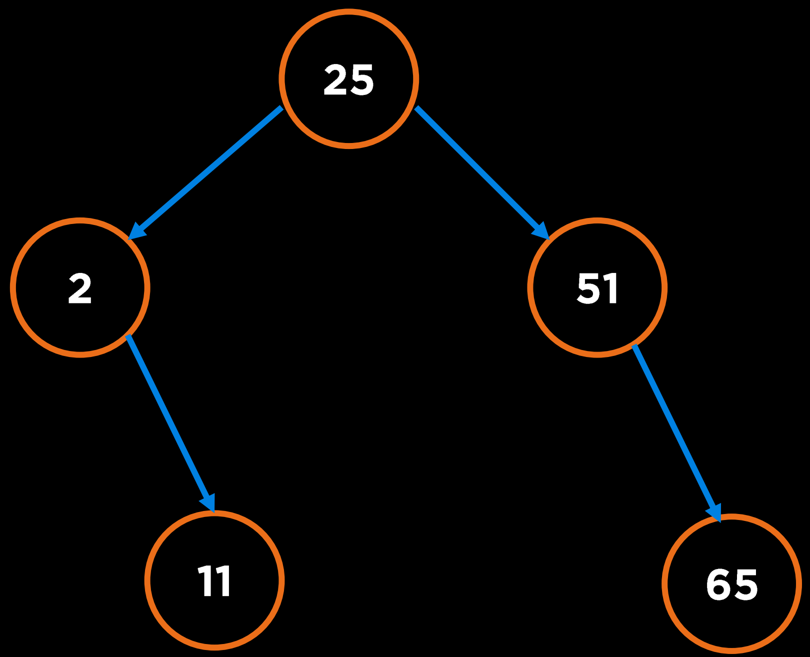
- A tree with each node consisting of at most **n** children.
- Tree has three properties: one root; each node has one parent; and no cycles.



Trees: Type

Binary Tree

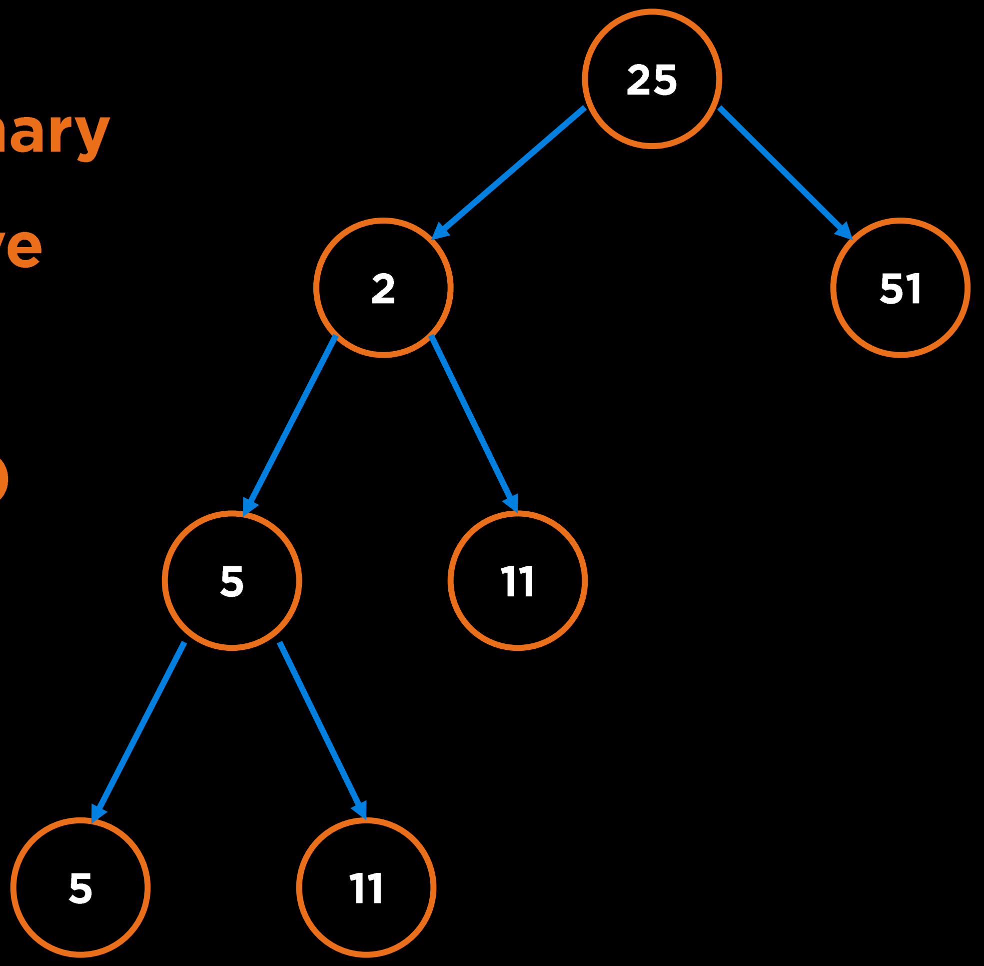
- A tree with each node consisting of at most **two** children.



Trees: Type

Full Binary Tree

- A full binary tree is a binary tree where all nodes have either 2 children or 0 children (the leaf nodes)

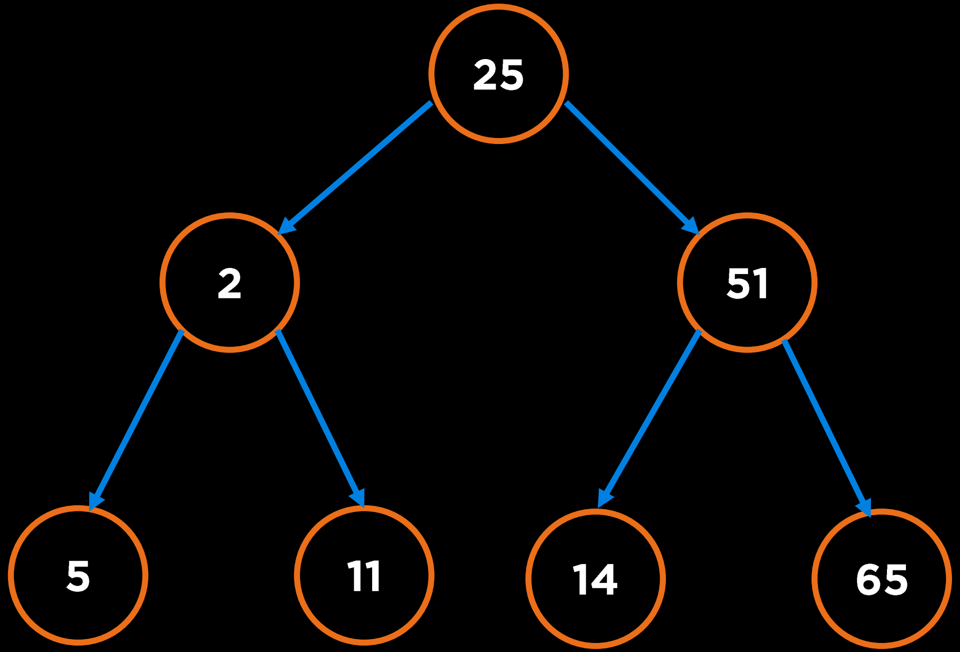


Trees: Type

Perfect Binary Tree

- A perfect binary tree is a full binary tree of height **h** with exactly $2^{h+1} - 1$ nodes. Here, **h** is 0-based (height of a tree with one node is 0).

- In a perfect binary tree with **n** nodes, the height of the tree is $\log_2 (n+1) - 1$.



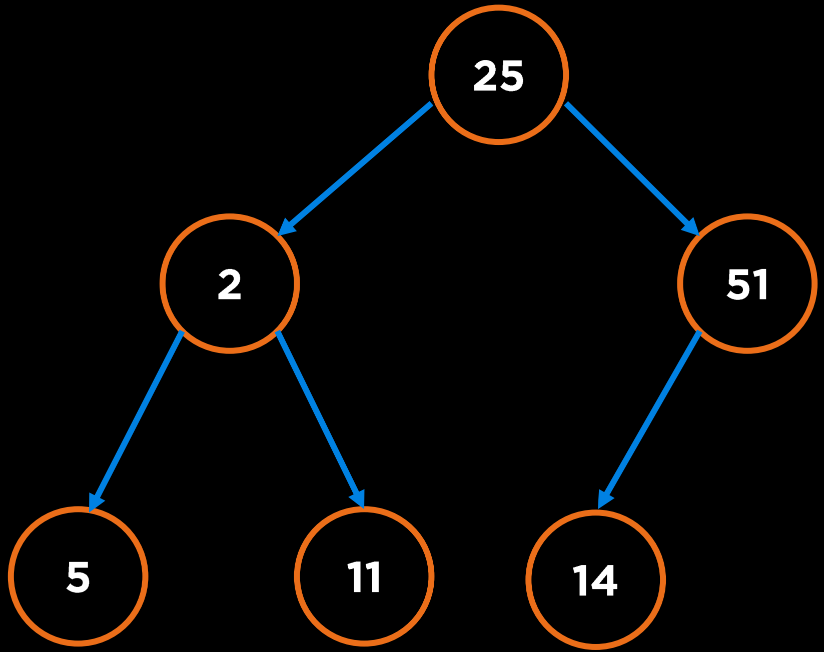
$h = 2$ and $2^{h+1} - 1 = 7$

$n = 7$ and $\log_2 (7+1) - 1 = 2$

Trees: Type

Complete Binary Tree

- A complete binary tree is a perfect binary tree through level **h - 1** with some extra leaf nodes at level **h** (the tree height), all towards the left
- The height of a complete binary tree is also **ceil($\log_2$ (n+1) - 1)**



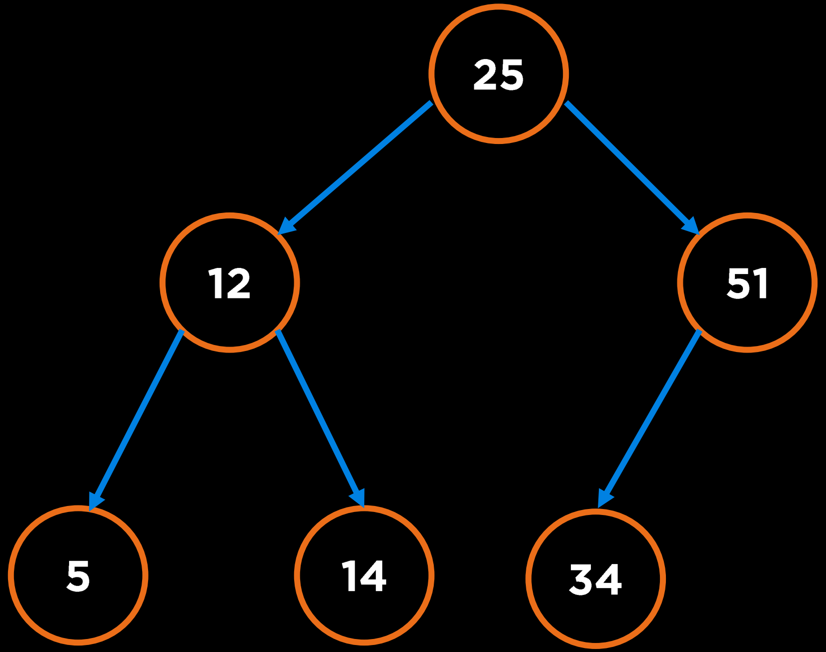
`Is a full tree complete? Is a complete tree full?`

Trees: Type

Binary Search Tree (BST)

- A binary tree in which all values of a node's left subtree or descendants to the left are less than the node and all values of a node's right subtree are greater than the node.

- An ordered binary tree.



Trees Representation

Trees: Representation

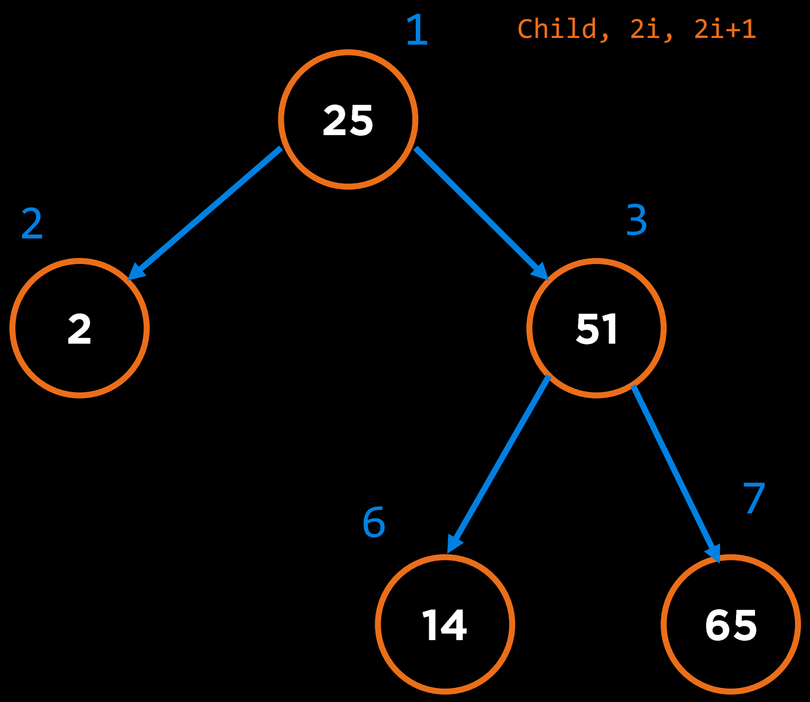
Trees: Representation

1 **Array Representation**

Tree[]											
		25	2	51			14	65			
	0					5					10

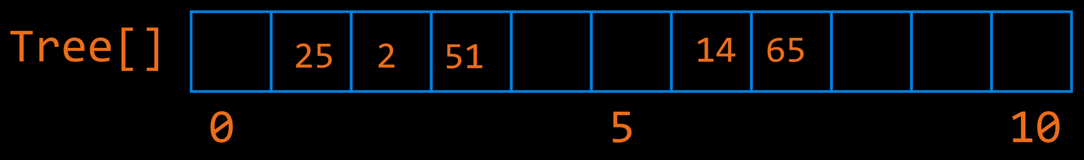
Parent, i

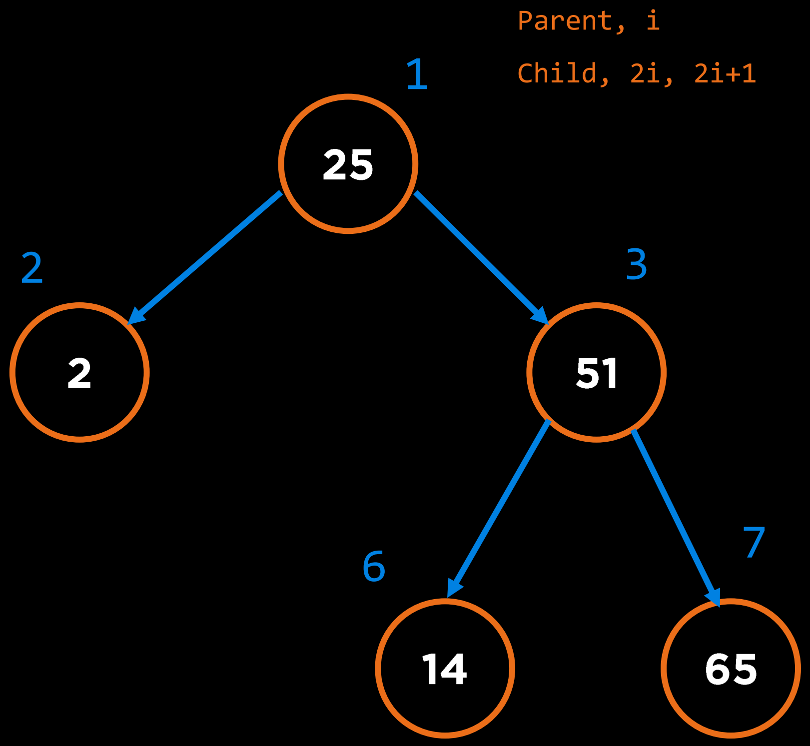
Child, 2i, 2i+1



Trees: Representation

1 **Array Representation**



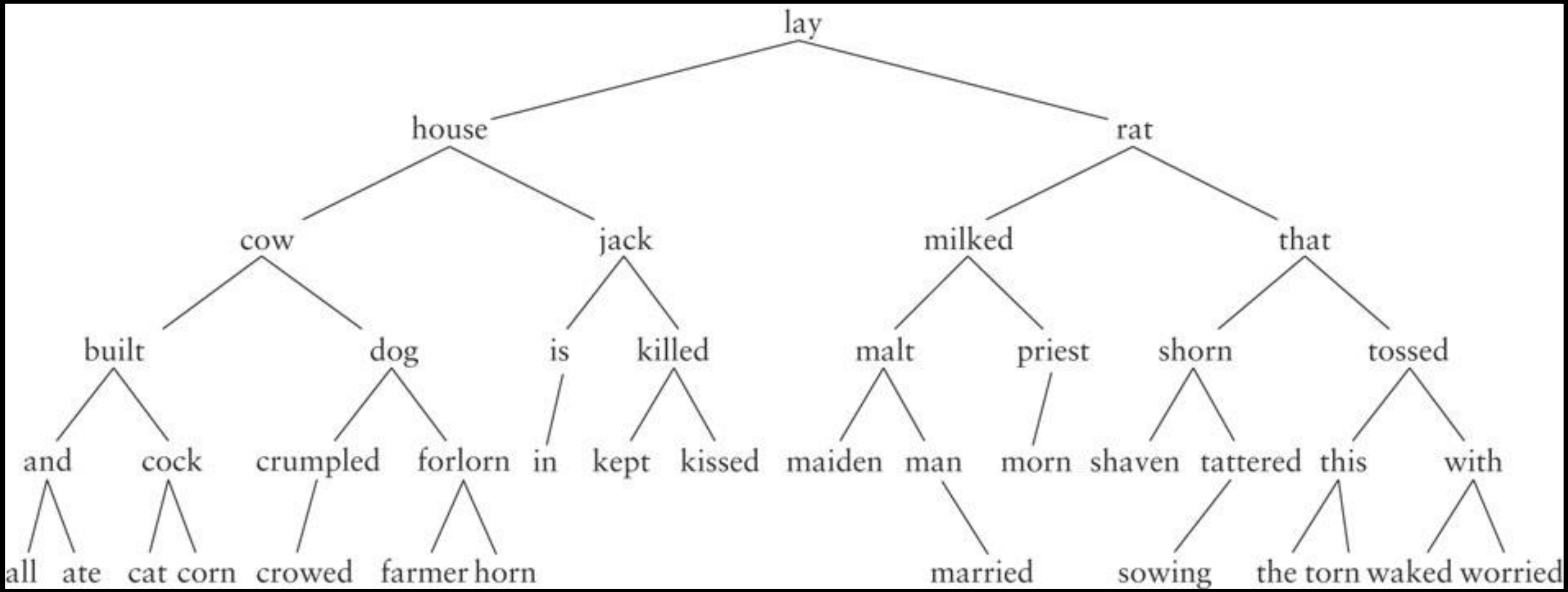


2 **Linked Representation**

```
class TreeNode
{
    public:
        int val;
        TreeNode *left;
        TreeNode *right;
        TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
};
```

Binary Search Trees

Binary Search Tree (BST): Dictionary



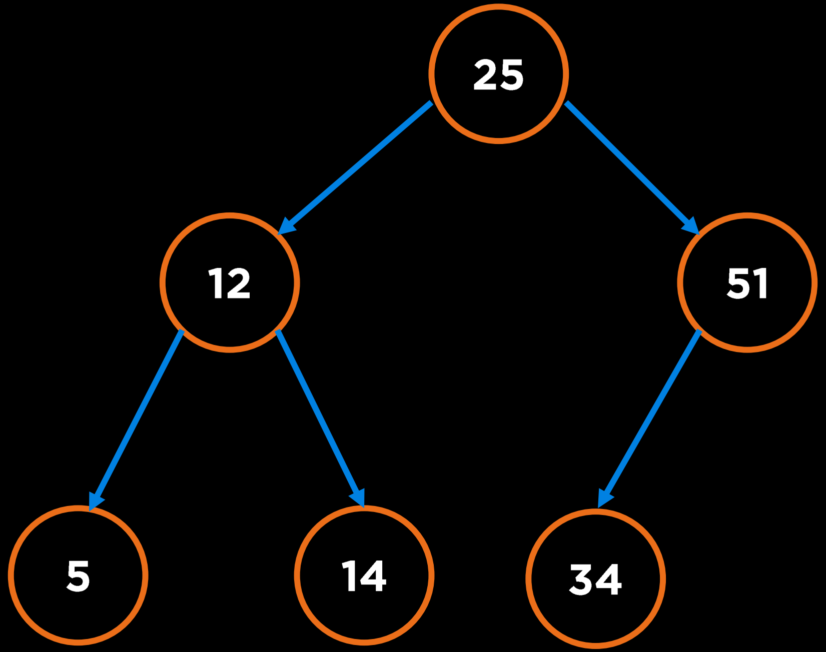
Binary Search Tree: C++ Node Class

```
class TreeNode
{
    public:
        int val;
        TreeNode *left;
        TreeNode *right;
        TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
};
```

https://onlinegdb.com/BygDgQCjI

Binary Search Tree Search

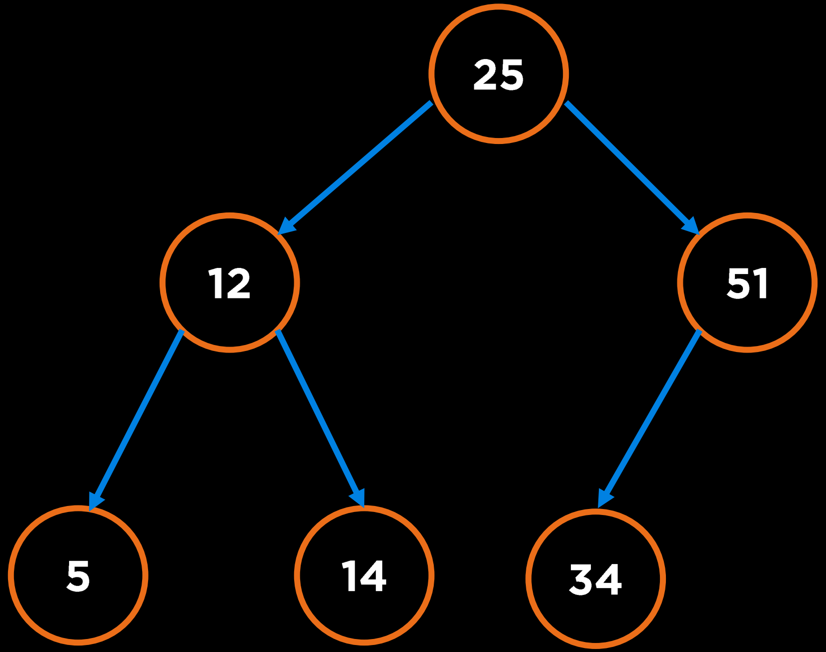
```
if the tree is empty
        return null (target is not found)
else if the target matches the root node's data
        return the data stored at the root node
else if the target is less than the root node's data
        return the result of searching the left subtree of the root
else
        return the result of searching the right subtree of the root
```



Binary Search Tree Insertion

Recursive Algorithm for Insertion in a Binary Search Tree

1. `if` the root is `null`
2. Replace empty tree with a new tree with the item at the root and return `true`.
3. `else if` the item is equal to `root.data`
4. The item is already in the tree; return `false`.
5. `else if` the item is less than `root.data`
6. Recursively insert the item in the left subtree.
7. `else`
8. Recursively insert the item in the right subtree.



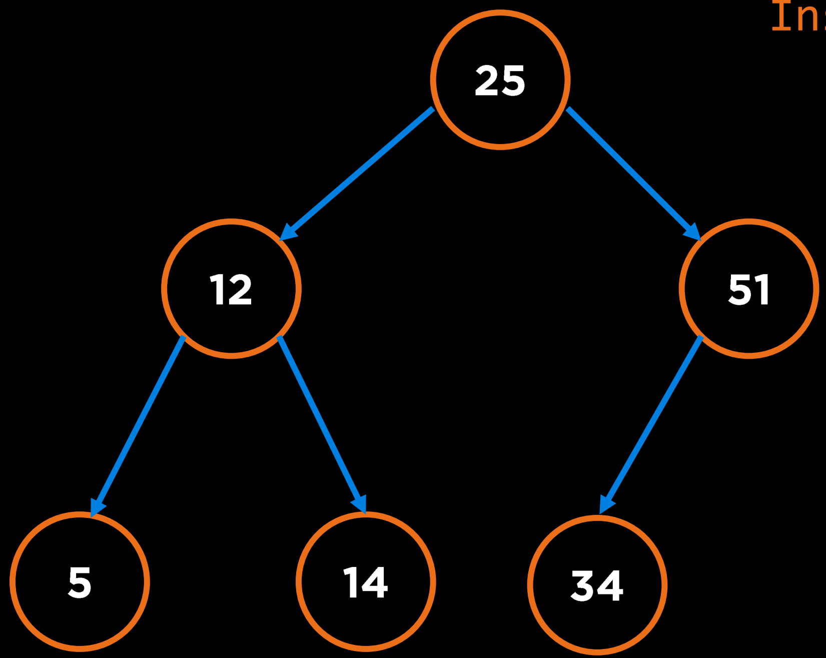
Binary Search Tree: C++ Insert

```cpp
class TreeNode
{
    public:
        int val;
        TreeNode *left;
        TreeNode *right;
        TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
};

TreeNode* insert(TreeNode* root, int key)
{
    if (root == nullptr)
        return new TreeNode(key);
    if (key < root->val)
        root->left = insert(root->left, key);
    else
        root->right = insert(root->right, key);
    return root;
}
```

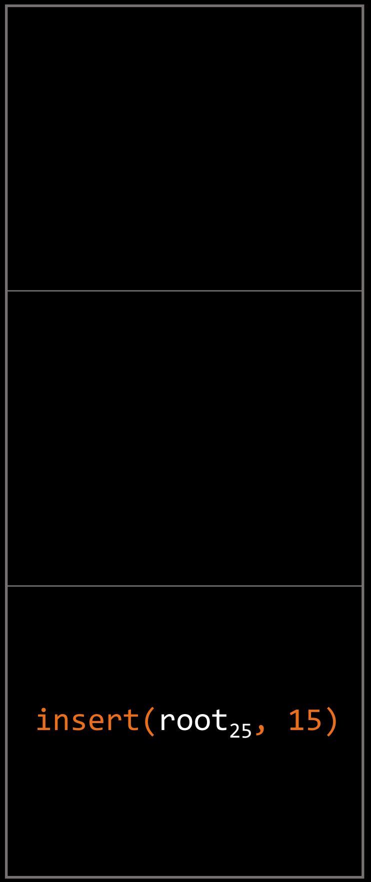
https://onlinegdb.com/BygDgQCjI

Binary Search Tree: C++ Insert Example

Insert 15

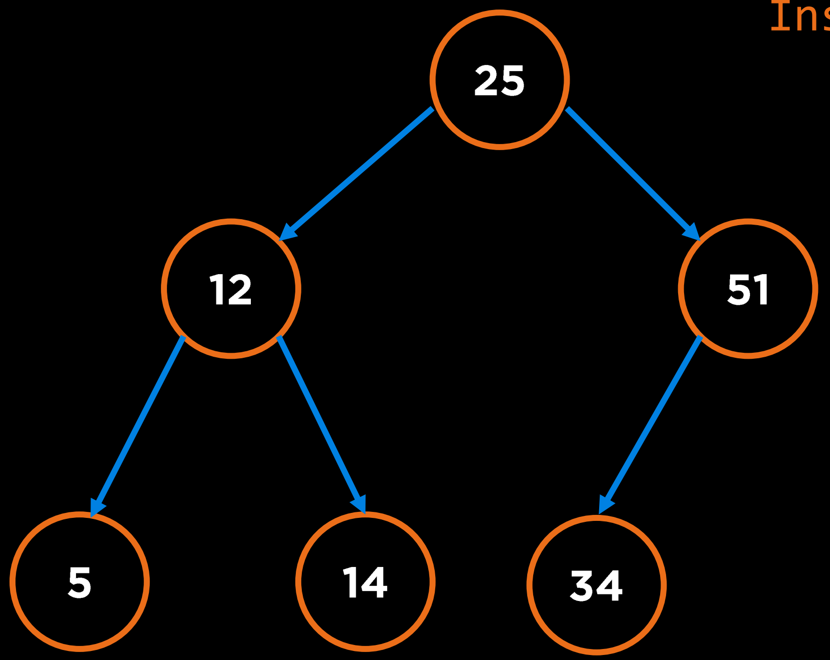



```cpp
TreeNode* insert(TreeNode* root, int key)
{
    if (root == nullptr)
        return new TreeNode(key);
    if (key < root->val)
        root->left = insert(root->left, key);
    else
        root->right = insert(root->right, key);
    return root;
}
```



https://onlinegdb.com/BygDgQCjI

Binary Search Tree: C++ Insert Example



Insert 15

```cpp
TreeNode* insert(TreeNode* root, int key)
{
    if (root == nullptr)
        return new TreeNode(key);
    if (key < root->val)
        root->left = insert(root->left, key);
    else
        root->right = insert(root->right, key);
    return root;
}
```

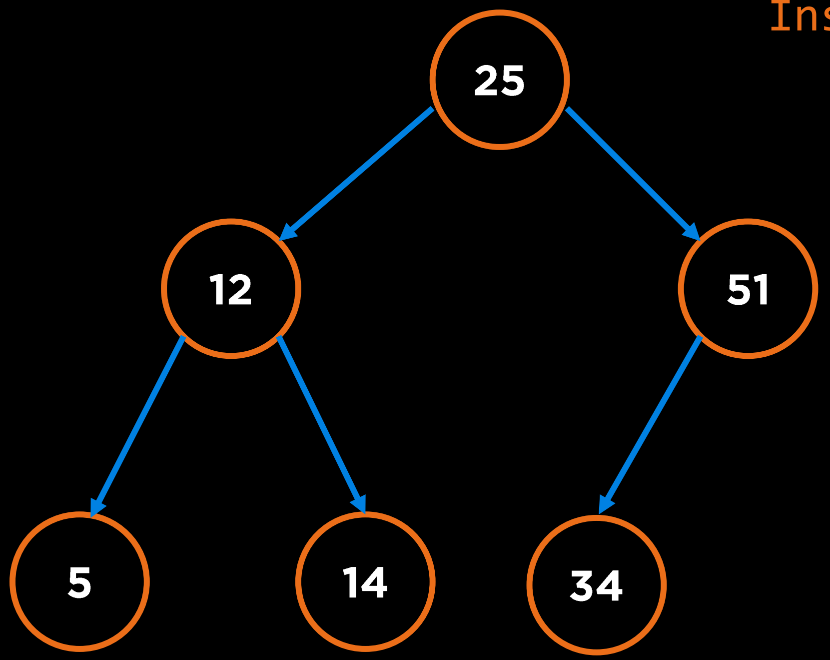
```cpp
TreeNode* insert(TreeNode* root, int key)
{
    if (root == nullptr)
        return new TreeNode(key);
    if (key < root->val)
        root->left = insert(root->left, key);
    else
        root->right = insert(root->right, key);
    return root;
}
```

insert($root_{12}$, 15)

insert($root_{25}$, 15)

https://onlinegdb.com/BygDgQCjI

Binary Search Tree: C++ Insert Example



Insert 15

```
TreeNode* insert(TreeNode* root, int key)
{
    if (root == nullptr)
        return new TreeNode(key);
    if (key < root->val)
        root->left = insert(root->left, key);
    else
        root->right = insert(root->right, key);
    return root;
}
```


```
TreeNode* insert(TreeNode* root, int key)
{
    if (root == nullptr)
        return new TreeNode(key);
    if (key < root->val)
        root->left = insert(root->left, key);
    else
        root->right = insert(root->right, key);
    return root;
}
```

```
TreeNode* insert(TreeNode* root, int key)
{
    if (root == nullptr)
        return new TreeNode(key);
    if (key < root->val)
        root->left = insert(root->left, key);
    else
        root->right = insert(root->right, key);
    return root;
}
```


insert($root_{14}$, 15)

insert($root_{12}$, 15)

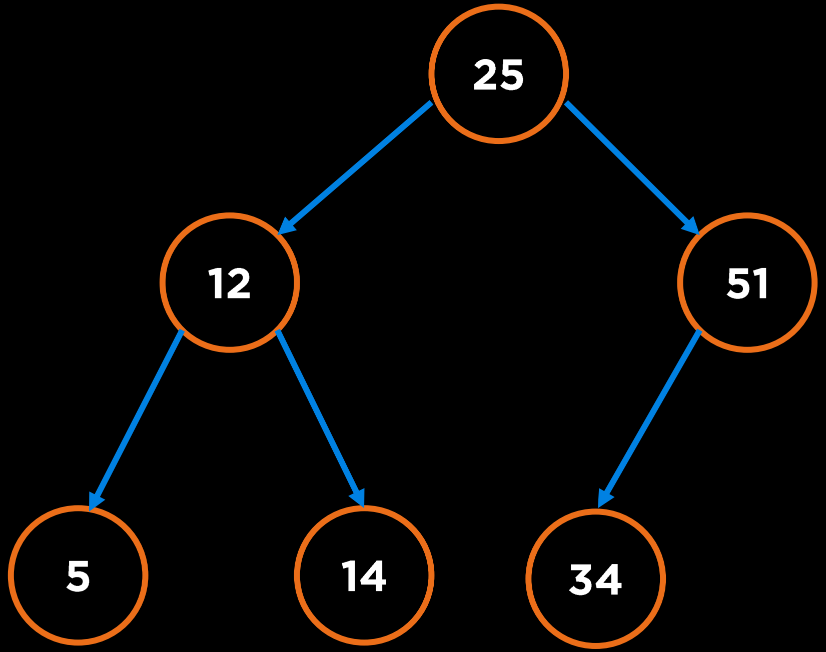
insert($root_{25}$, 15)

https://onlinegdb.com/BygDgQCjI

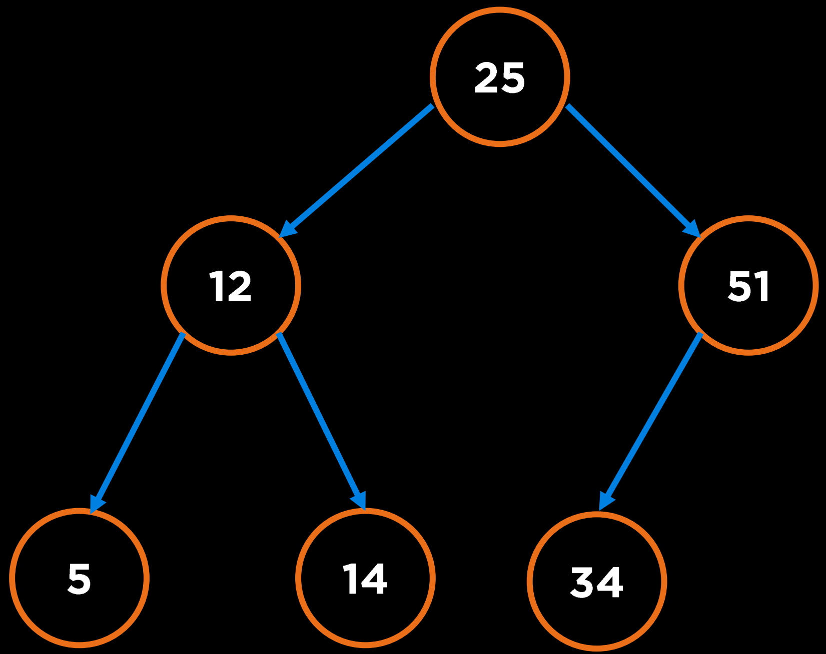
Binary Search Tree Deletion

Recursive Algorithm for Removal from a Binary Search Tree

```
1.  if the root is null
2.      The item is not in tree – return null.
3.  Compare the item to the data at the local root.
4.  if the item is less than the data at the local root
5.      Return the result of deleting from the left subtree.
6.  else if the item is greater than the local root
7.      Return the result of deleting from the right subtree.
8.  else // The item is in the local root
9.      Store the data in the local root in deletedReturn.
10.     if the local root has no children
11.         Set the parent of the local root to reference null.
12.     else if the local root has one child
13.         Set the parent of the local root to reference that child.
14.     else // Find the inorder predecessor
15.         if the left child has no right child it is the inorder predecessor
16.             Set the parent of the local root to reference the left child.
17.         else
18.             Find the rightmost node in the right subtree of the left
                child.
19.             Copy its data into the local root's data and remove it by
                setting its parent to reference its left child.
```

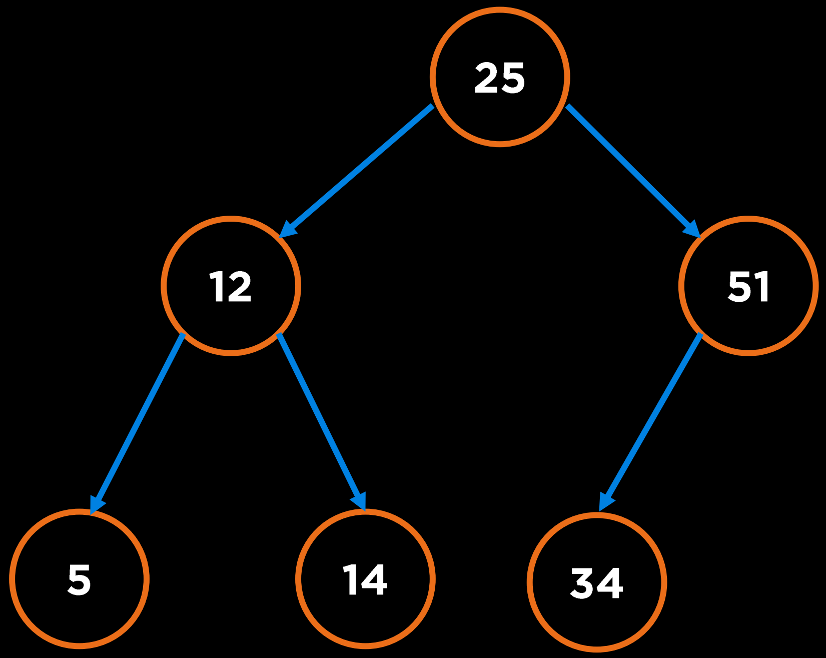


Binary Search Tree Deletion: Example



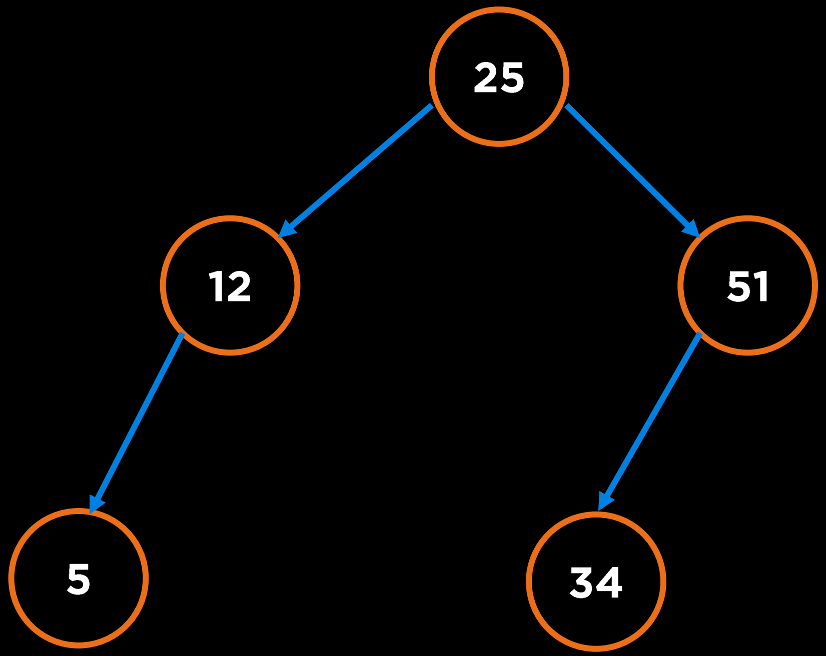
Delete 14: No child

Binary Search Tree Deletion: Example

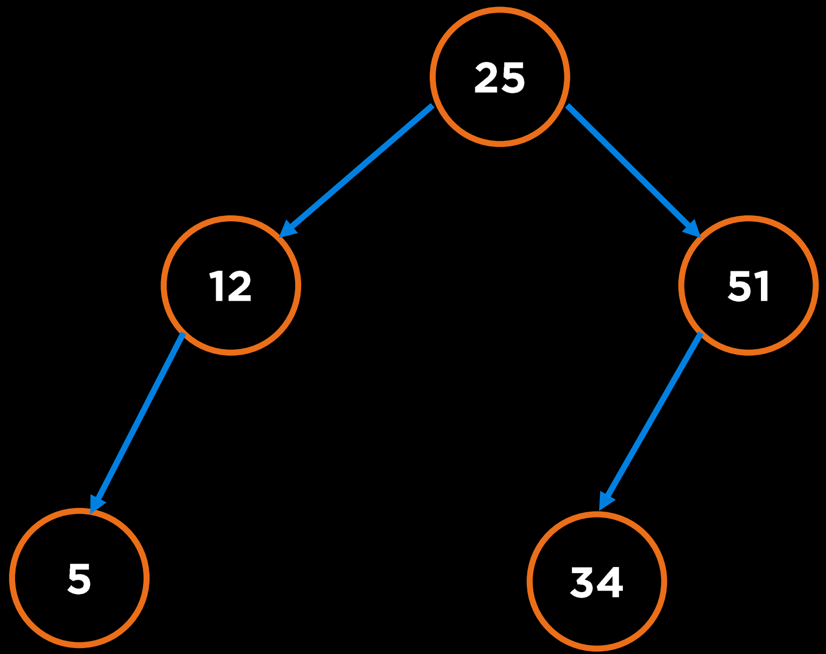


Delete 14: No child

At the parent of 14, set the child with value 14 to null.

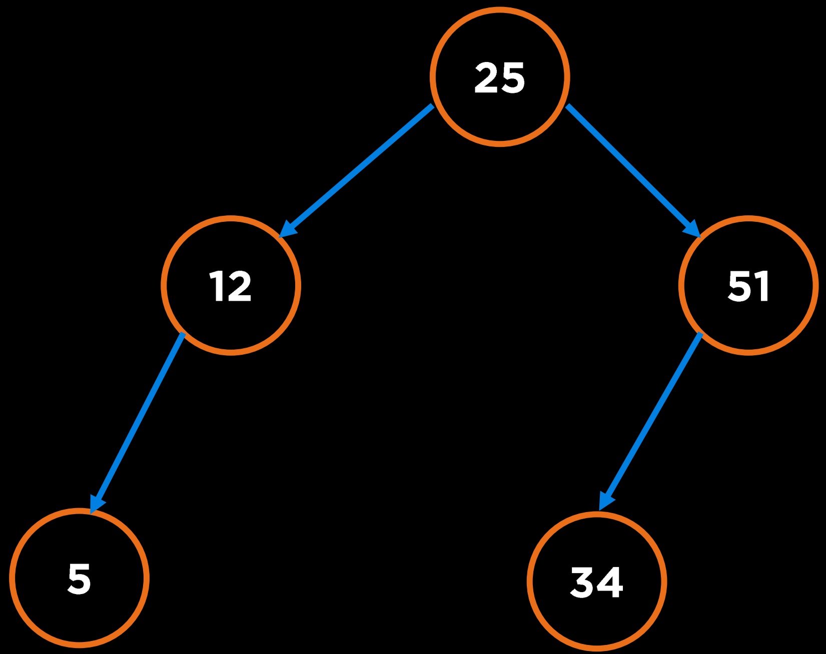


Binary Search Tree Deletion: Example



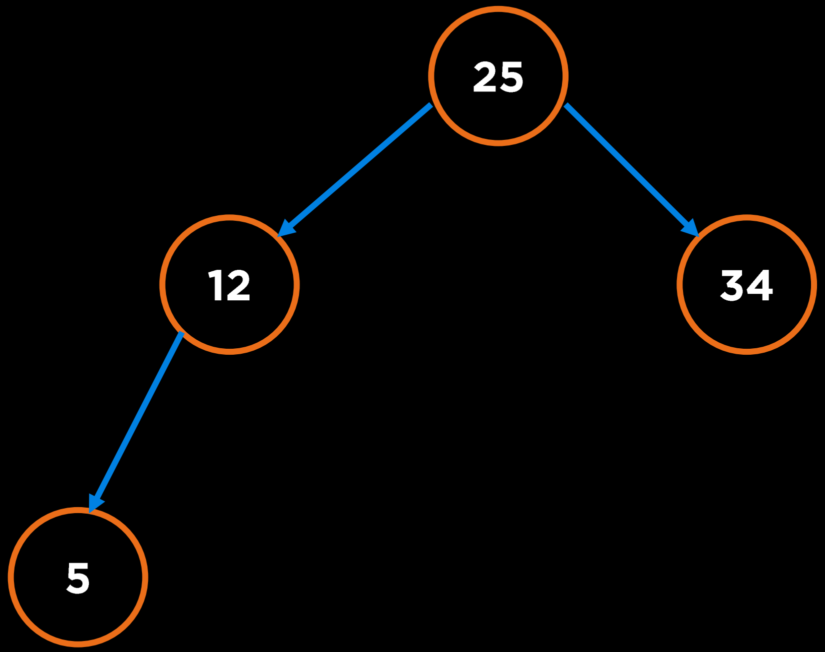
Delete 51: One child

Binary Search Tree Deletion: Example

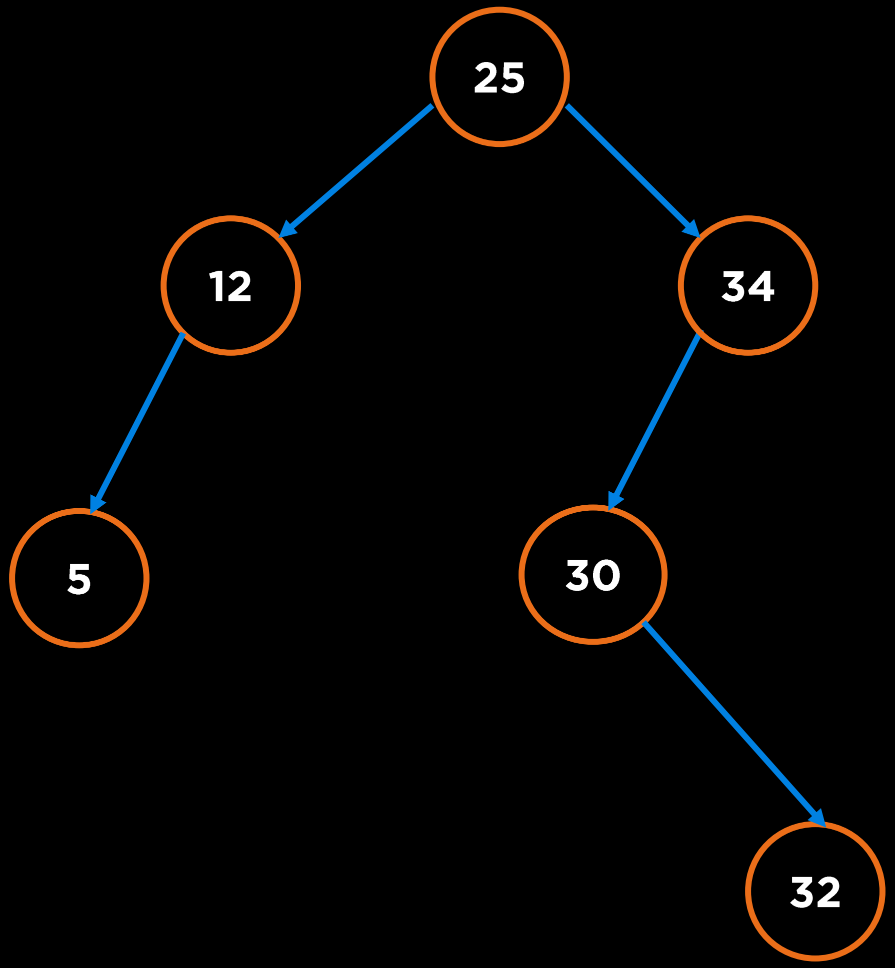


Delete 51: One child

At the parent of 51, set the pointer with the child value 51 to the pointer of the node 51's child.

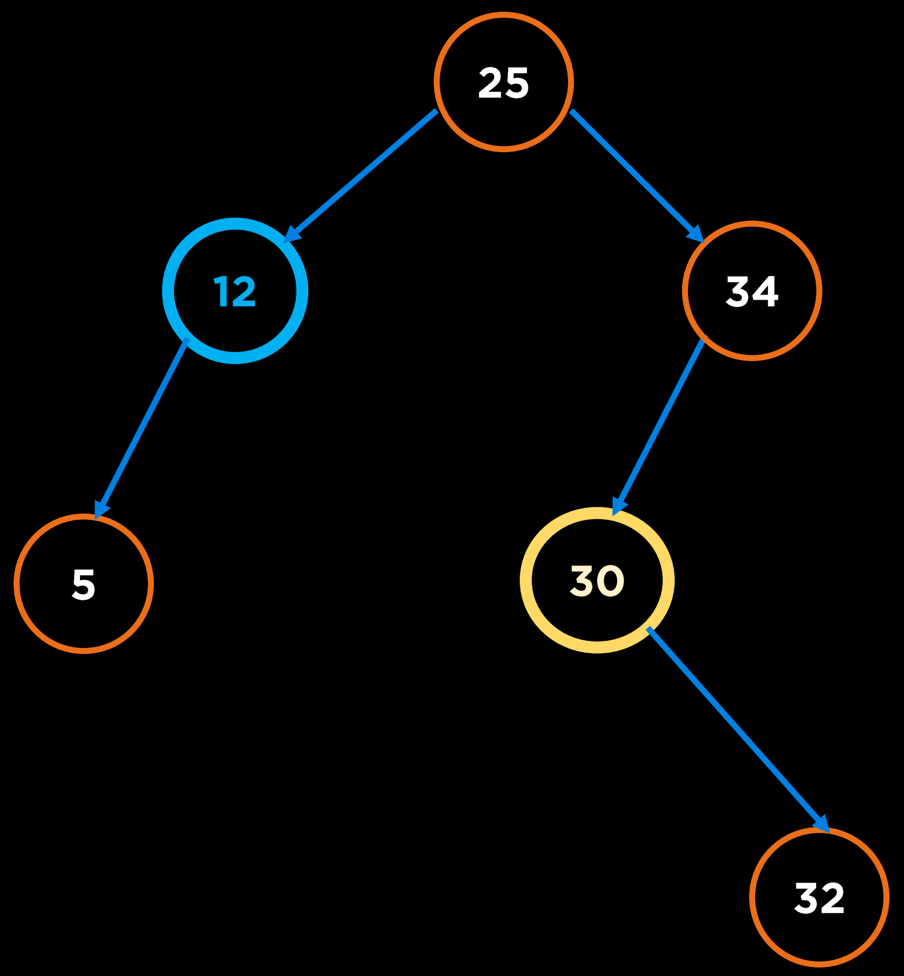


Binary Search Tree Deletion: Example



Delete 25: Two children

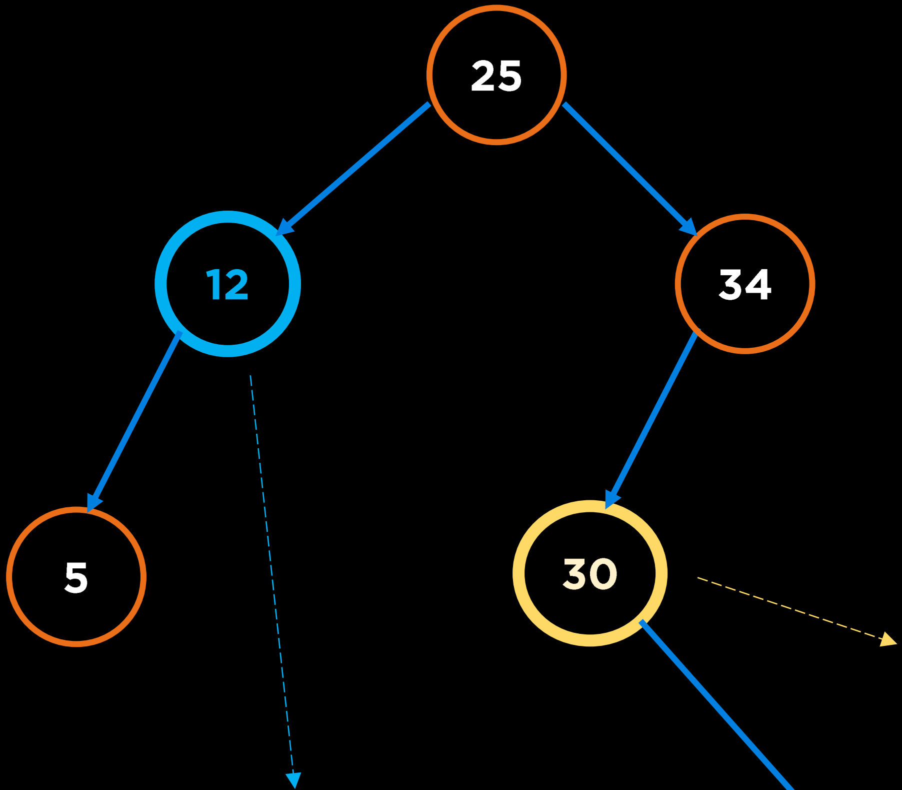
Binary Search Tree Deletion: Example



Delete 25: Two children

Change the value of the node to be deleted to the value of inorder successor or predecessor. Then delete the successor or predecessor value at the subtree.

Binary Search Tree Deletion: Example



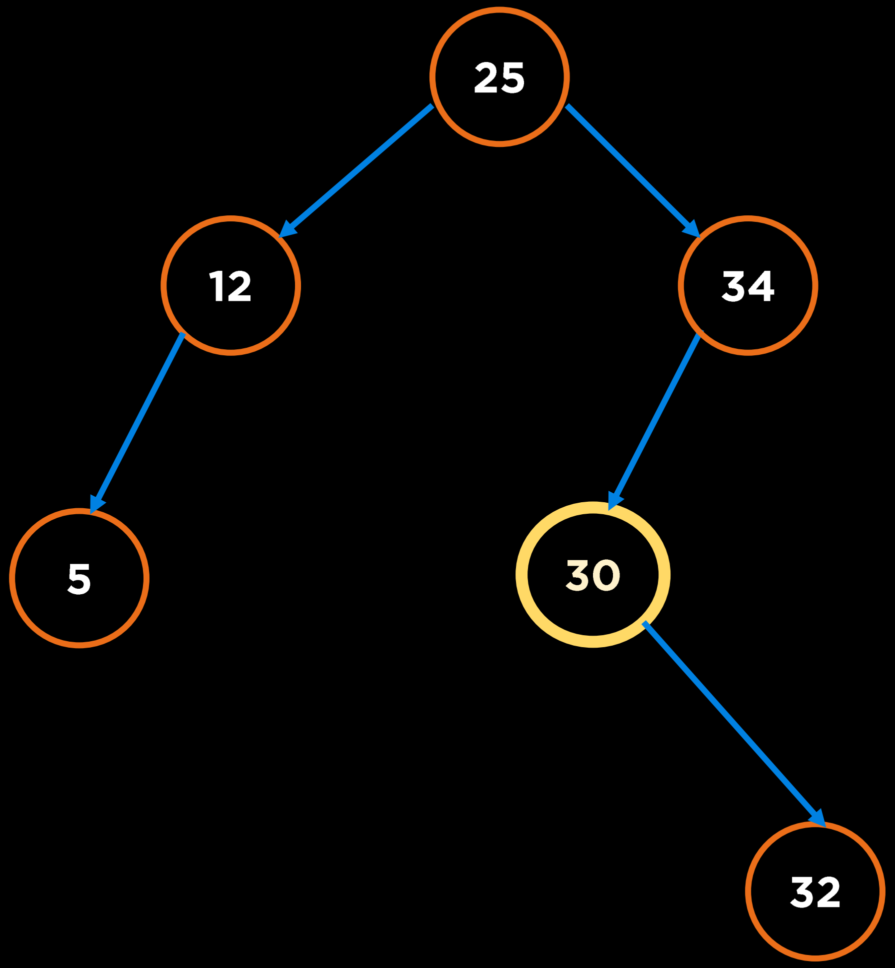
Delete 25: Two children

Change the value of the node to be deleted to the value of inorder successor or predecessor. Then delete the successor or predecessor value at the subtree.

Successor: right subtree’s leftmost child

Predecessor: left subtree’s rightmost child

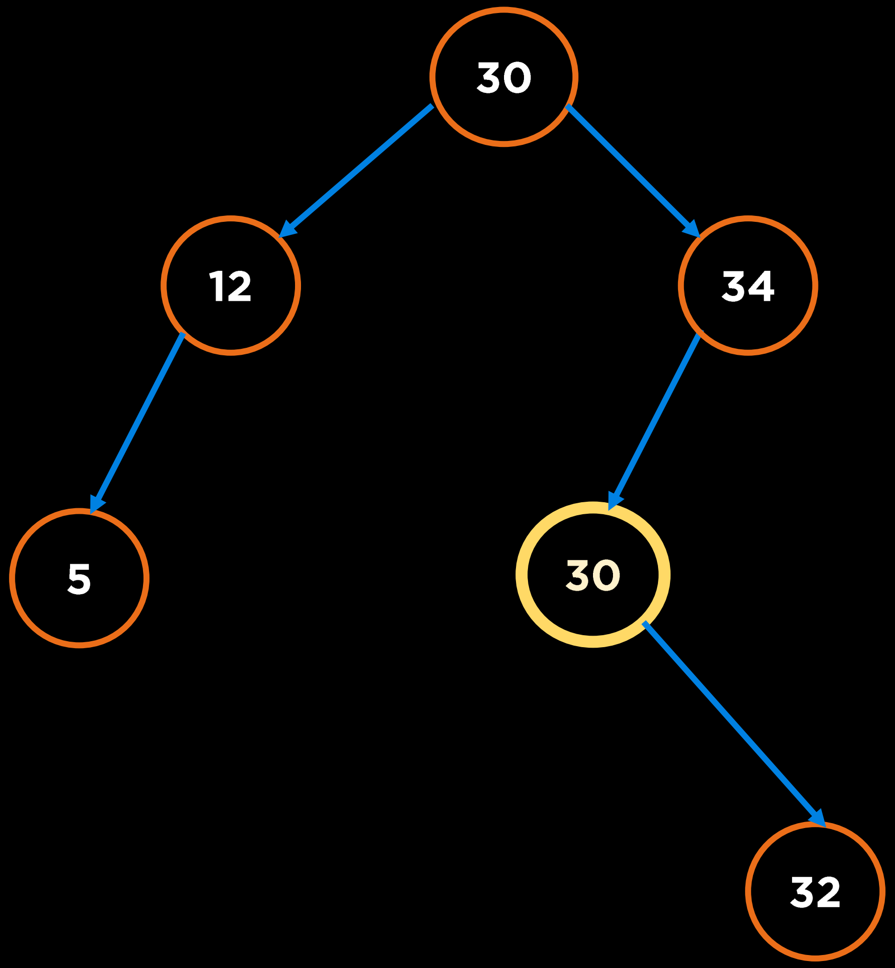
Binary Search Tree Deletion: Example



Delete 25: Two children

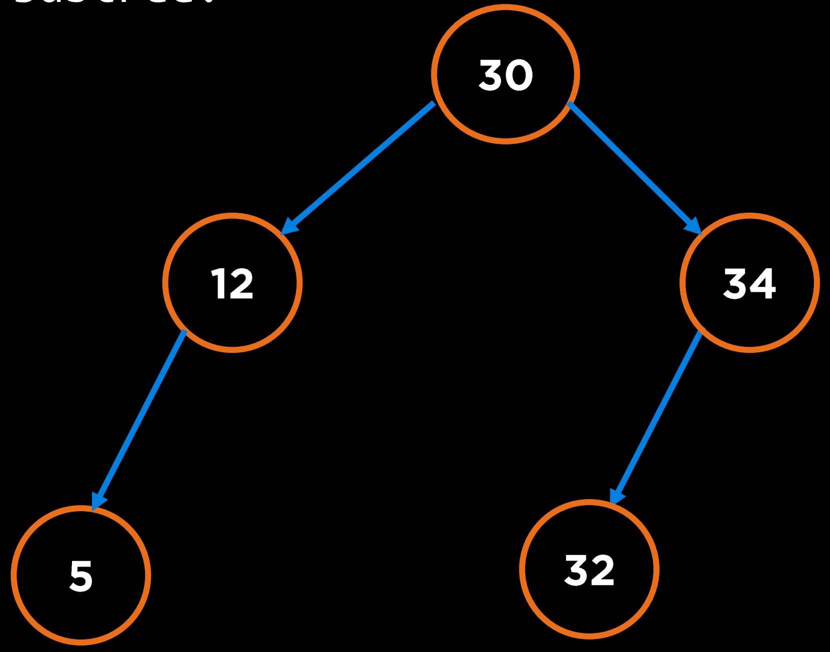
Change the value of the node to be deleted to the value of inorder successor. Then delete the successor or predecessor value at the subtree.

Binary Search Tree Deletion: Example



Delete 25: Two children

Change the value of the node to be deleted to the value of inorder successor. Then delete the successor or predecessor value at the subtree.



Traversals

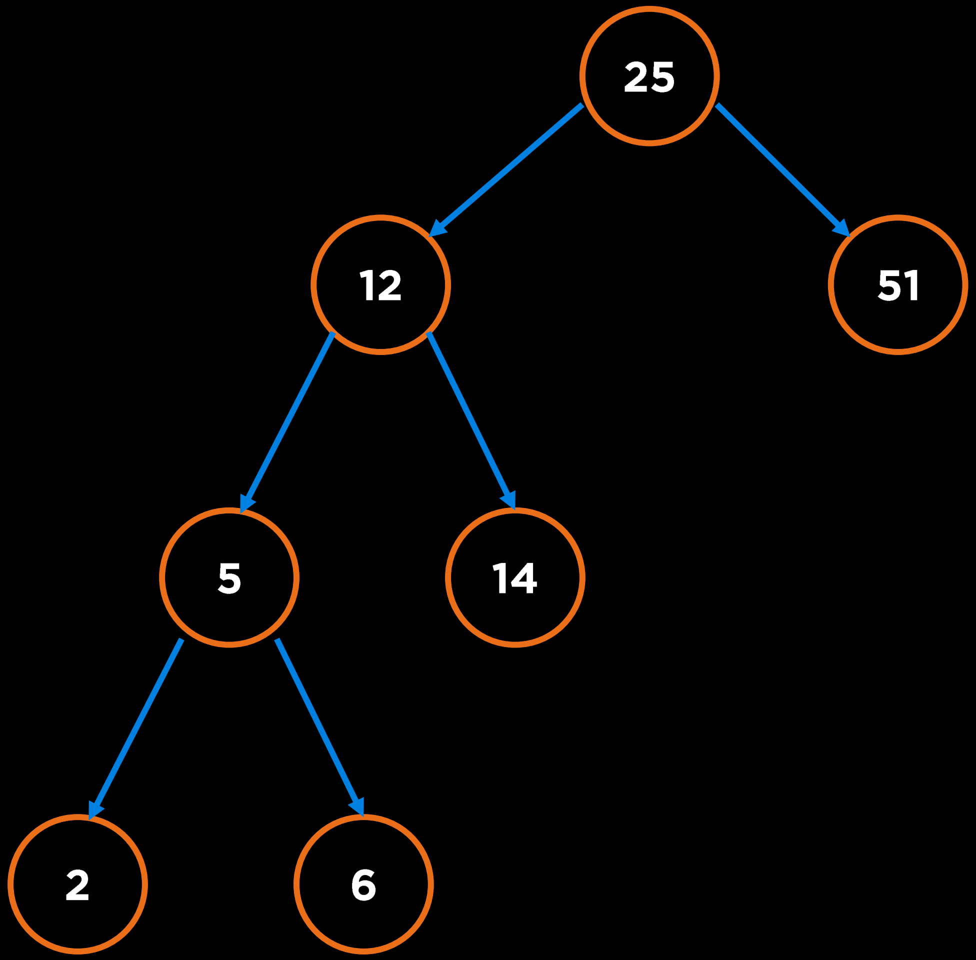
BST Traversals

Visiting each node in the tree

- Depth First Strategy
  - Inorder
  - Preorder
  - Postorder
- Breadth First Strategy

Traversal vs Search

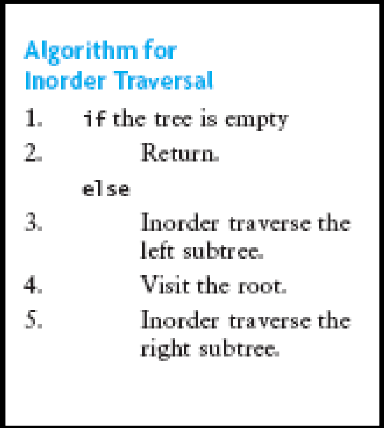
- Traversal requires you to visit each node; Not necessarily in search



BST Traversals: Inorder

Strategy

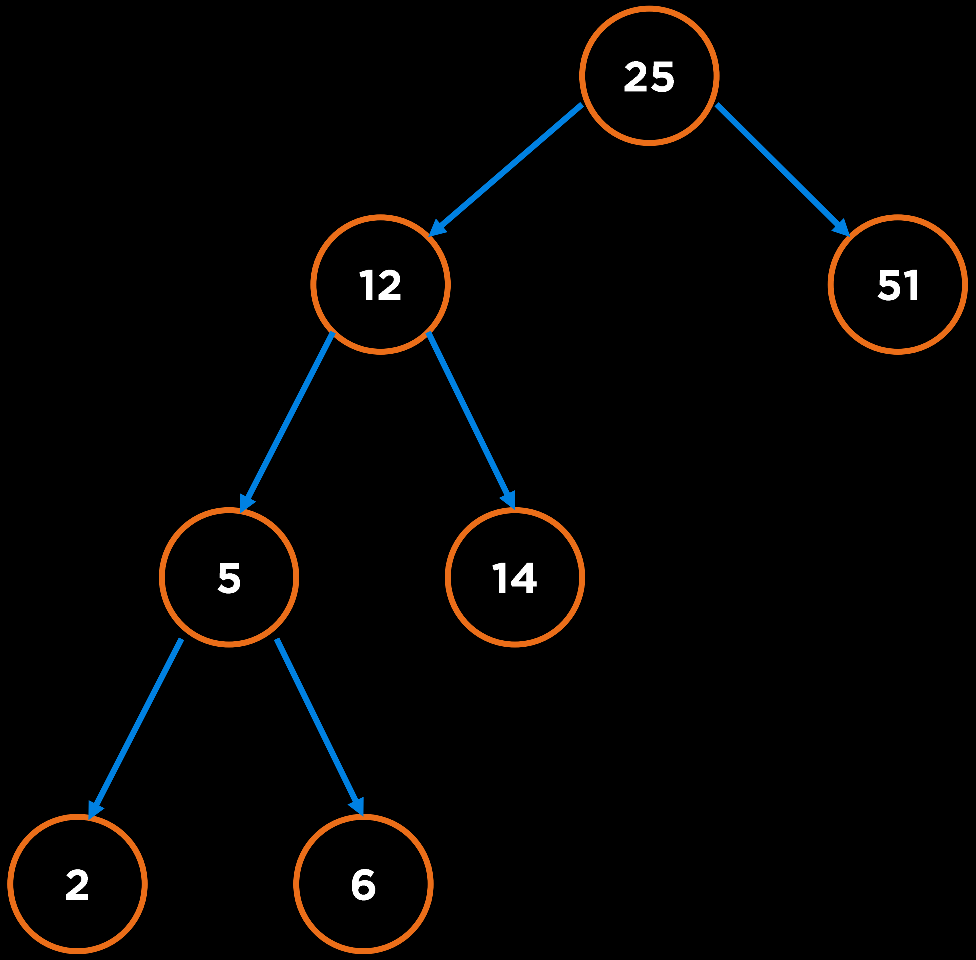
- Visit Left Subtree
- Visit Root
- Visit Right Subtree



Algorithm for Inorder Traversal

1. `if` the tree is empty
2. Return.

   `else`
3. Inorder traverse the left subtree.
4. Visit the root.
5. Inorder traverse the right subtree.



2, 5, 6, 12, 14, 25, 51

BST Traversals: Inorder

Strategy

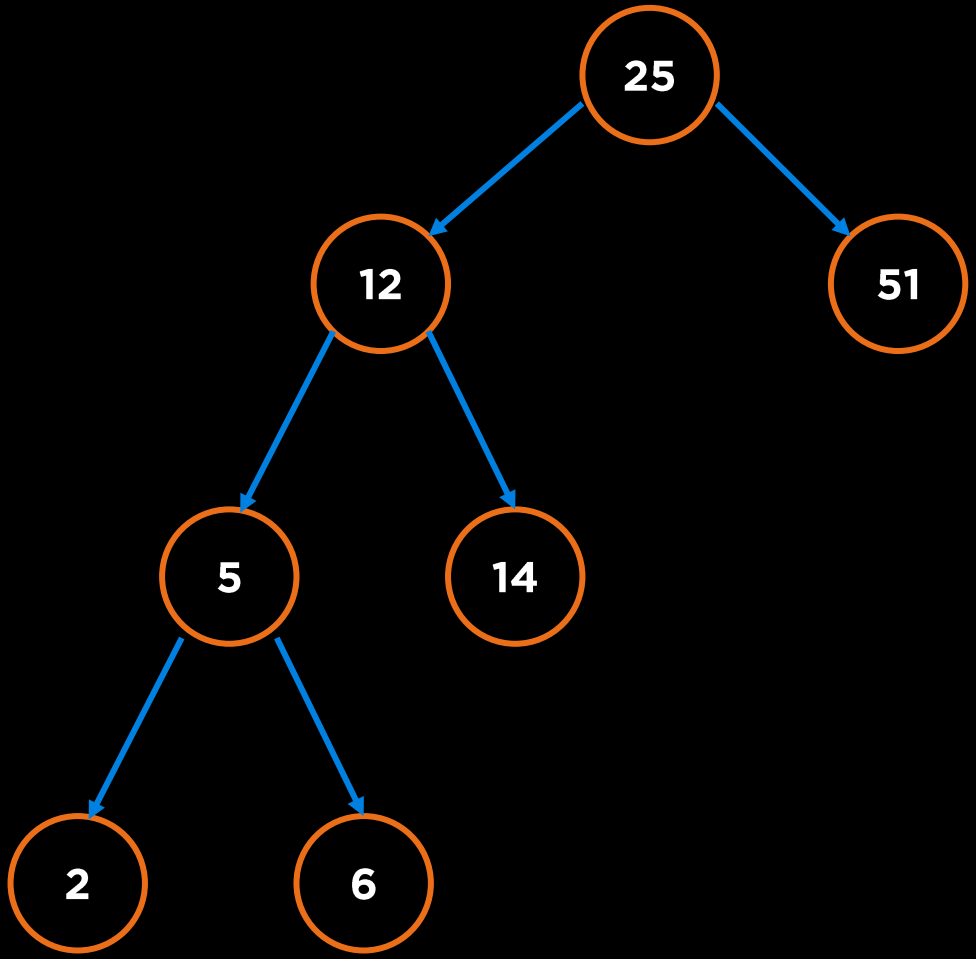
- Visit Left Subtree
- Visit Root
- Visit Right Subtree

Algorithm for Inorder Traversal

1. `if` the tree is empty
2. Return.

   `else`

3. Inorder traverse the left subtree.
4. Visit the root.
5. Inorder traverse the right subtree.



Binary Search Tree: C++ Inorder Traversal

```
class TreeNode
{
    public:
        int val;
        TreeNode *left;
        TreeNode *right;
        TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
};

void inorder(TreeNode* head)
{
    if(head == nullptr)
        cout << "";
    else
    {
        inorder(head->left);
        cout << head->val << " ";
        inorder(head->right);
    }
}
```

https://onlinegdb.com/BygDgQCjI

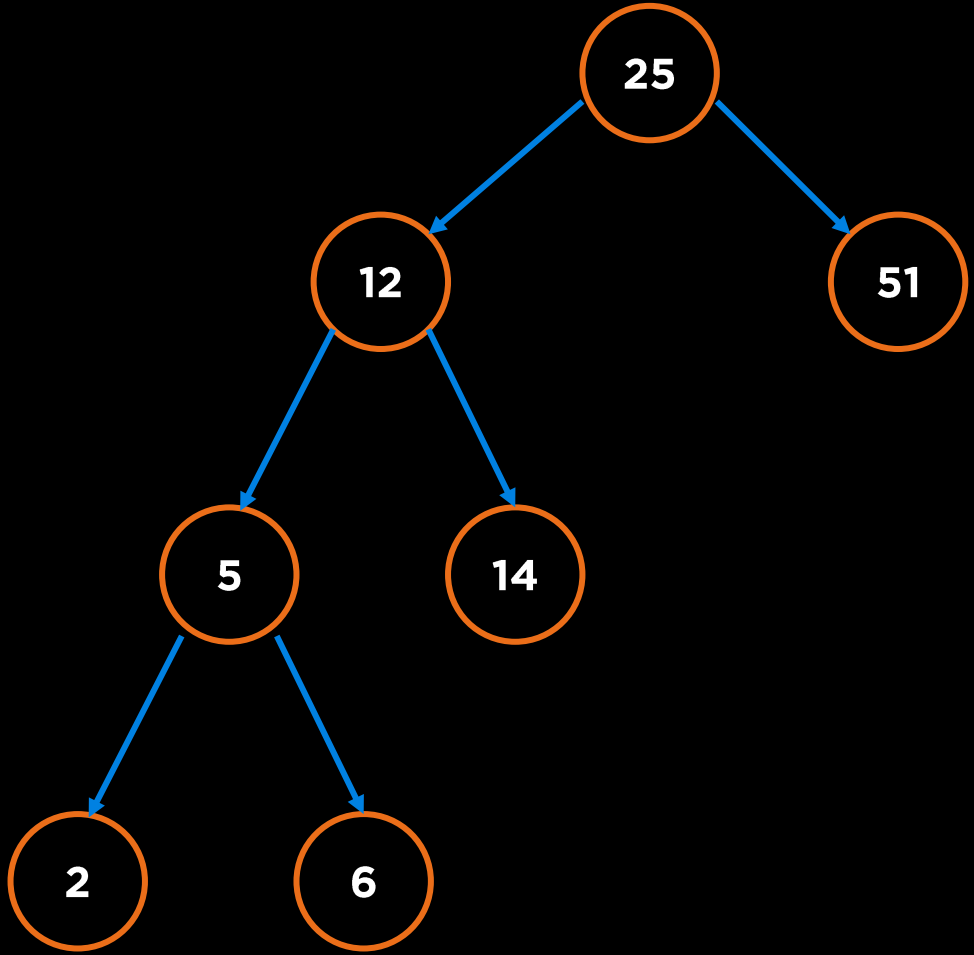
BST Traversals: Preorder

Strategy

- Visit Root
- Visit Left Subtree
- Visit Right Subtree

Algorithm for Preorder Traversal

1. if the tree is empty
2. Return.
   else
3. Visit the root.
4. Preorder traverse the left subtree.
5. Preorder traverse the right subtree.



BST Traversals: Preorder

Strategy

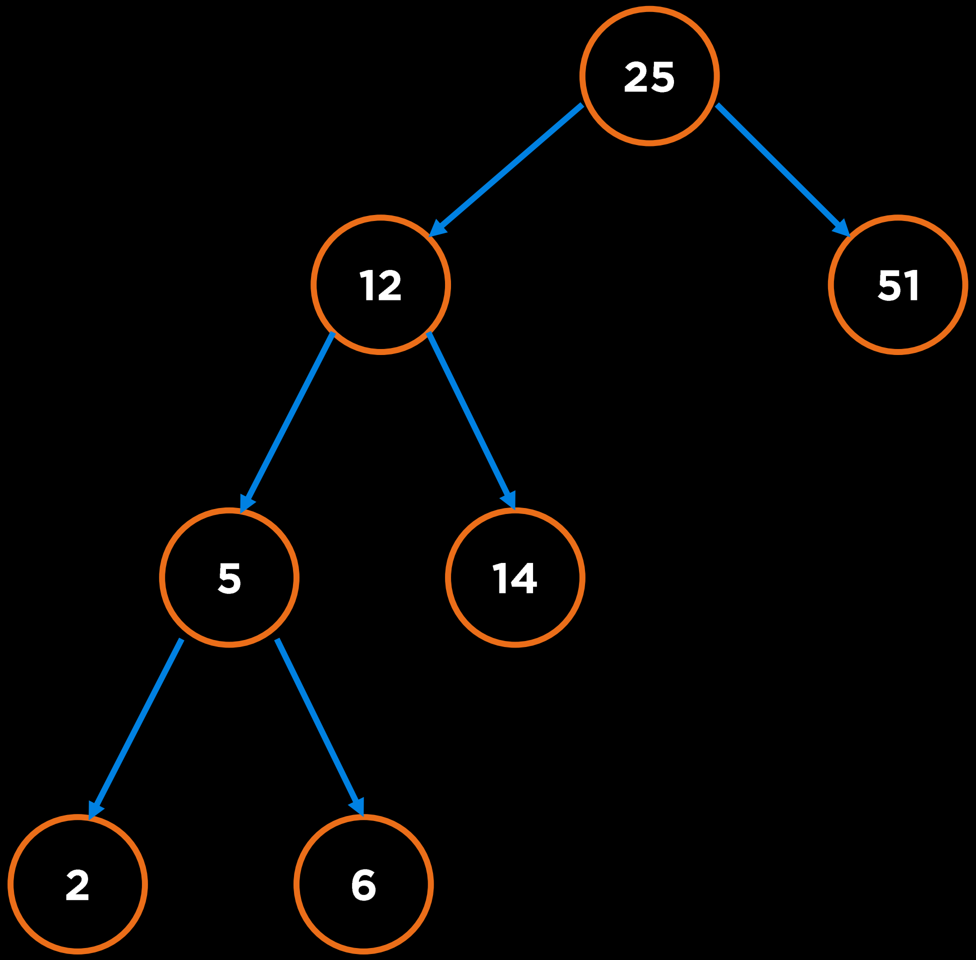
- Visit Root
- Visit Left Subtree
- Visit Right Subtree

Algorithm for Preorder Traversal

1. `if` the tree is empty
2. Return.

   `else`

3. Visit the root.
4. Preorder traverse the left subtree.
5. Preorder traverse the right subtree.



25, 12, 5, 2, 6, 14, 51

BST Traversals: Postorder

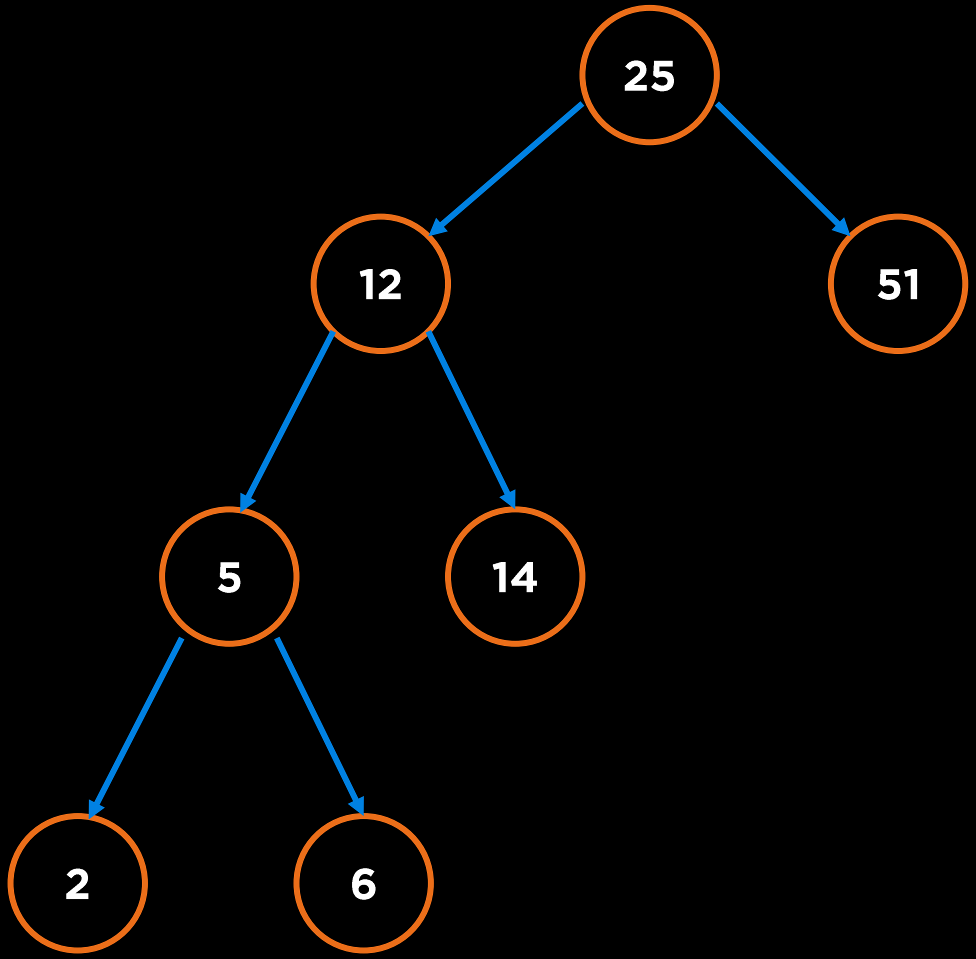
Strategy

- Visit Left Subtree
- Visit Right Subtree
- Visit Root

Algorithm for Postorder Traversal

1. `if` the tree is empty
2. Return.

   `else`
3. Postorder traverse the left subtree.
4. Postorder traverse the right subtree.
5. Visit the root.



BST Traversals: Postorder

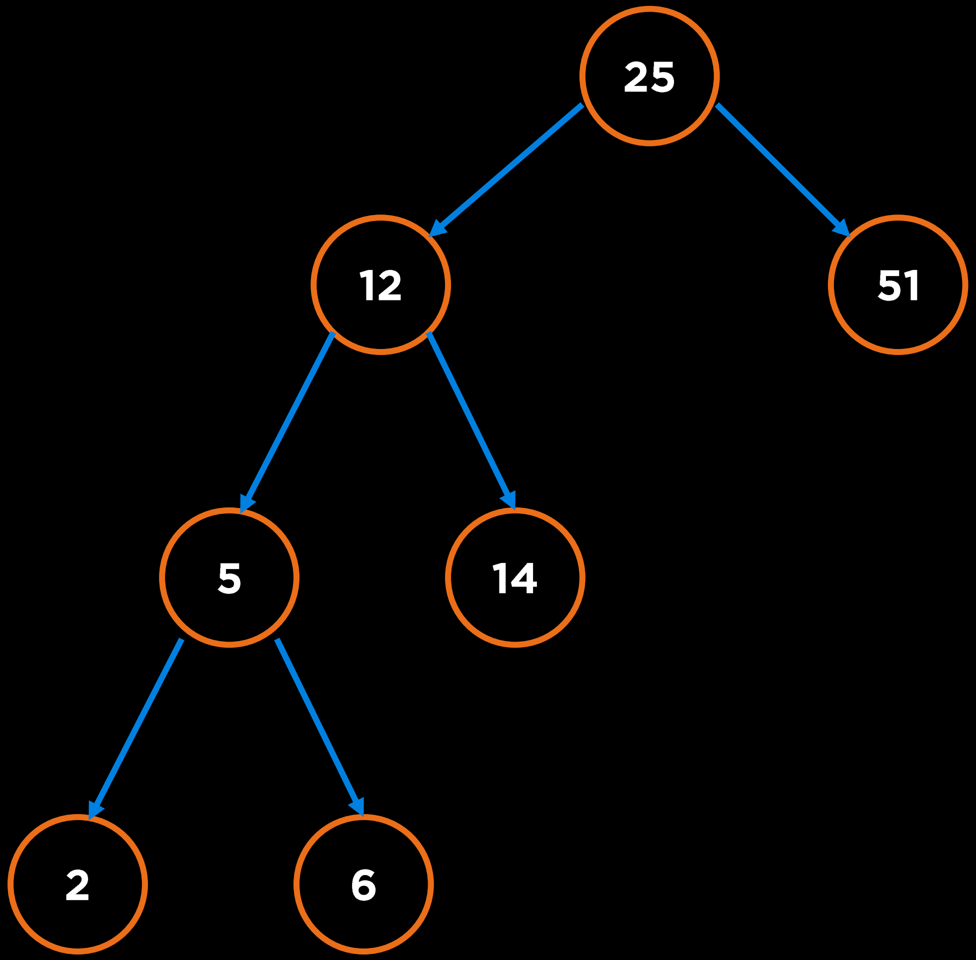
Strategy

- Visit Left Subtree
- Visit Right Subtree
- Visit Root

Algorithm for Postorder Traversal

1. `if` the tree is empty
2. Return.

   `else`
3. Postorder traverse the left subtree.
4. Postorder traverse the right subtree.
5. Visit the root.

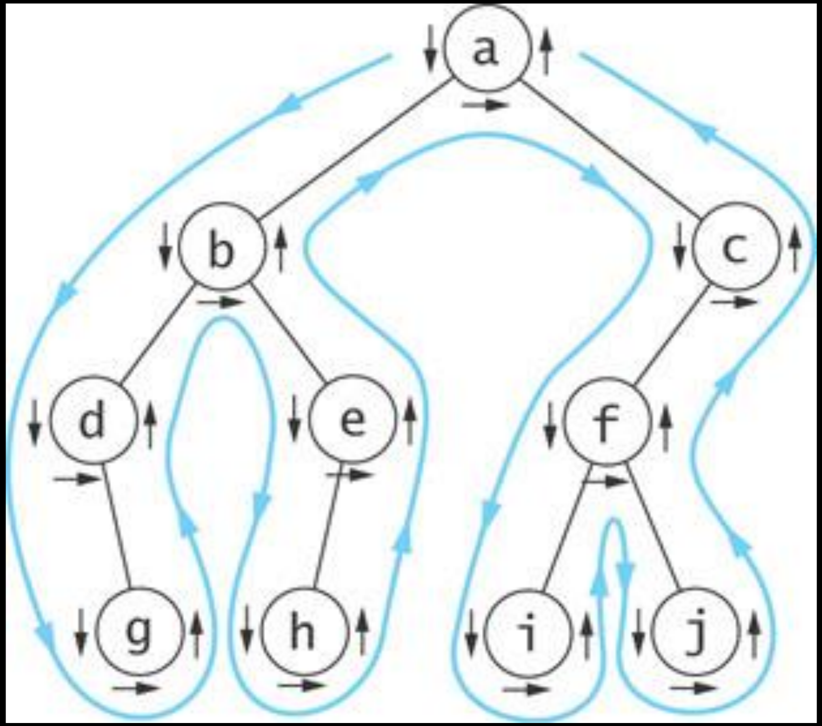


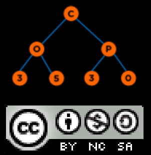
`2, 6, 5, 14, 12, 51, 25`

BST Traversals: Euler Tour

Visiting each node in the tree

- Depth First Strategy
  - Preorder (down arrow)
  - Inorder (horizontal arrow)
  - Postorder (up arrow)

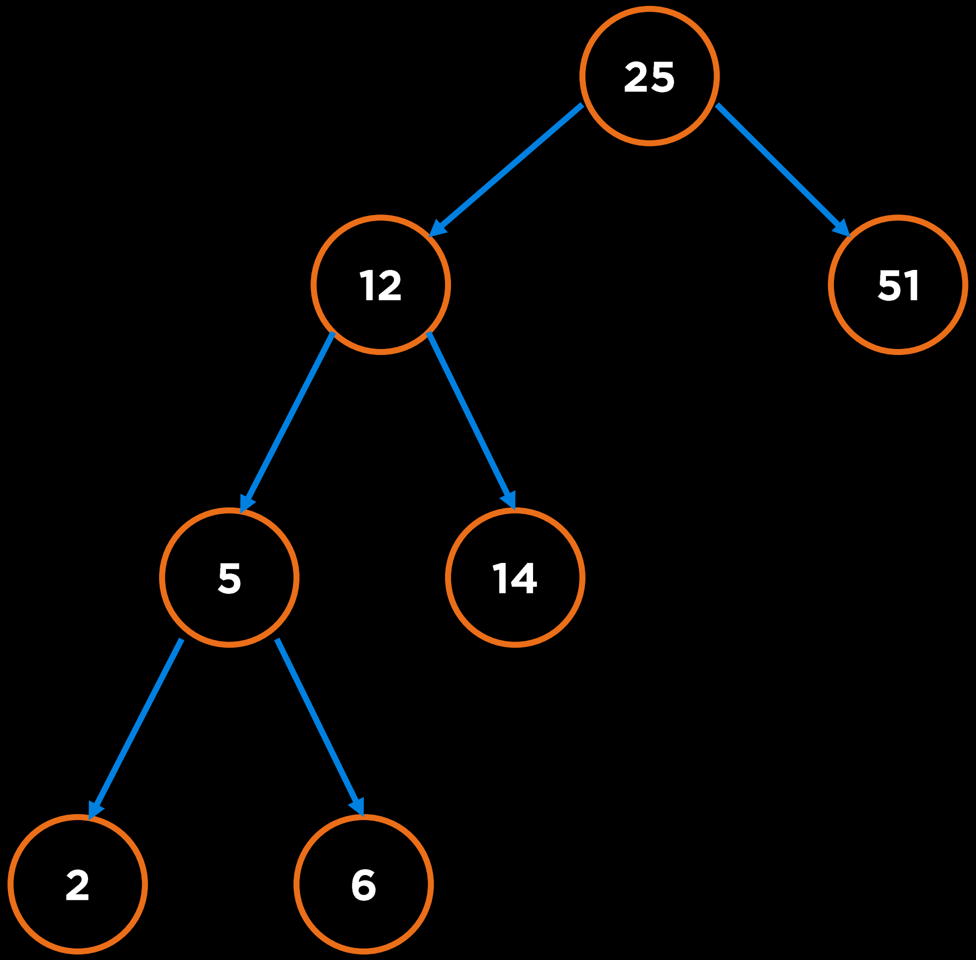




BST Traversals: Level Order

Strategy

- Traverse all nodes in Level 0 upto n-1



25, 12, 51, 5, 14, 2, 6

Use Cases of Traversal

Binary Tree: Sum of Right Leaves

Binary Tree: Sum of Right Leaves

```
void sumOfRightLeaves(TreeNode* root)
{
  queue<TreeNode*> q;
  int sum = 0;

  if(root != NULL)
      q.push(root);

  while (!q.empty())
  {



      if (q.front()->left != NULL)
        q.push(q.front()->left);

      if (q.front()->right != NULL)
        q.push(q.front()->right);
      q.pop();
  }
  cout << sum;
}
```

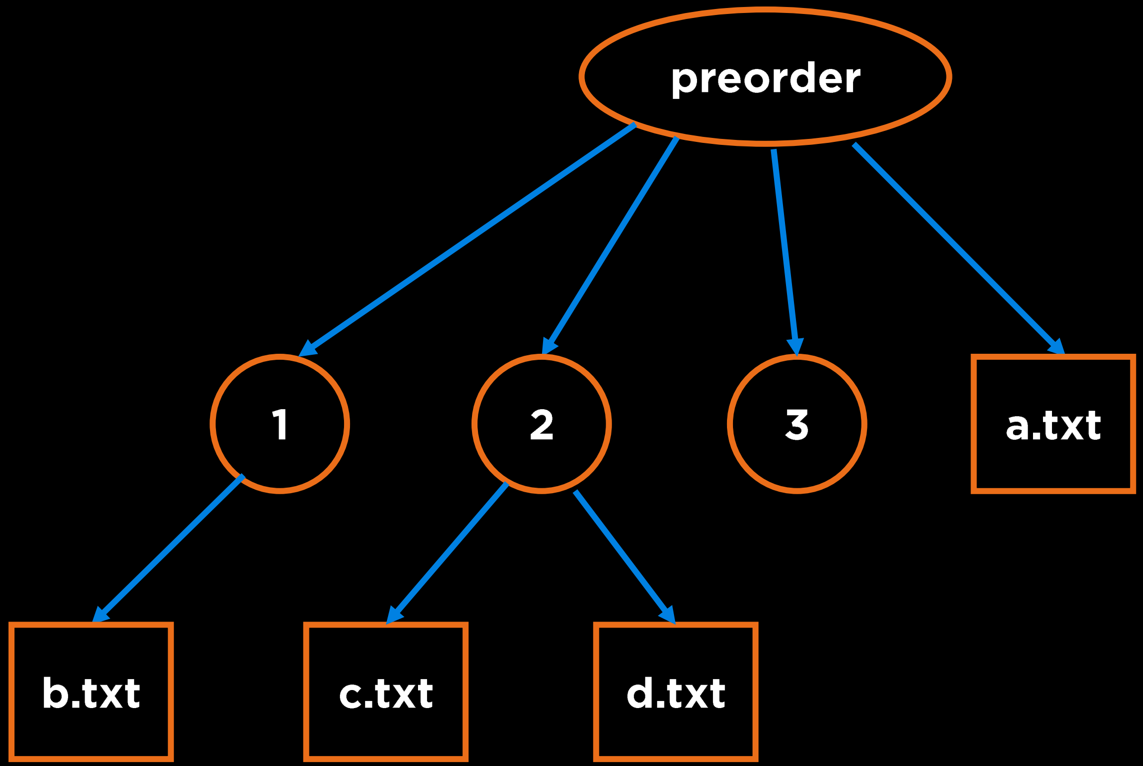
Binary Tree: Sum of Right Leaves

```cpp
void sumOfRightLeaves(TreeNode* root)
{
  queue<TreeNode*> q;
  int sum = 0;

  if(root != NULL)
      q.push(root);

  while (!q.empty())
  {
      if (q.front()->right != NULL && q.front()->right->right == NULL && q.front()->right->left == NULL)
        sum += q.front()->right->val;

      if (q.front()->left != NULL)
        q.push(q.front()->left);

      if (q.front()->right != NULL)
        q.push(q.front()->right);
      q.pop();
  }
  cout << sum;
}
```

Trees Traversal: Use Cases

- **Preorder Traversal**
    - **Printing Directory Listings**

```
PS C:\Users\kapoo\Desktop\COP3530> cd .\preorder\
PS C:\Users\kapoo\Desktop\COP3530\preorder> ls


    Directory: C:\Users\kapoo\Desktop\COP3530\preorder


Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
d-----          6/2/2020  11:12 PM                1
d-----          6/2/2020  11:12 PM                2
d-----          6/2/2020  11:11 PM                3
-a----          6/2/2020  11:12 PM              0 a.txt
```

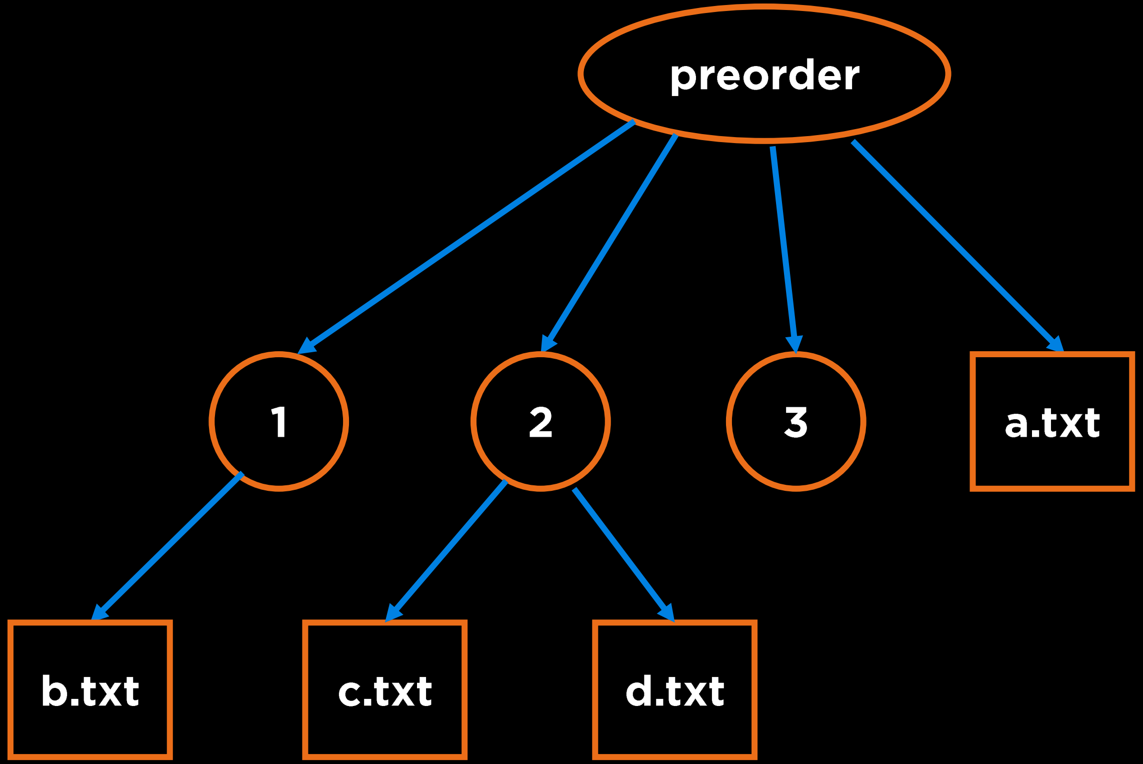


https://sp19.datastructur.es/

Trees Traversal: Use Cases

- **Preorder Traversal**
  - Printing Directory Listings

```
PS C:\Users\kapoo\Desktop\COP3530\preorder> ls -r


    Directory: C:\Users\kapoo\Desktop\COP3530\preorder


Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
d-----          6/2/2020  11:12 PM                1
d-----          6/2/2020  11:12 PM                2
d-----          6/2/2020  11:11 PM                3
-a----          6/2/2020  11:12 PM              0 a.txt


    Directory: C:\Users\kapoo\Desktop\COP3530\preorder\1


Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
-a----          6/2/2020  11:12 PM              0 b.txt


    Directory: C:\Users\kapoo\Desktop\COP3530\preorder\2


Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
-a----          6/2/2020  11:12 PM              0 c.txt
-a----          6/2/2020  11:12 PM              0 d.txt
```

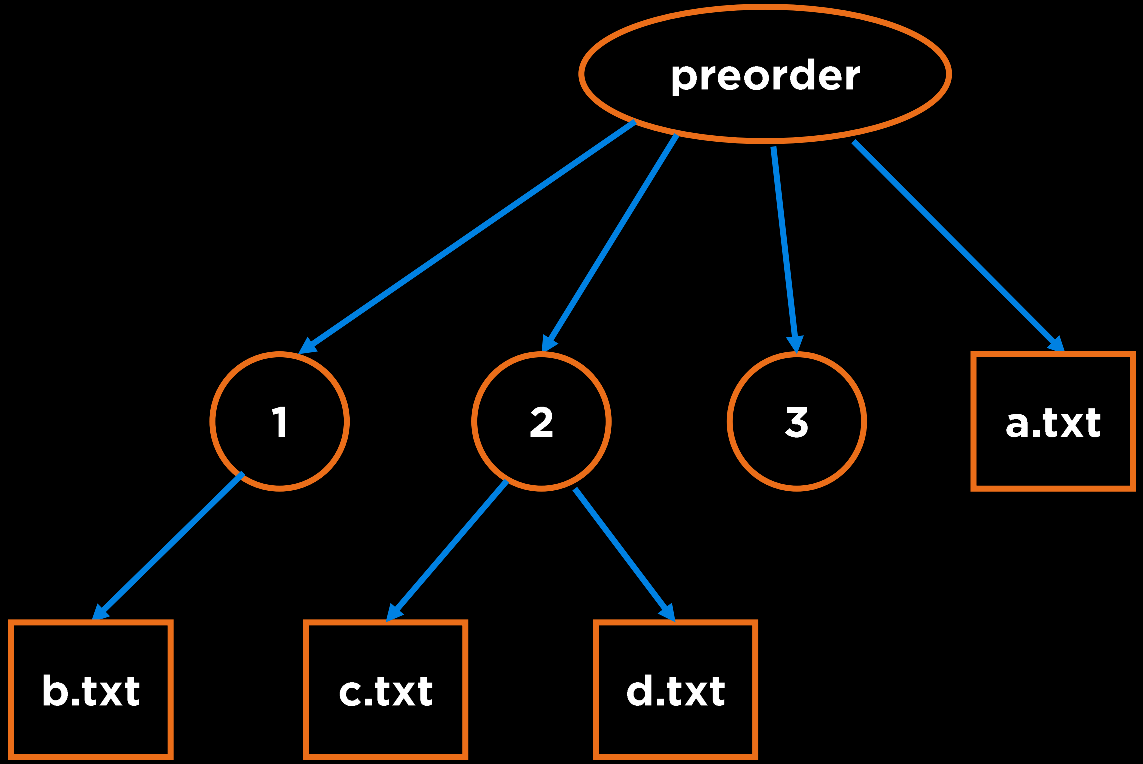


https://sp19.datastructur.es/

Trees Traversal: Use Cases

- **Preorder Traversal**
  - **Printing Directory Listings**

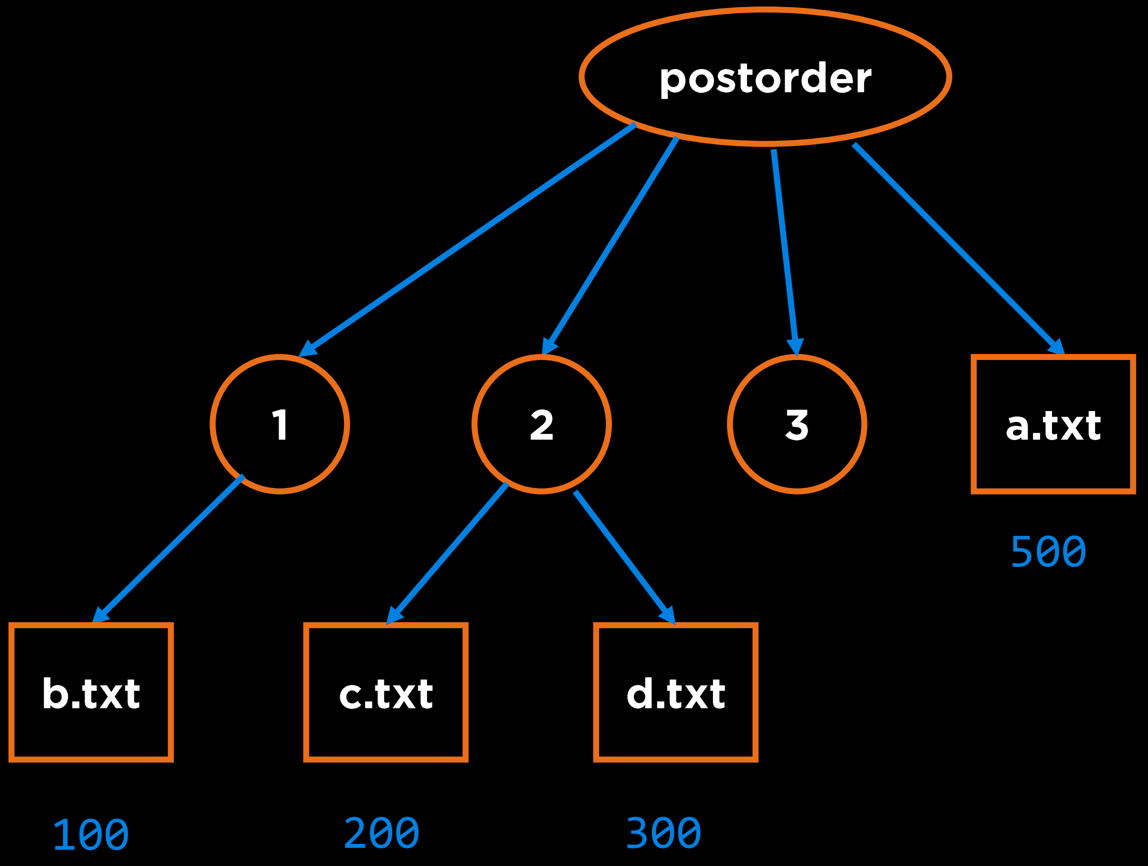
```
preorder/
    1/
        b.txt
    2/
        c.txt
        d.txt
    3/
    a.txt
```



Trees Traversal: Use Cases

- **Postorder Traversal**
  - Gathering File Sizes

```
    1/
        100
    2/
        200
        300
    3/
    500
postorder/
```

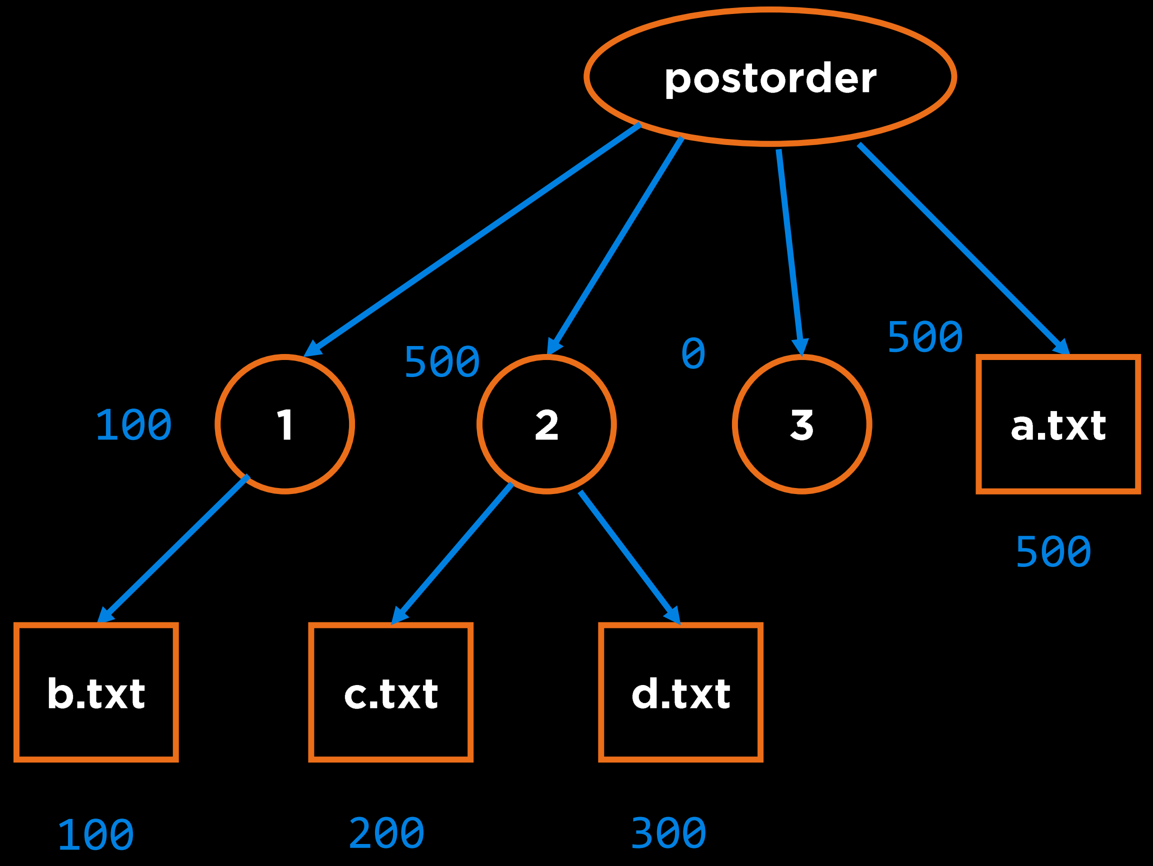


https://sp19.datastructur.es/

Trees Traversal: Use Cases

- **Postorder Traversal**
  - Gathering File Sizes

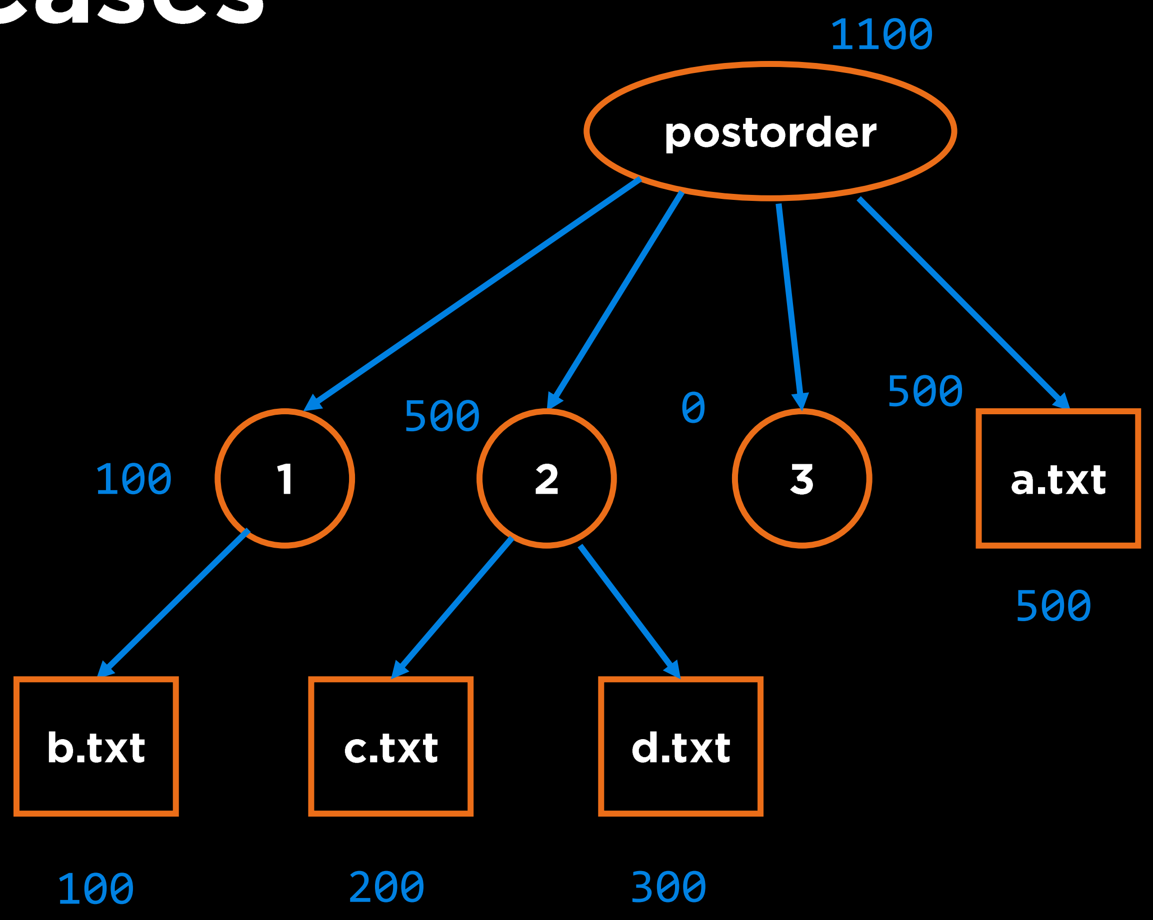
```
1/
    100
2/
    200
    300
3/
500
postorder/
```



Trees Traversal: Use Cases

- **Postorder Traversal**
  - Gathering File Sizes

```
  1/
      100
  2/
      200
      300
  3/
  500
postorder/
```



Binary Search Tree: Size of directories

```
class TreeNode {
     public:
       int size;
       char name;
       TreeNode *left;
       TreeNode *right;
       TreeNode(char n, int s) : size(s), name(n), left(nullptr), right(nullptr) {}
 };

int postOrder(TreeNode* head)
{
    if(head == NULL)
        return 0;

    int total = 0;
    total += postOrder(head->left);
    total += postOrder(head->right);
    total += head -> size;
    return total;
}
```

https://onlinegdb.com/_zZnXu1X5u

Trees Traversal: Other Use Cases

- **Postorder Traversal**
  - Managing memory when deleting a tree

- **Preorder Traversal**
  - Creating a copy of a tree

Mentimeter

Questions

n-Ary Tree: Level Order Traversal

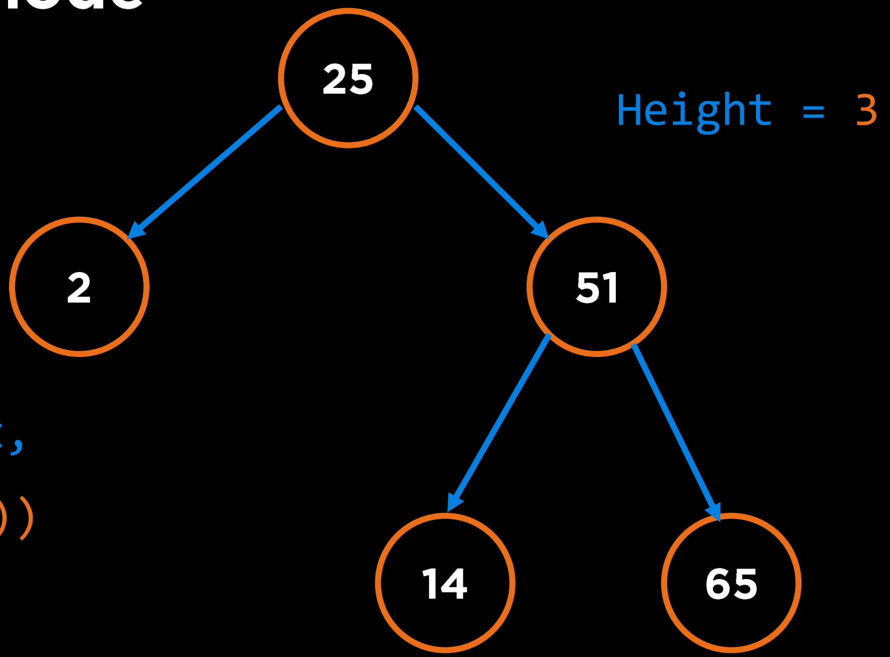
```
void levelOrder(Node* root) {
    int level = 1;
    queue<Node*> q;
    q.push(root);

    while (!q.empty())
    {
        vector<int> vec;
        int size = q.size();
        for (int i = 0; i < size; i++)
        {
            Node* temp = q.front();
            vec.push_back(temp->val);
            q.pop();
            for (auto element : temp->children)
                   q.push(element);
        }
        cout << "Level: " << level++ << endl;
        for (int i = 0; i < vec.size(); i++ )
            cout << vec[i] << " ";
        cout << endl;
    }
}
```

More Properties Related to Height

Trees: Terminology

Height

The height of a tree is the number of nodes in the longest path from the root node to a leaf node

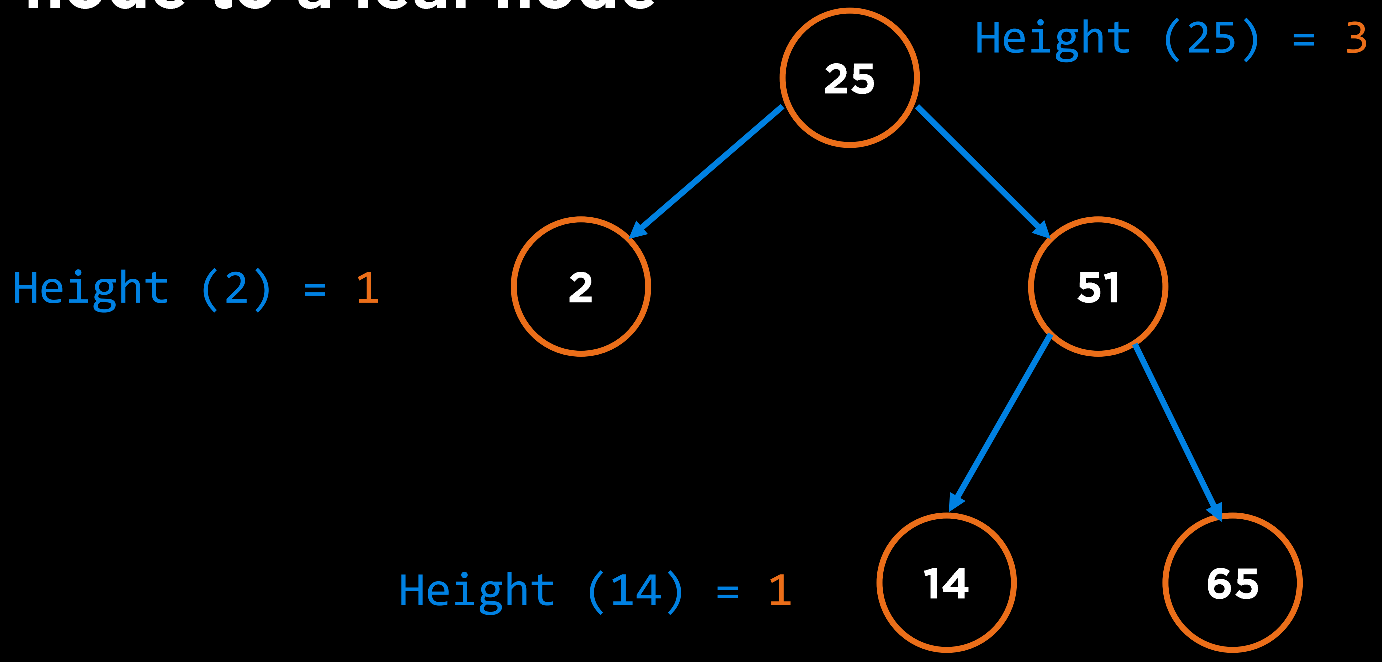



```
If tree has just the root,
        Height = 1

If tree has more nodes than the root,
    Height = 1 + max(Height(children))
```

Trees: Terminology

Height of a Node

The height of a node is the number of nodes in the longest path from the node to a leaf node



Trees: Relationship between Height and Number of Nodes

Minimum number of Nodes in a Tree with Height, h

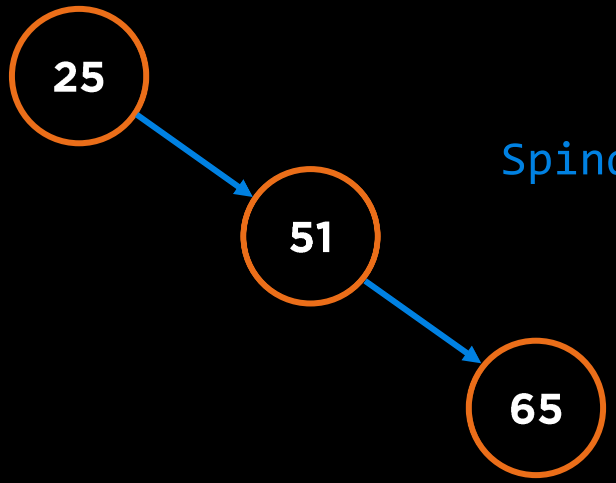
Trees: Relationship between Height and Number of Nodes

Minimum number of Nodes in a Tree with Height, h

If height = h, at least one node at each level, therefore

Number of Nodes, n = h

Height, h = 3



Spindly Tree

Trees: Relationship between Height and Number of Nodes

Maximum number of Nodes in a Tree with Height, h

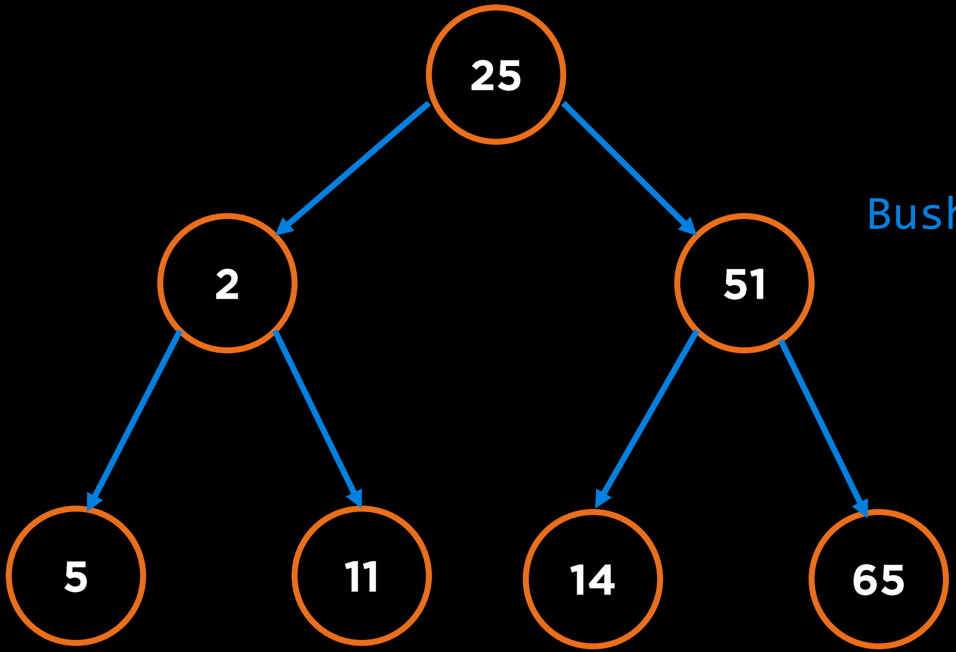
Trees: Relationship between Height and Number of Nodes

Maximum number of Nodes in a Tree with Height, h

If height = h, all possible node at each level, therefore

Number of Nodes, n = $1 + 2 + 4 + \ldots + 2^{h-1} = 2^h - 1$

Height, h = 3
Perfect Binary Tree



Bushy Tree

https://en.wikipedia.org/wiki/1_%2B_2_%2B_4_%2B_8_%2B_%E2%8B%AF

Trees: Relationship between Height and Number of Nodes

Number of Nodes is between h and $2^h - 1$

$$h <= n <= 2^h - 1$$

$$n + 1 <= 2^h$$

$$\log (n + 1) <= \log (2^h)$$

$$\log (n + 1) <= \log (2^h)$$

$$\log (n + 1) <= h$$

Height of a Tree, h is between $\log_2 (n+1)$ and n

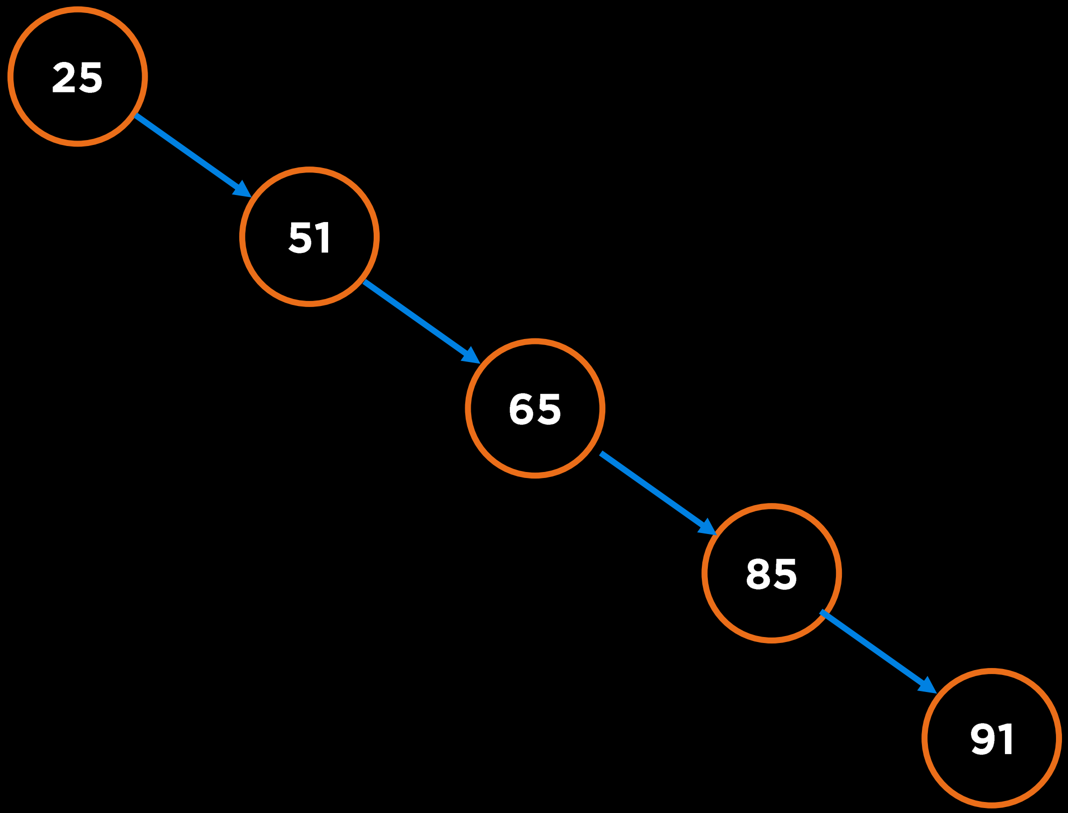
Trees Height: Takeaway

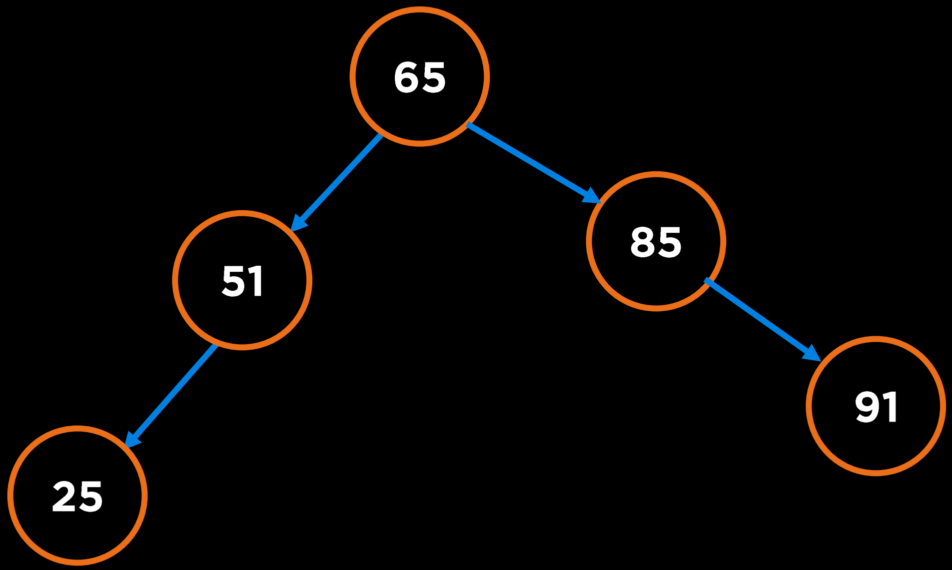
- Trees with n nodes can have a height that is proportional to n or proportional to log (n).
- For good performance we want a tree that is as perfect as possible or as "bushy" as possible, i.e. height ~ log (n)

BST Performance

BST Insertion

n! different ways to insert n elements



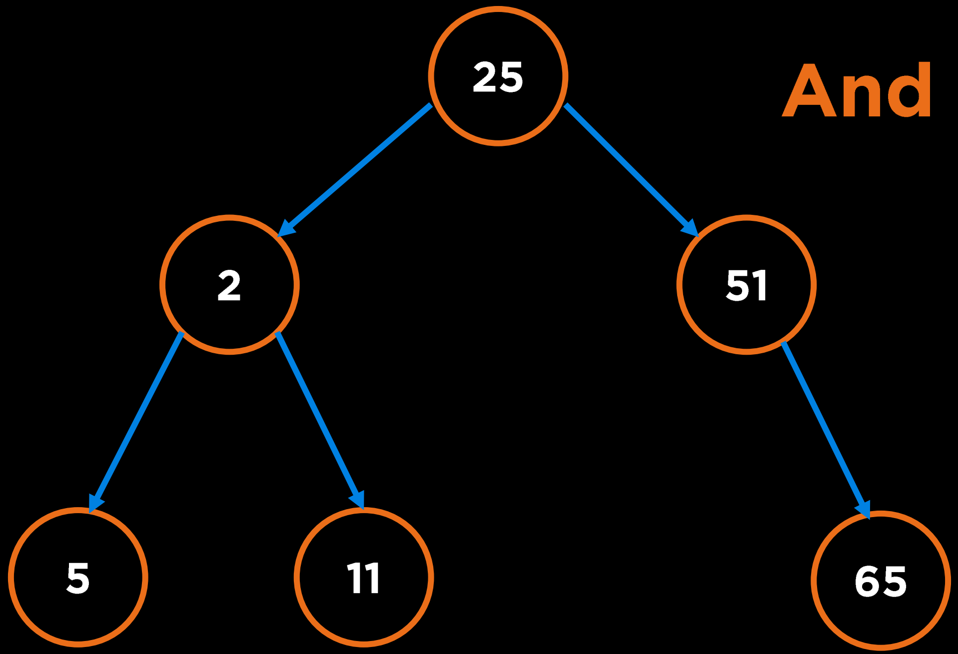


BST Insert, Delete and Search

Worst Case ~ Height = n

And Common Operations will be O(n)

BST Insert, Delete and Search

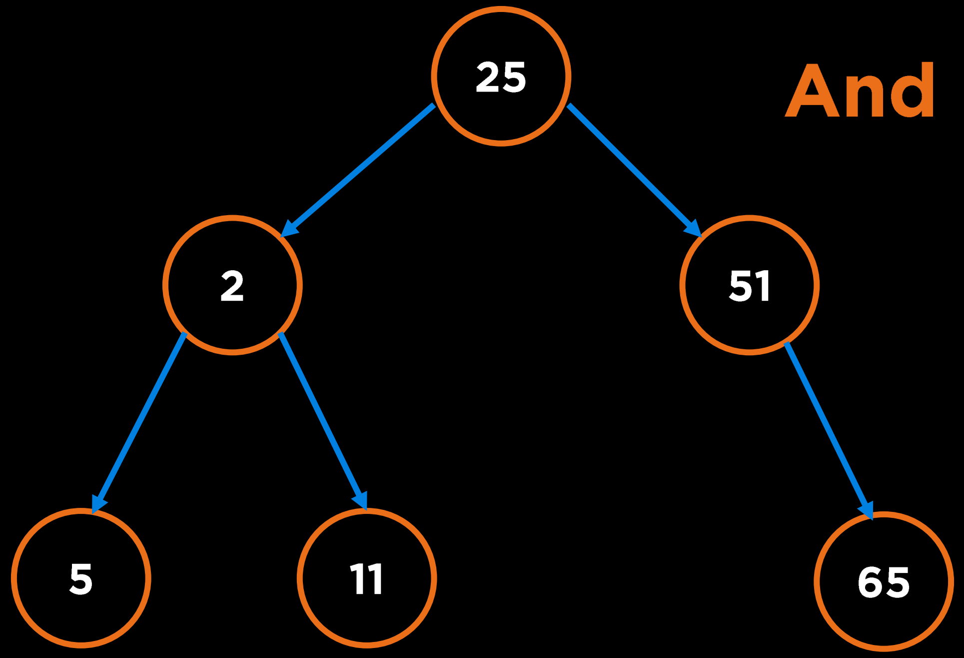


Average Case ~ Height = log n

And Common Operations will be O(log n)

Reed, 2003: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.152.1289&rep=rep1&type=pdf

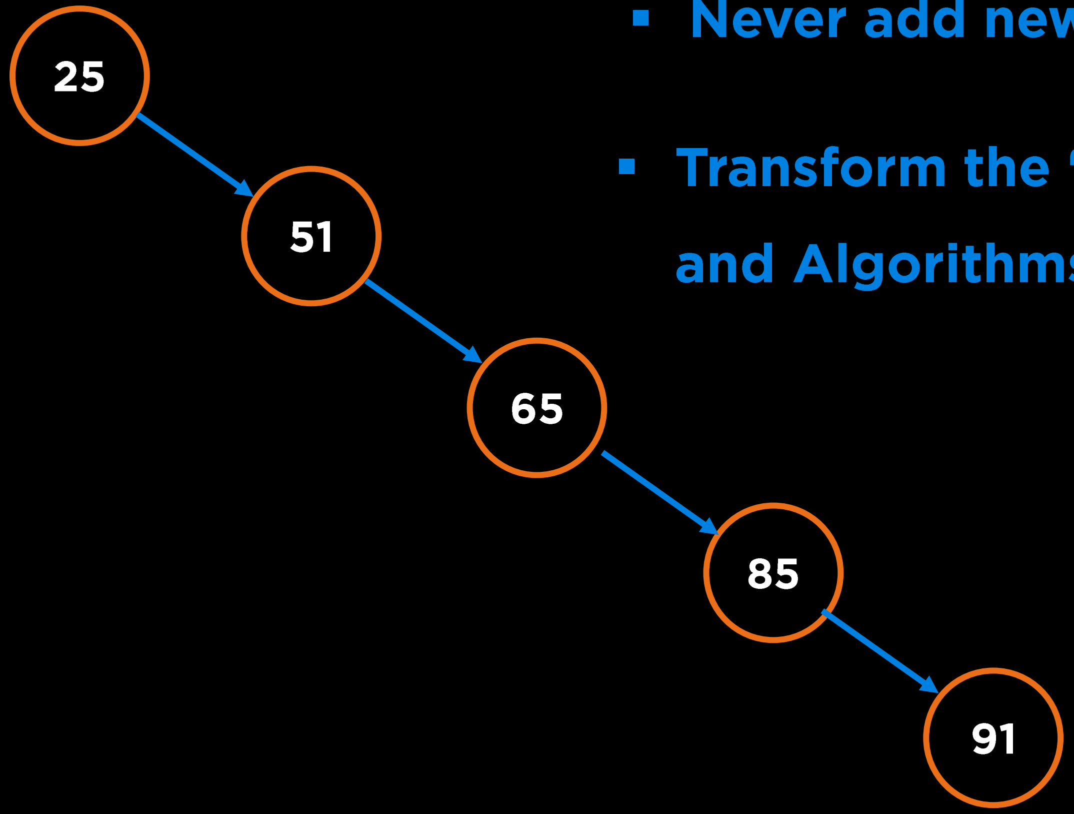
BST Insert, Delete and Search



Best Case ~ Height = log n

And Common Operations will be O(log n)

How do we fix the Worst Case?



- Never add new leaves at the bottom: Increase size of node
- Transform the “Spindly” Tree to “Bushy Tree” using Tools and Algorithms

Questions