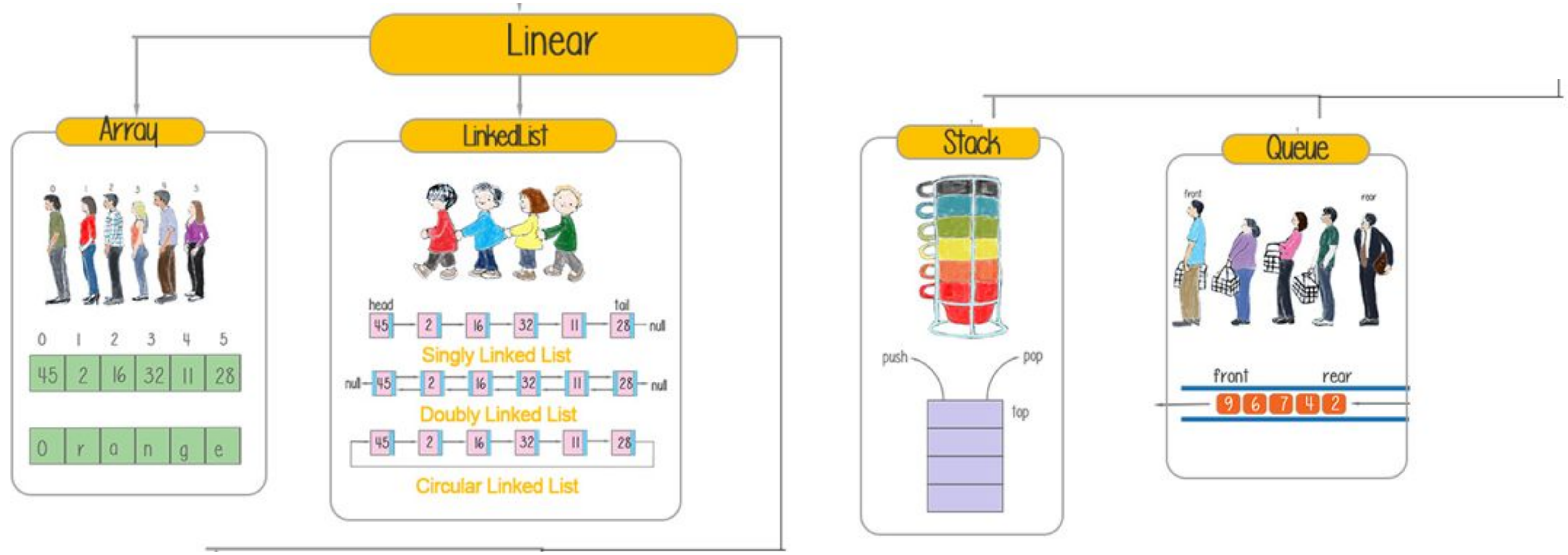
Week 10-11: Tree ADT

CS-250 Data Structure and Algorithms

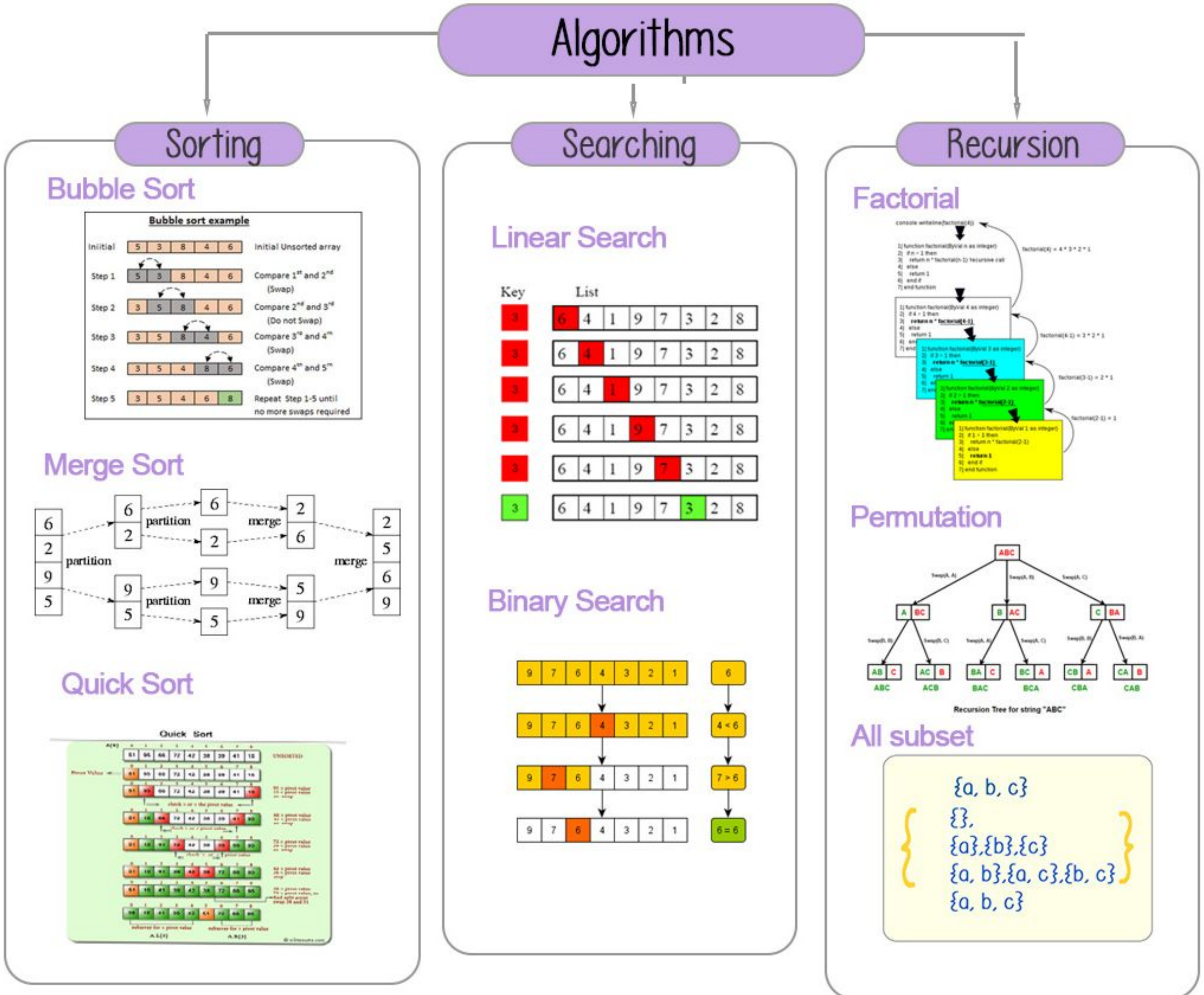
DR. Mehwish Fatima | Assist. Professor
Department of AI & DS | SEECS, NUST

Recap



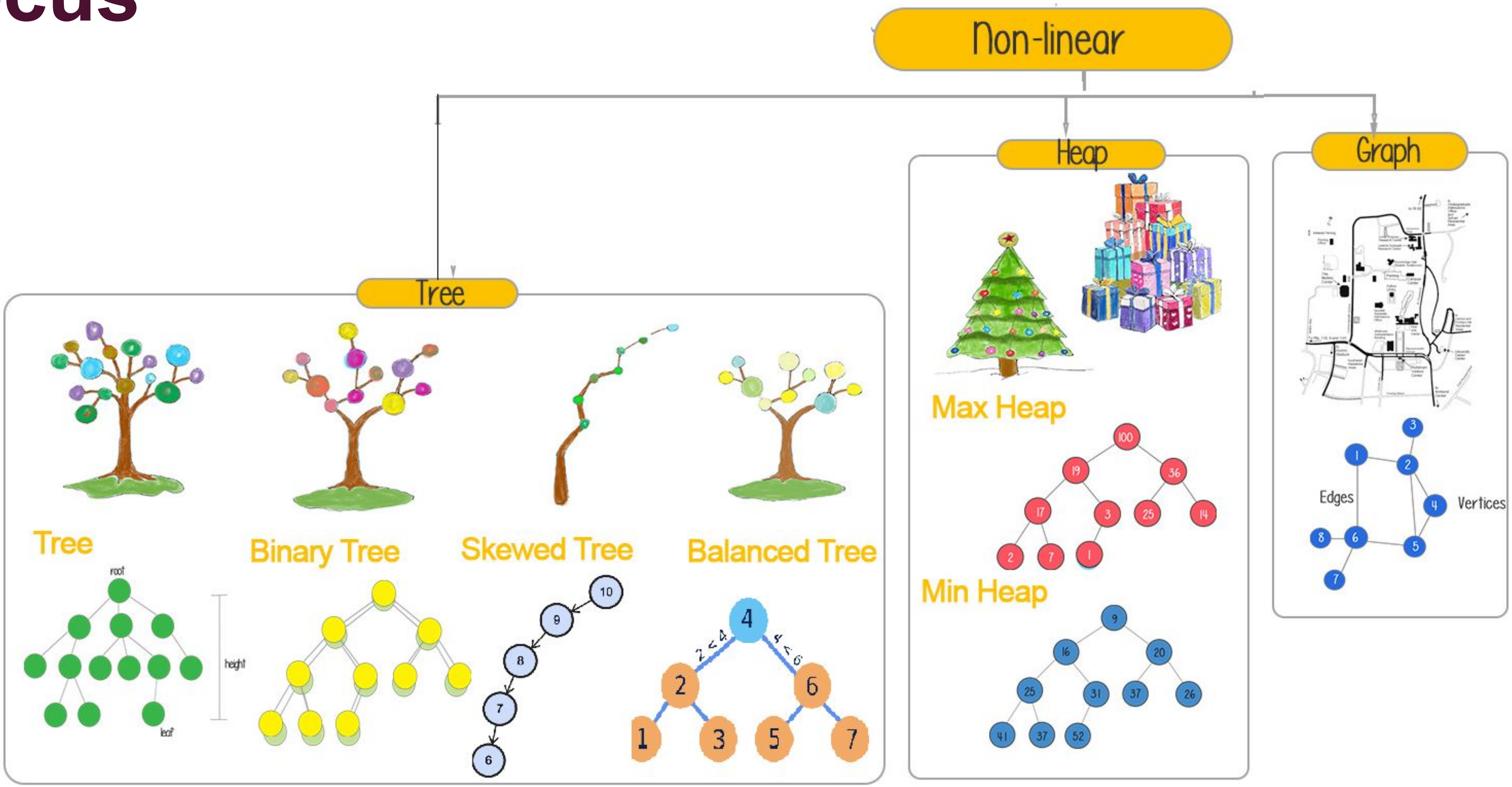
Linear
Array
0 1 2 3 4 5
45 2 16 32 11 28
o r a n g e
LinkedList
head
tail
45 2 16 32 11 28 null
Singly Linked List
null 45 2 16 32 11 28 null
Doubly Linked List
45 2 16 32 11 28
Circular Linked List
Stack
push
pop
top
Queue
front
rear
front
rear
9 6 7 4 2

Recap



Algorithms
Sorting
Searching
Recursion
Bubble Sort
Bubble sort example
Initial
Initial Unsorted array
Step 1
Step 2
Step 3
Step 4
Step 5
Repeat Step 1-5 until no more swaps required
Merge Sort
partition
merge
Quick Sort
Quick Sort
Linear Search
Key
List
Binary Search
Factorial
Permutation
Recursion Tree for string "ABC"
All subset
{a, b, c}
{},
{a},{b},{c}
{a, b},{a, c},{b, c}
{a, b, c}

Focus



Non-linear
Tree
Heap
Graph
Tree
Binary Tree
Skewed Tree
Balanced Tree
root
height
leaf
Max Heap
Min Heap
Edges
Vertices

Linear vs. Non-Linear Data Structures

- Linear vs non-linear classification of data structures is dependent upon how individual elements are connected to each other.
  - All linear data structures have one thing in common that they are sequential
  - Not efficient for information retrieval →O(n)

- Linear lists are useful for serially ordered data.
  - (e0, e1, e2, …, en-1)
  - Days of week
  - Months in a year
  - Students in this class

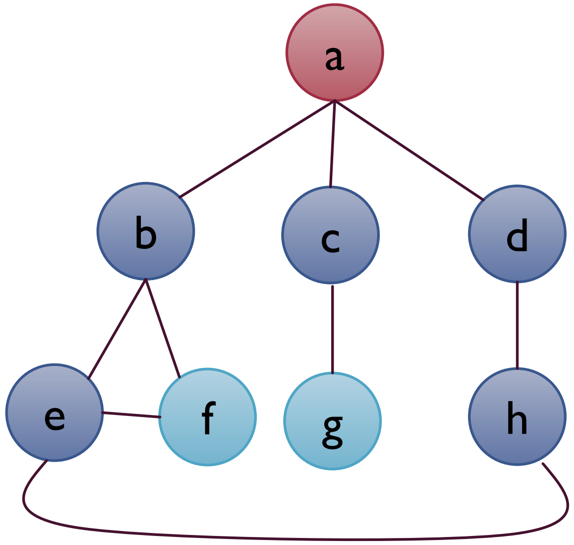
Linear vs. Non-Linear Data Structures

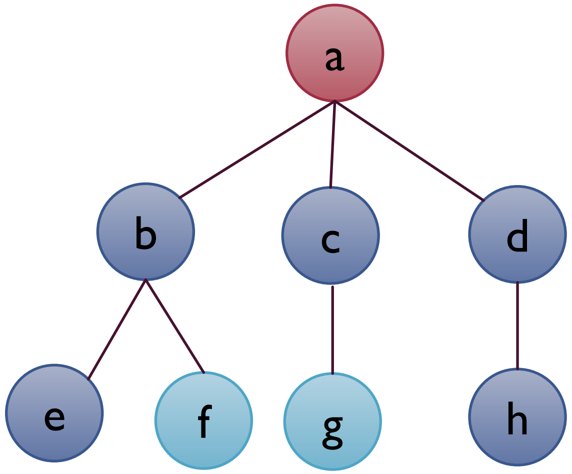
- Linear vs non-linear classification of data structures is dependent upon how individual elements are connected to each other.
  - All linear data structures have one thing in common that they are sequential
  - Not efficient for information retrieval →O(n)

- Trees are useful for hierarchically ordered data
  - Structure of directories
  - Employees of a corporation
  - President, vice presidents, managers, and so on

Tree vs. Graph

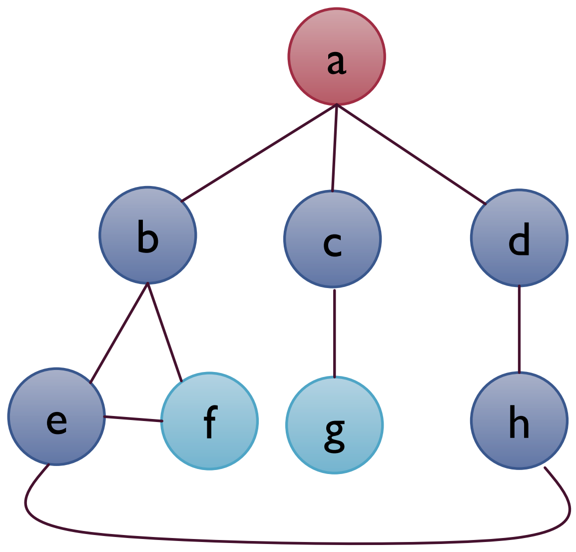
- A graph is non-linear data structure which is use to model complex problems.
  - Like finding shortest path from on city to other city in maps
  - It has set of vertices/nodes and edges. Vertices are connected through edges

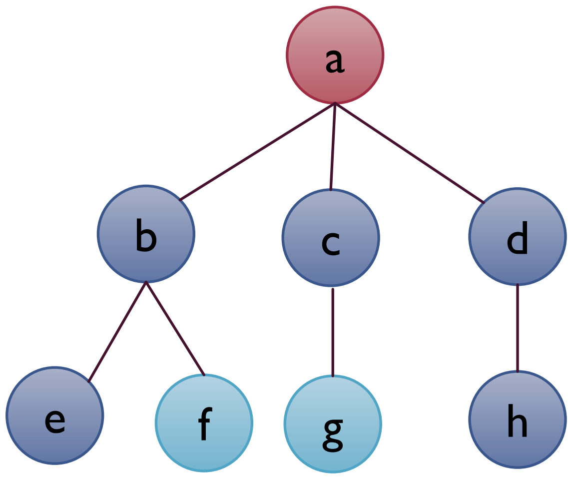




Tree vs. Graph

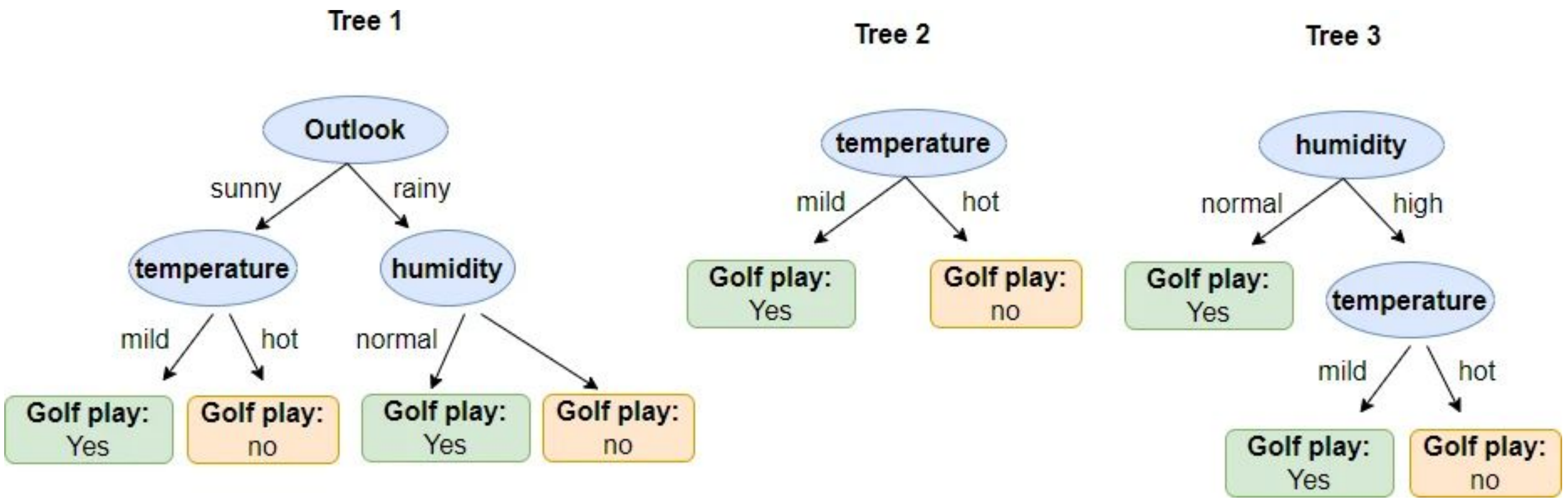
- A tree is a graph with no cycles.
  - Cycle means a path which starts and ends at same node (a-b-e-h-d-a)





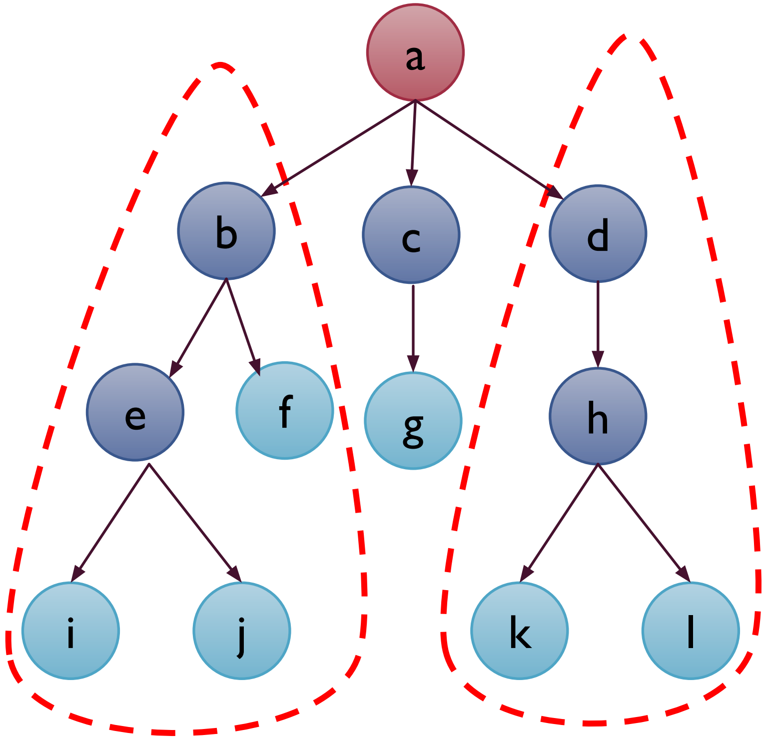
Tree

- A non linear data structure which presents hierarchical relationship between data elements.
  - Hierarchical means some elements are below and some are above from others.
  - Like family tree, folder structure, table of contents



Tree: Recursive Data Structure

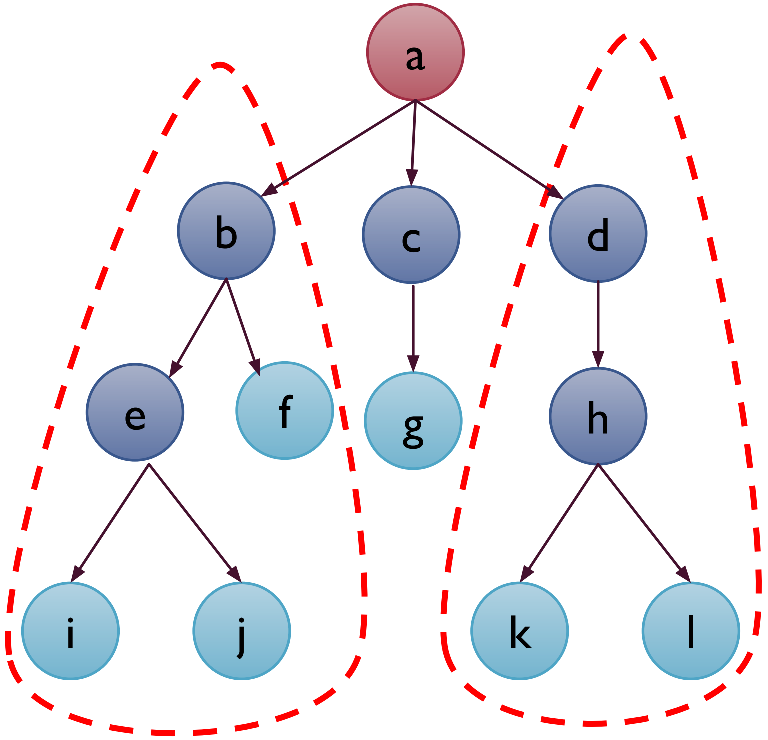
- Tree is a recursive data structure
  - it contain certain patterns that are themselves are trees.
  - it is composed of smaller pieces of its own data type.
  - Such as list and trees.
    - a is root of all nodes like b, c, d etc.
    - b, c, d are also root of their sub trees



Tree: Recursive Data Structure

So, a tree T can be defined recursively as

- Tree T is a collection of nodes such that
  - T is empty/NULL (No node) OR
  - There is a special node called root,
    - which can have 0 or more children (T1, T2, T3 …Tn)
    - which are also sub-trees themselves.
    - T1, T2, T3 …Tn are disjoint subtrees (no shared node)



Why Trees

- They are suitable for
  - Hierarchical structure representation, e.g.,
    - File directory.
    - Organizational structure of an institution.
    - Class inheritance tree.
  - Problem representation, e.g.,
    - Expression tree.
    - Decision tree.
  - Efficient algorithmic solutions, e.g.,
    - Search trees.
    - Efficient priority queues via heaps.

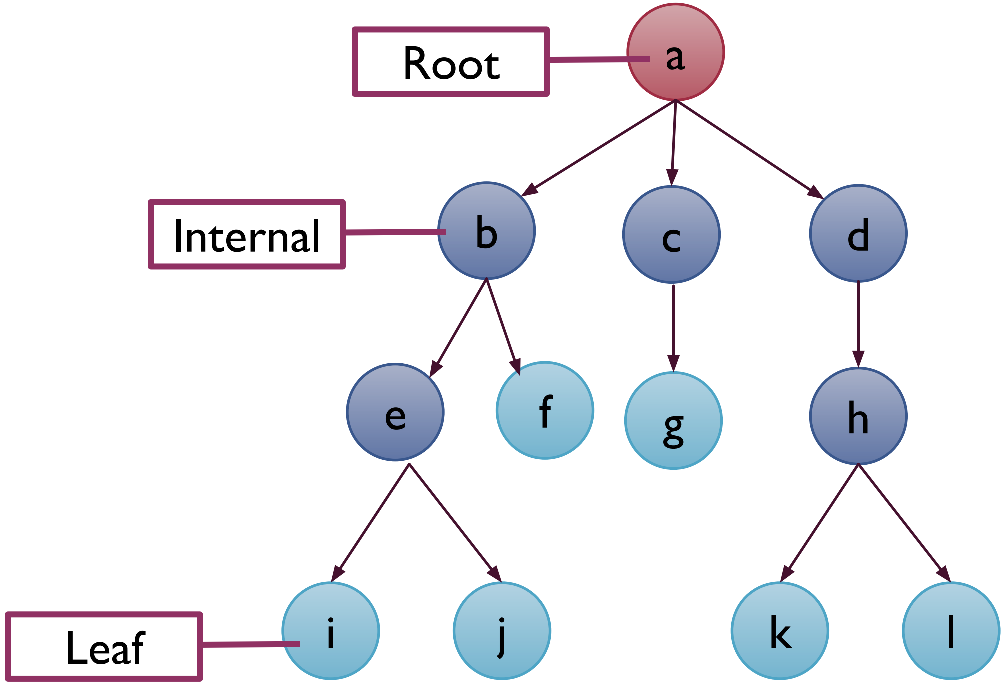
Trees as tool for abstraction

- We often use trees in our everyday lives
  - To keep track of our ancestors: a family tree
    - Most of the terminology used in trees in computer science here
  - When describing the organization of a company: a company chart
  - Organization of files in a computer: a directory structure
- In here we want to concentrate on trees as tools for the implementation of computer programs: trees as abstract data structures

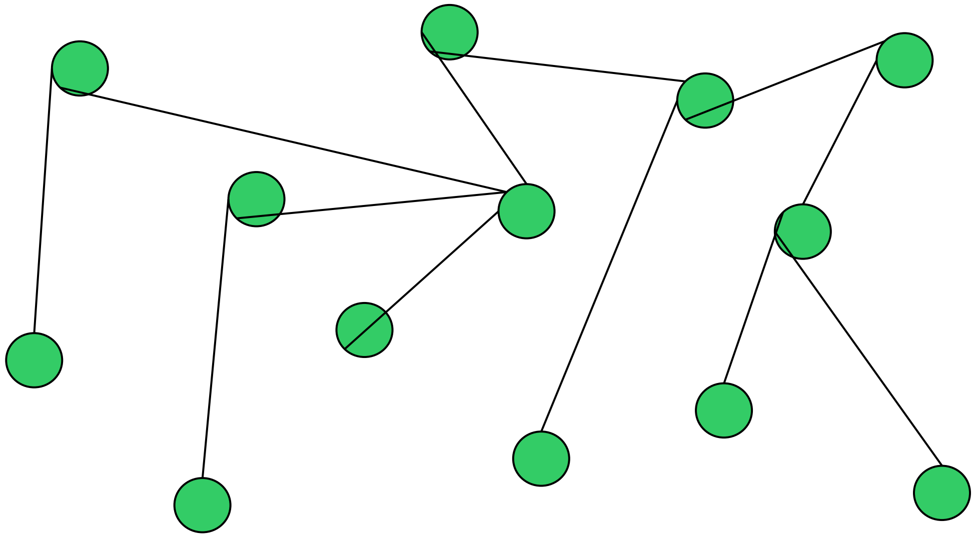
Terminologies

Tree Terminologies

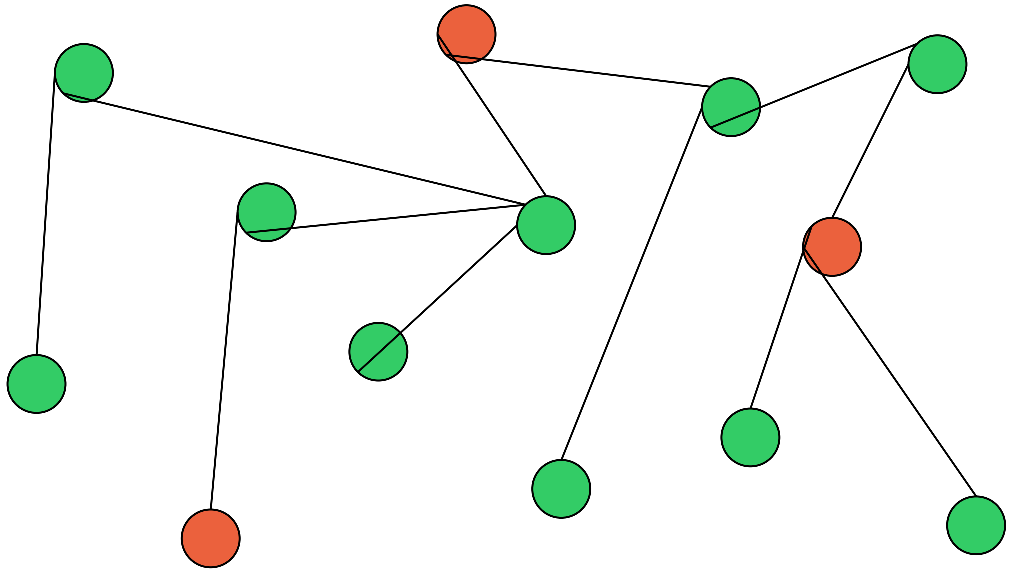
- Node/Vertex
  - One data unit of tree
- Edge
  - Arch/link from one node to other
- Path
  - A sequence of adjacent of vertices connected by edges
- Length of a Path
  - # of edges on the path from one node to other



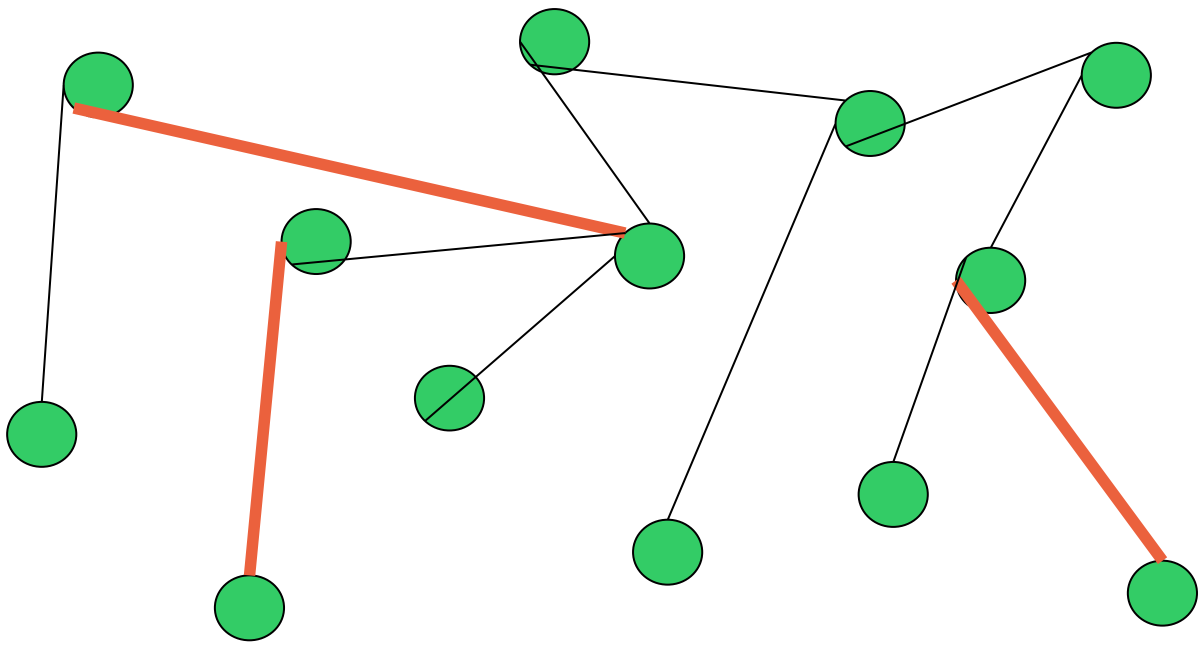
Tree



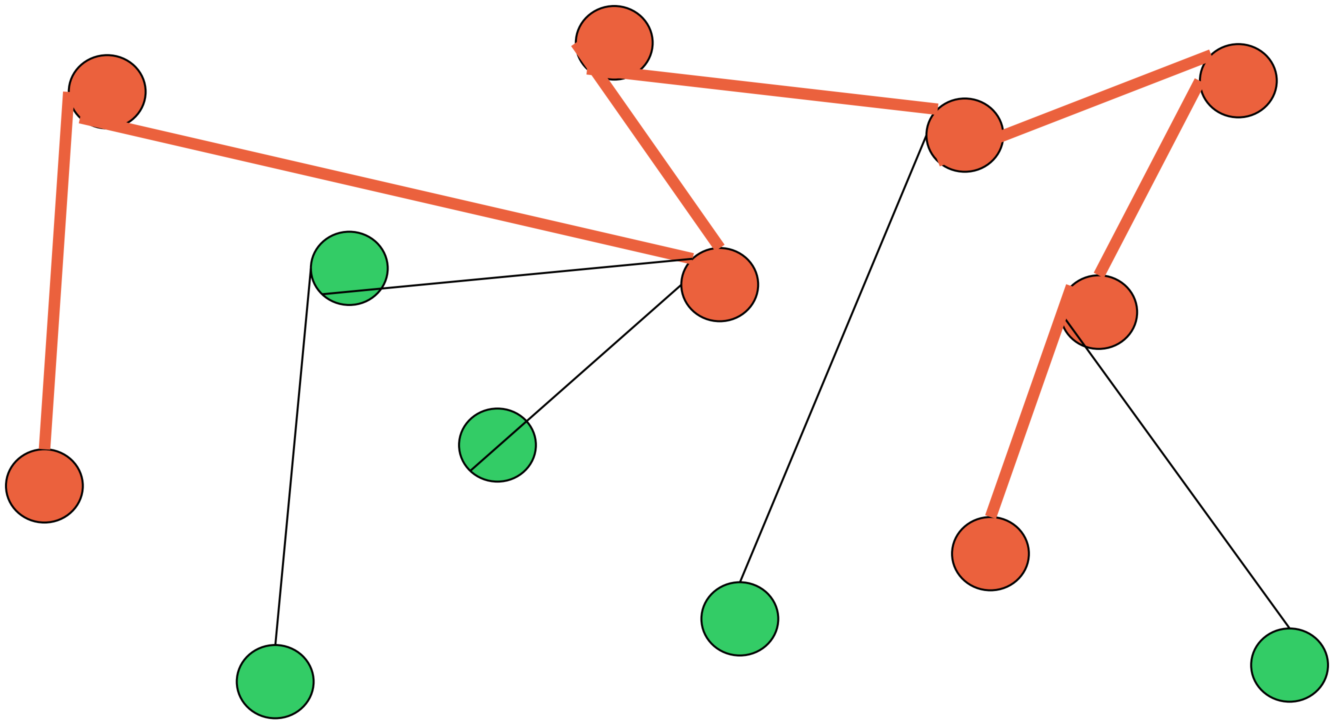
Nodes



Edges



Path



Trees, Paths and Forests

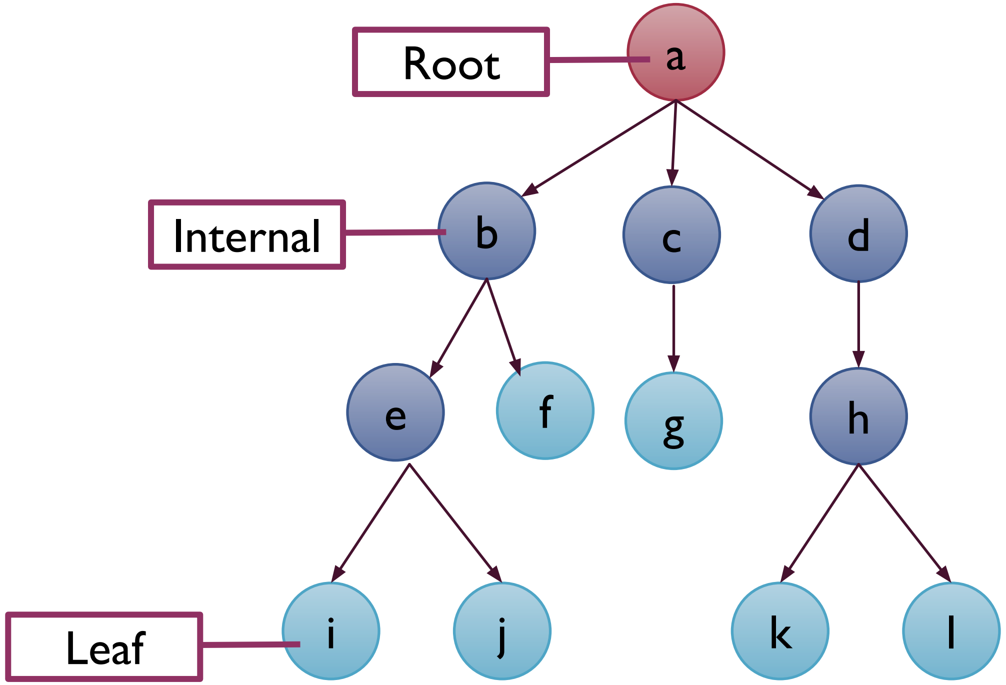
- The main property of a tree is based on paths
  - In a tree there is exactly one path between any two nodes

- If there exist more than one path between any pair of nodes or if there is no path between any of them we do not have a tree

- A disjoint set of trees is called a forest

Tree Terminologies

- Root node
  - The top node of tree.
  - A node with no parent
- Leaf/External node
  - Node with no child
- Internal Node
  - Node with child

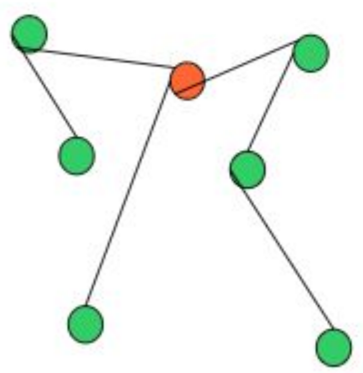


Rooted Trees

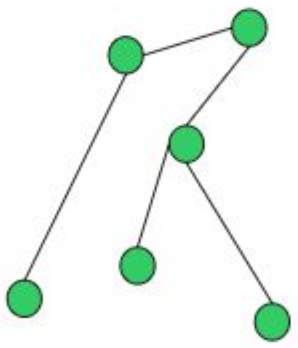
- A rooted tree is a tree where one node is "**special**" and is called the root of the tree
- A tree where there is no root is called a **free tree**

- Rooted trees are the most common in computer applications
  - so common that we'll use the term tree as a synonym for rooted tree

A rooted tree

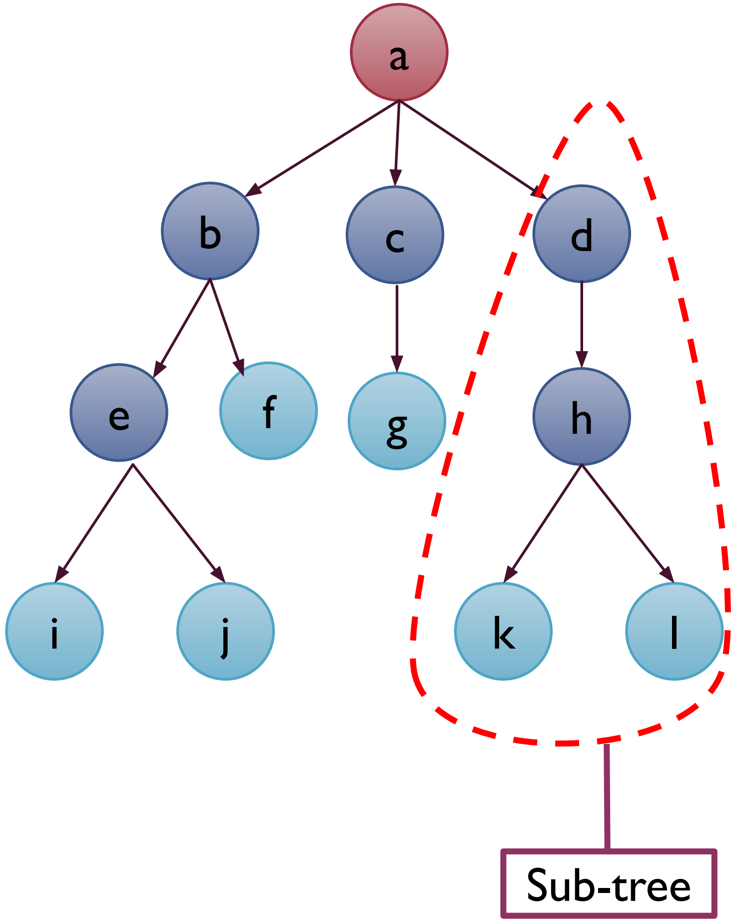


a free tree



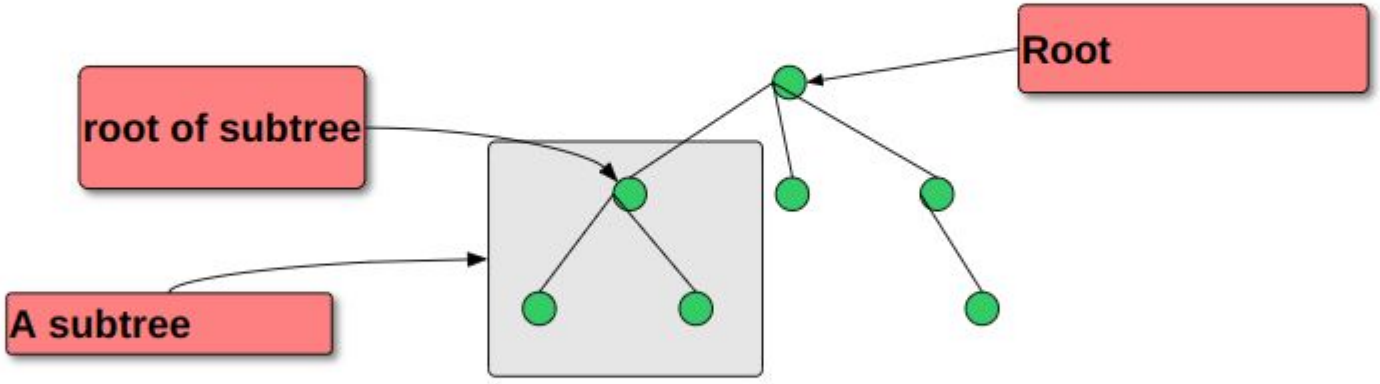
Tree Terminologies

- SubTree
  - A node within tree with descendants



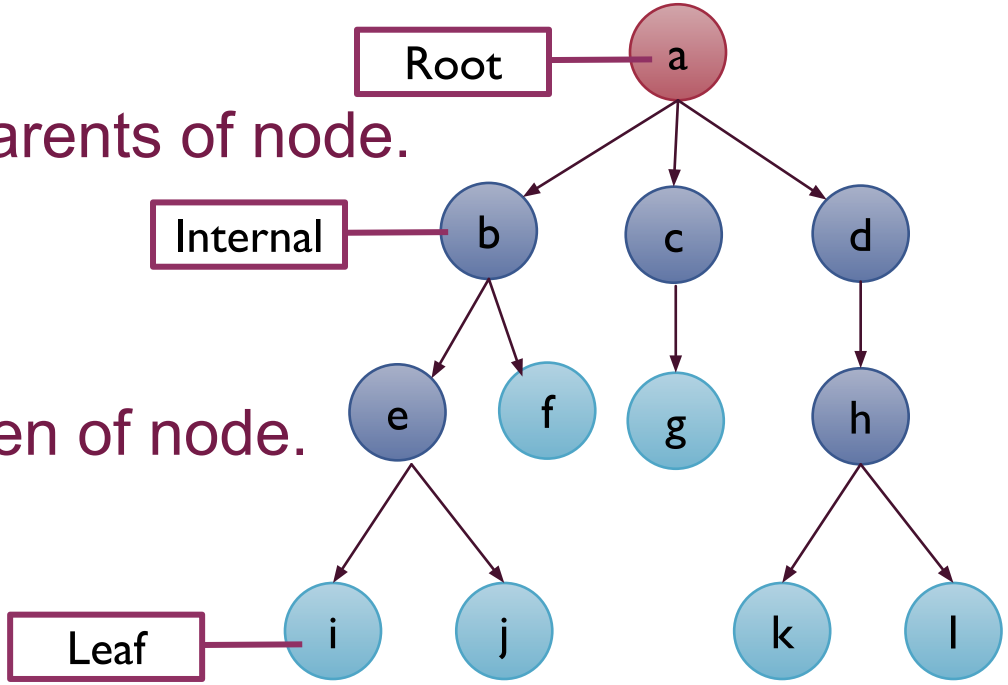
Tree vs. Subtree

- In a rooted tree, any node is a root of a subtree consisting of itself and the nodes "**below**" it
  - By definition there is only one path between the root and each of the other nodes.
    - Because a root is also a node the main property applies



Tree Terminologies

- Ancestor Nodes
  - Parent, all grandparents and all great grand parents of node.
    - a, b and e are ancestors of i.
- Descendant Nodes
  - Child , all grandchildren and great grandchildren of node.
    - i, j, e and f are descendants of b
- Siblings
  - Nodes with same parent and at same level
  - i and j
  - b and c and d



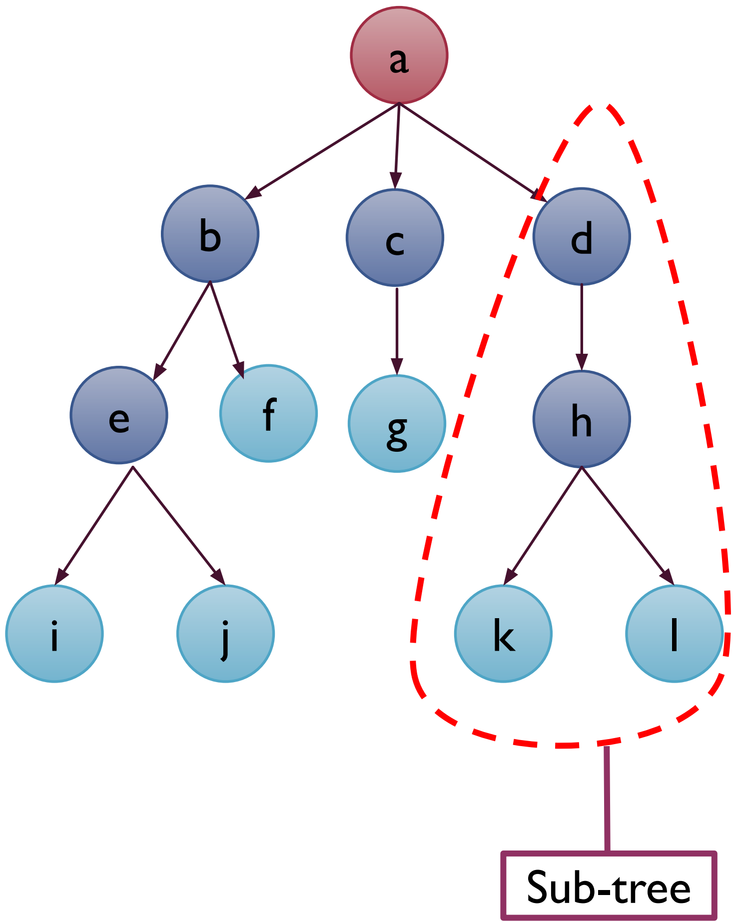
Trees with specific number of children

- It is possible that the order in which children are defined is important.
  - We call these trees **ordered trees**
- Sometimes a node must have a maximum number of children.
  - If for the whole tree the maximum number of children nodes can have is **M** and this tree is ordered we have a **M-ary tree**.
- A **binary tree** is a special case of a M-ary tree where all nodes (except the leaves) have at most 2 children.
  - Because the tree is ordered the children have a order and they are normally referred to as **left** and **right** child

Tree Terminologies

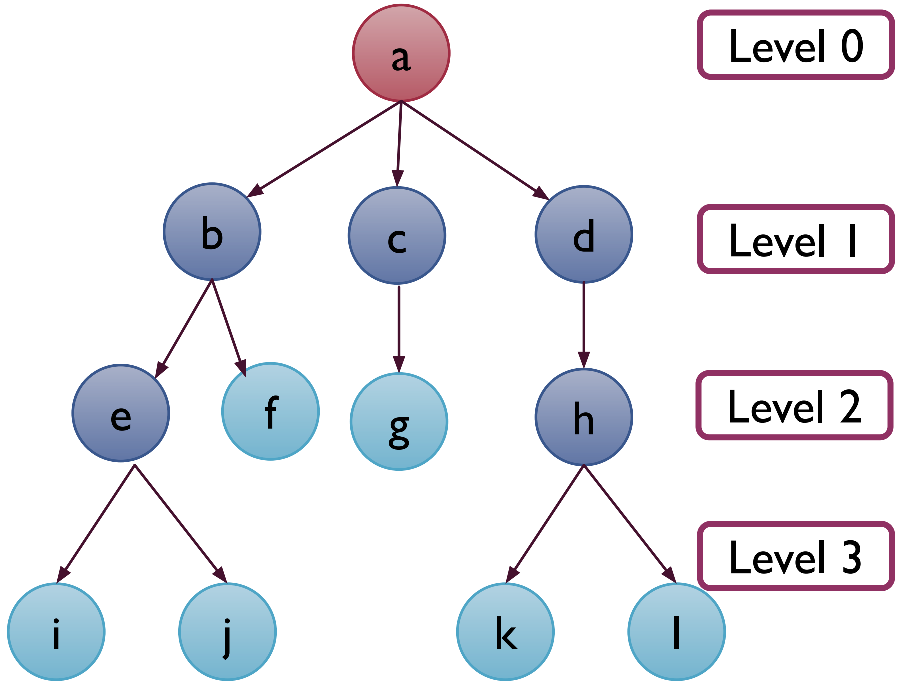
- Degree of Node
  - Number of its children
    - a’s degree is 3
    - b, h and e’s degree is 2
    - c’s degree is 1

- Degree of Tree
  - Maximum degree of any node
    - Since a has degree 3 that is maximum so degree of tree is 3



Tree Terminologies

- Depth/Level of Node
  - Number of branches on the path
  - j has depth 3
- Height of Tree
  - Number of nodes on the longest path from the root to a leaf
  - Maximum level OR Maximum depth →4
  - Number of nodes on the Longest path from root to any leaf node →4



Hierarchical Data And Trees

- The element at the top of the hierarchy is the root
  - Elements next in the hierarchy are the children of the root
  - Elements next in the hierarchy are the grandchildren of the root, and so on
  - Elements at the lowest level of the hierarchy are the leaves

Applications

- Expression evaluations
  - note how different traversals result in different notation

- Storing and retrieving information by a key

- Representing structured objects
  - e.g. the universe in an adventure game

- Useful when needing to make a decision on how to proceed
  - tic-tac-toe, chess

Tree ADT

Tree ADT

A simple tree T provides following basic operations

- Tree Methods
  - size(root)
    - returns total number of nodes
  - isEmpty(root)
    - if tree is empty or not
  - root()
    - returns root node of tree

Tree ADT

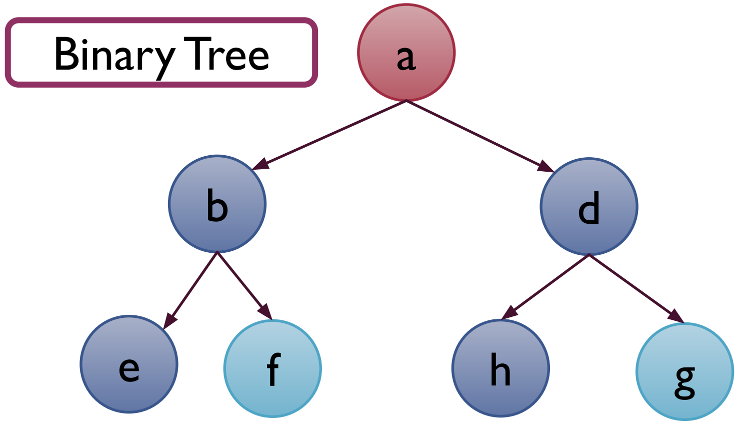
A simple tree T provides following basic operations

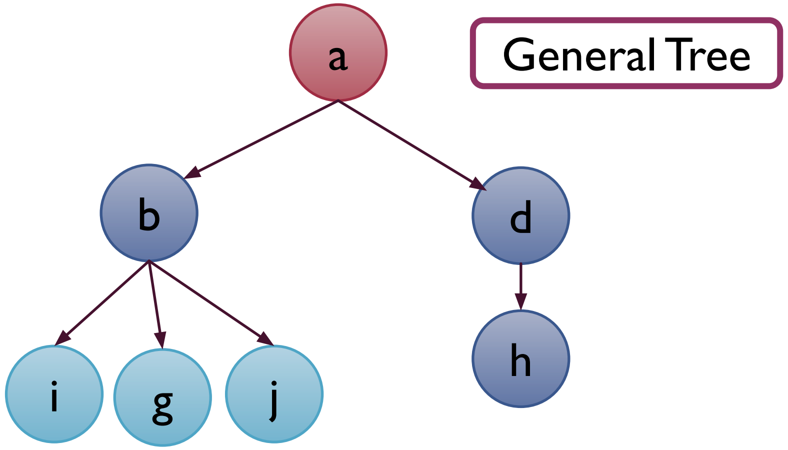
- Node Methods
  - parent(node)
    - returns parent of node
  - child(node)
    - returns list of all child’s of node
  - isInternal(node)
    - if node is non-leaf
  - isExternal(node)
    - if node is leaf
  - isRoot(node)
    - if node is root

Binary Trees

Binary Trees

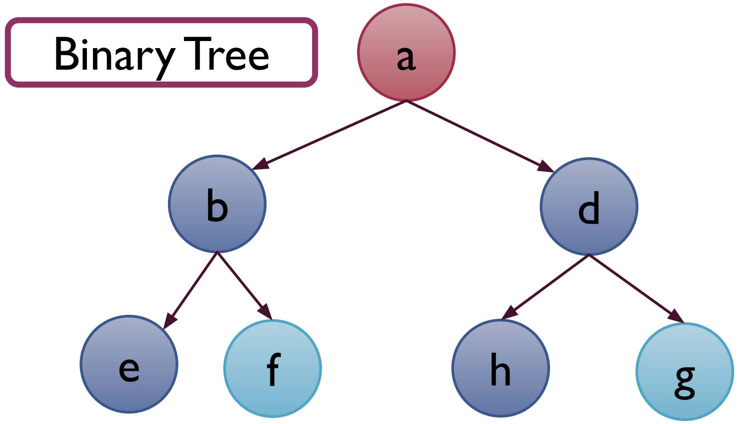
- A special tree where each node can have maximum two children.
  - Each node has a left child and a right child.
  - Even if a node has only one child, other child is still mentioned with NULL.

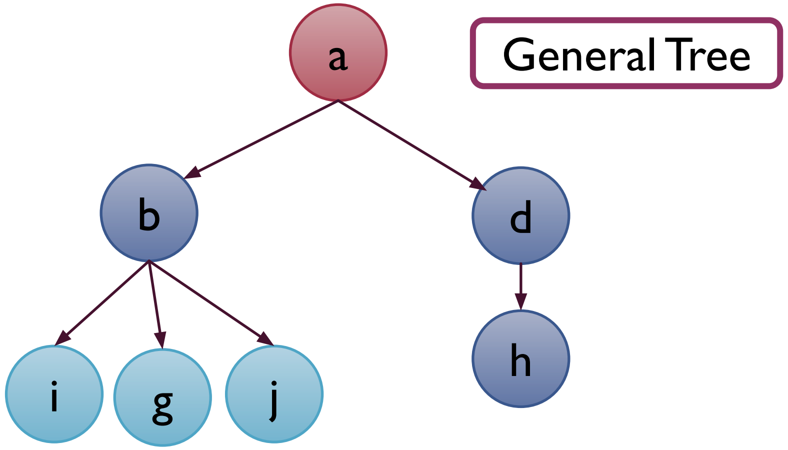




Binary Trees

- A non-empty binary tree has a root element
  - The remaining elements, if any, are partitioned into two binary trees
  - These are called the left and right subtrees of the binary tree

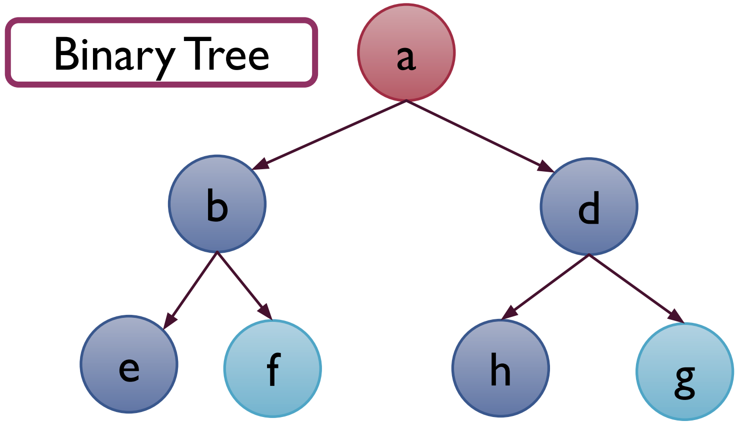


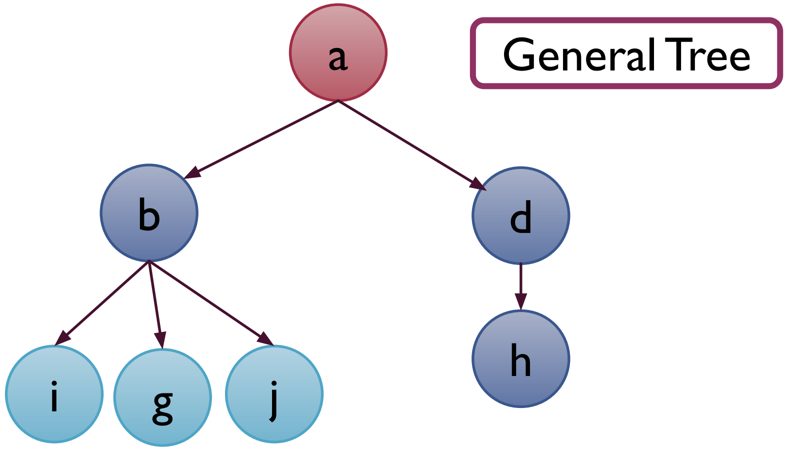


Tree vs. Binary Trees

- No node in a binary tree may have a degree more than 2
- whereas there is no limit on the degree of a node in a tree.

- A binary tree may be empty
- The subtrees of a binary tree are ordered
  - those of a tree are not ordered.





Tree vs. Binary Trees

- The subtrees of a binary tree are ordered
- those of a tree are not ordered

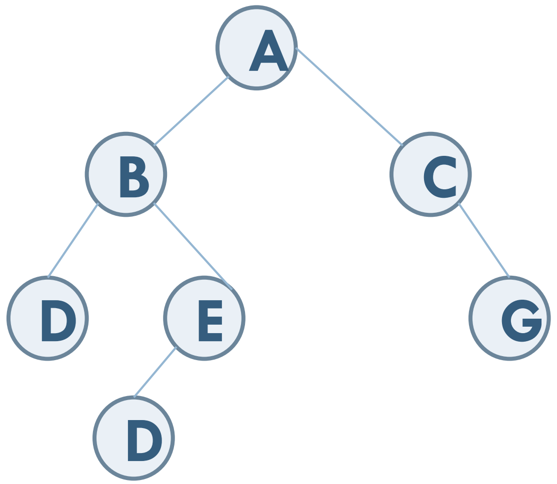
- Are different when viewed as binary trees
- Are the same when viewed as trees

Binary Trees

- Informal defn
  - each node has 0, 1, or 2 children

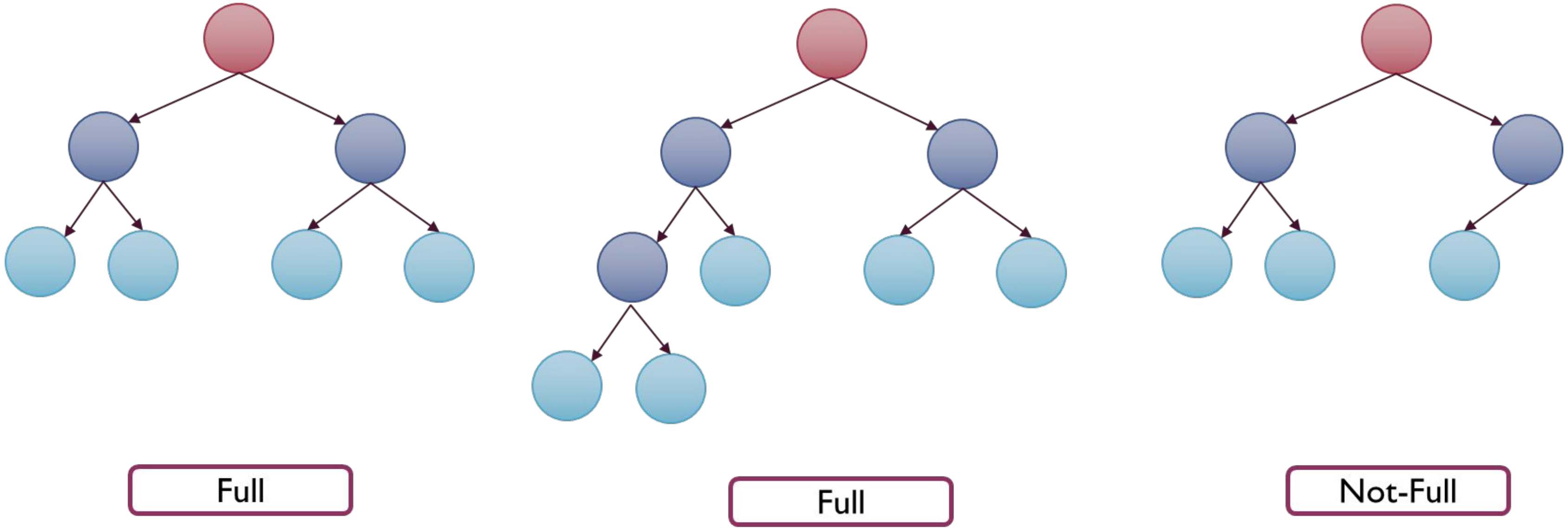
- Formal defn
  - a binary tree is a structure that
  - contains no nodes, or
  - is comprised of three disjoint sets of nodes
    - a root
    - a binary tree called its left subtree
    - a binary tree called its right subtree

  - A binary tree that contains no nodes is called empty
- Note: the position of a child matters!



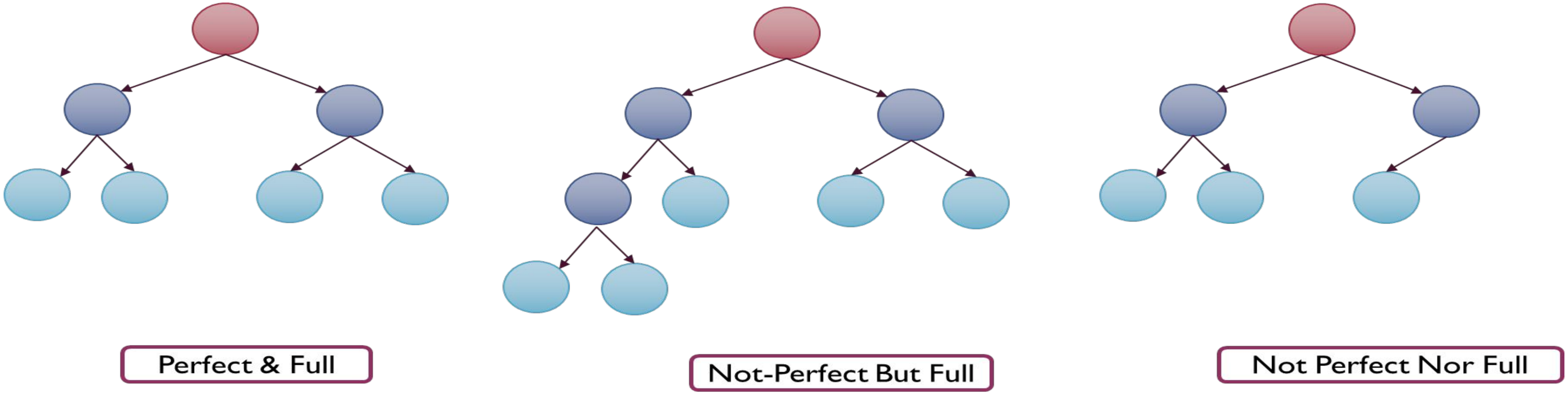
Binary Trees

- Full Binary Tree
  - Every node except leaf nodes has its maximum number of children.
  - It is a tree in which every node in the tree has either 0 or 2 children.



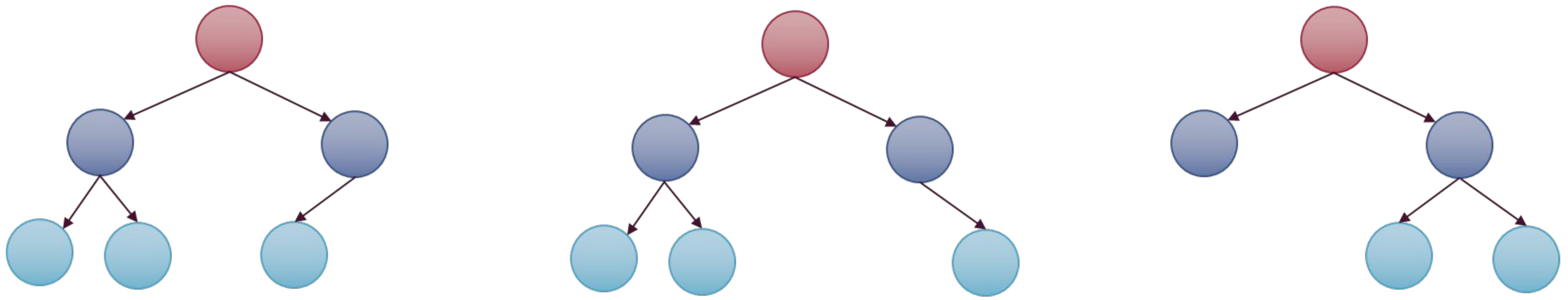
Binary Trees

- Perfect Binary Tree
  - A Full binary tree in which each leaf node has same depth/level.



Binary Trees

- Complete Binary tree
  - A tree that is completely filled at all levels, except the last level which is filled from left to right



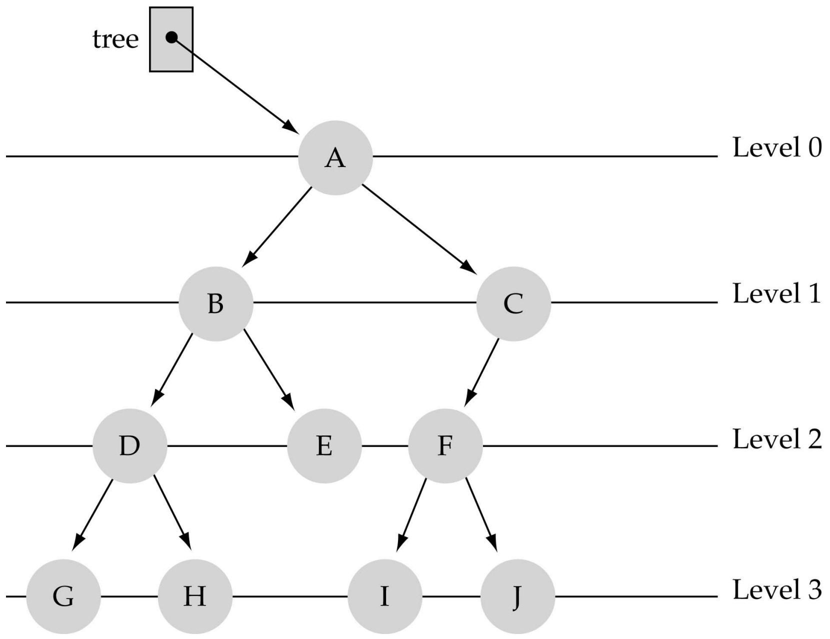
Complete But Not Full

Not-Complete

Not-Complete But Full

What is a binary tree?

- Property1: each node can have up to two **successor nodes** (children)
  - The predecessor node of a node is called its parent
  - The "beginning" node is called the root (no parent)
  - A node without children is called a leaf



$<= 2^0$

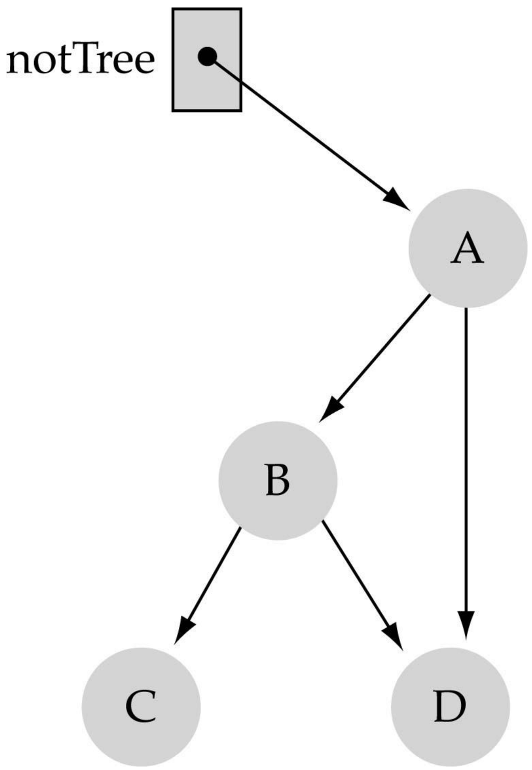
$\leq 2^1$

$\leq 2^2$

$\leq 2^3$

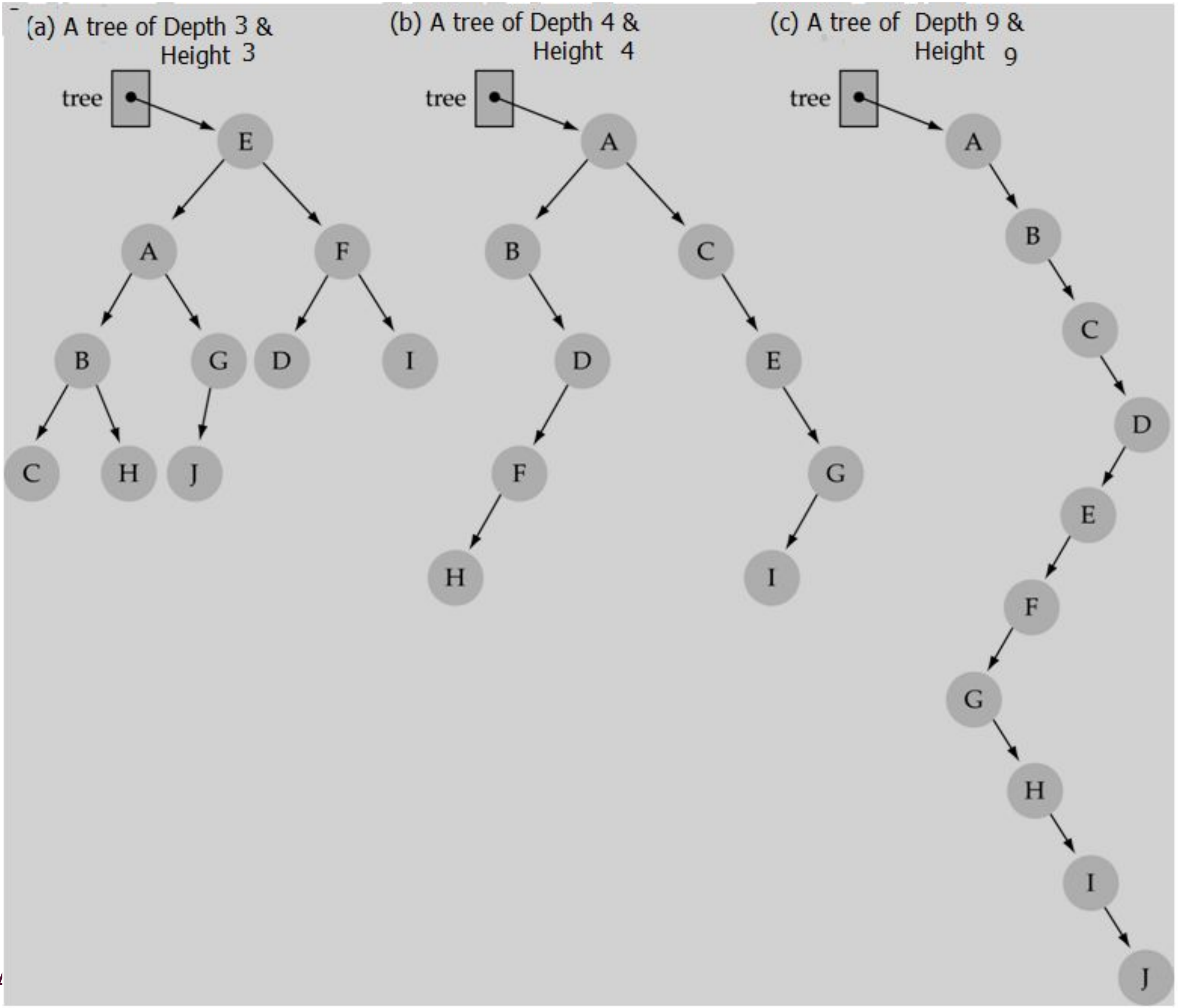
What is a binary tree?

- Property2: a unique path exists from the root to every other node
  - Below given example is not binary tree due to multiple paths from A to D



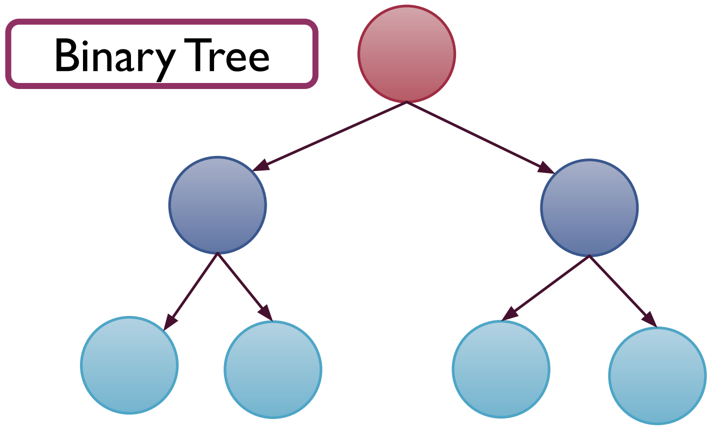
Binary Tree Terminologies

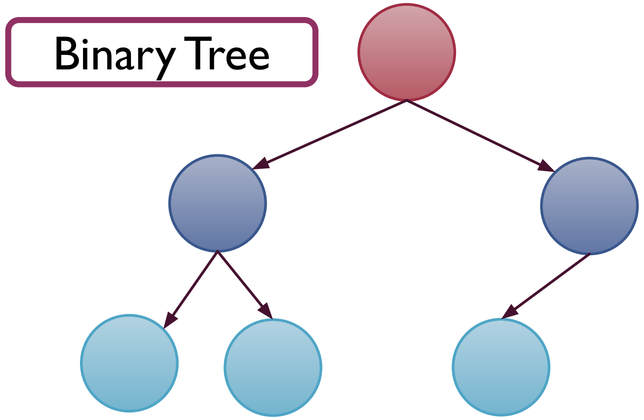
- Ancestor of a node
  - any node on the path from the root to that node
- Descendant of a node
  - any node on a path from the node to the last node in the path
- Depth of a node
  - number of edges in the path from the root to that node.
    - Remember, the depth of root node is 0; the depth of root's child nodes is 1 and so on
- Height of a node x
  - In the subtree of tree rooted at node x, find the leaf node y which is farthest from x;
  - start counting nodes upwards from y to x.
  - Height of x is the final count.

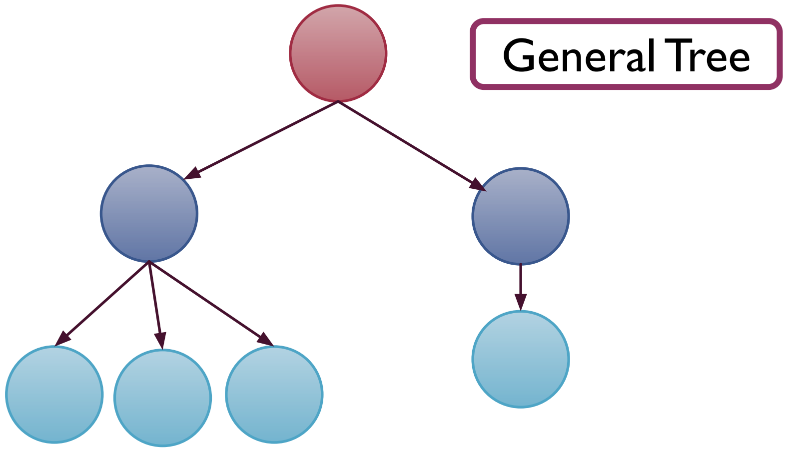

(a) A tree of Depth 3 & Height 3
tree
E
A
F
B
G
D
I
C
H
J
(b) A tree of Depth 4 & Height 4
tree
A
B
C
D
E
F
G
H
I
(c) A tree of Depth 9 & Height 9
tree
A
B
C
D
E
F
G
H
I
J

Binary Trees

- Recursive Definition
  - T is a binary tree if
  - T is empty (NULL) OR
  - T's root node has maximum two children's, where each child is itself a binary tree.
    - Left child is called left subtree and right child is called right subtree

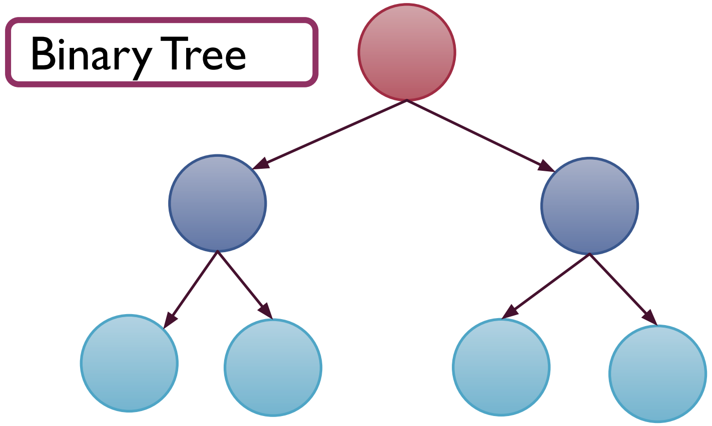






Mathematical Properties of Binary Trees

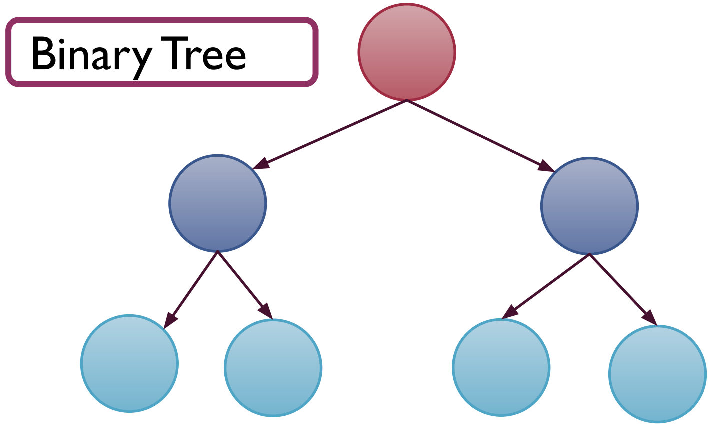
- Let's us look at some important mathematical properties of binary trees
  - A good understanding of these properties will help the understanding of the performance of algorithms that process trees



- Some of the properties relate to the structural properties of these trees.
  - This is the case because performance characteristics of many algorithms depend on these structural properties and not only the number of nodes

Mathematical Properties of Binary Trees

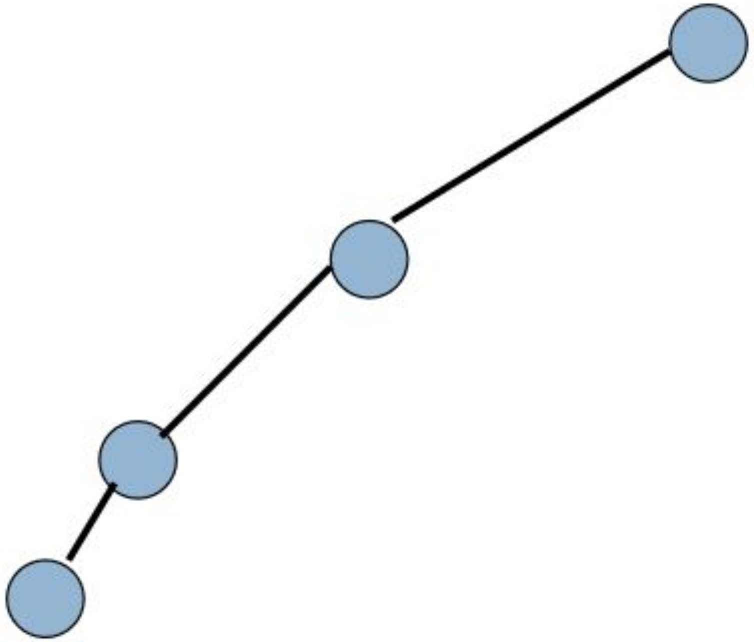
- Maximum nodes at level i of binary tree?
  - 2^i



Minimum Number Of Nodes

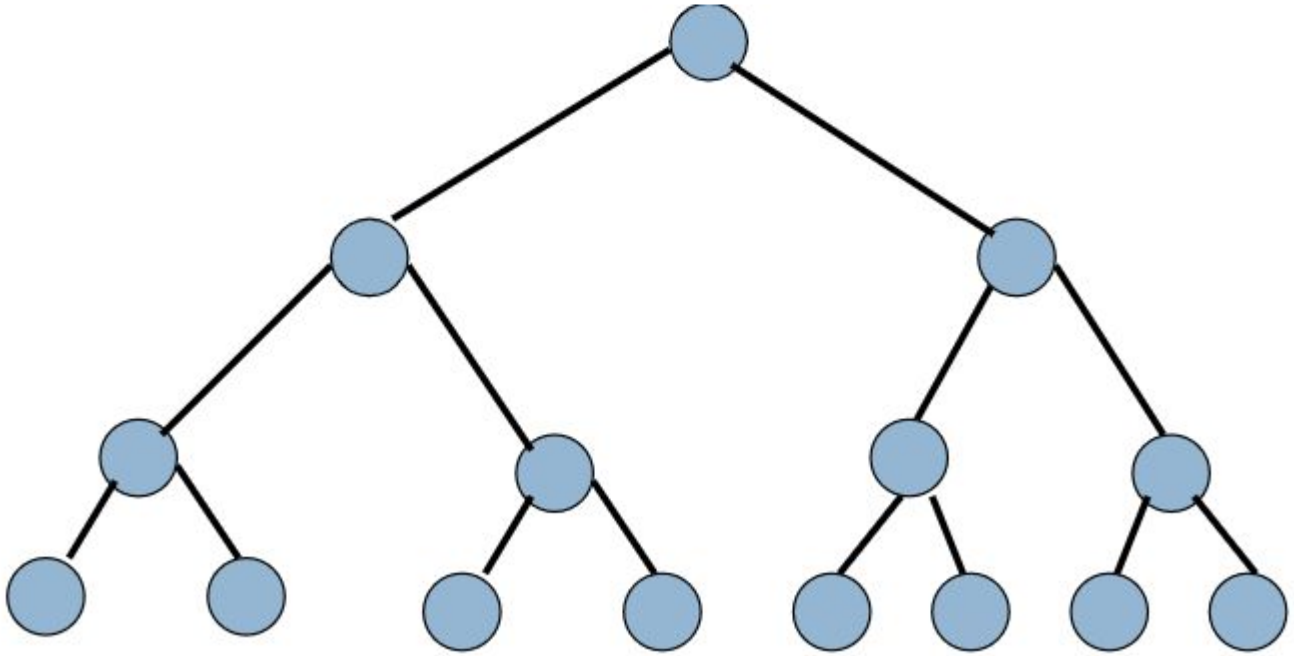
- If binary tree has height h, minimum number of nodes is h
  - in case of left skewed and right skewed binary tree

- minimum number of nodes is h
  - N=h



Maximum Number Of Nodes

- All possible nodes at first d depths are present
  - Maximum number of nodes
  - $= 1 + 2 + 4 + 8 + \ldots + 2^d = 2^h - 1$



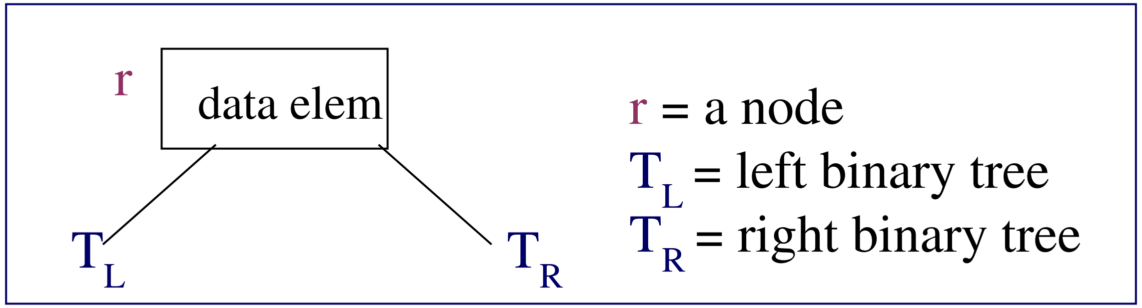
Number Of Nodes & Height

- Let n be the number of nodes in a binary tree whose height is h.
- (h) <= n <= 2h – 1
- ⌊ log2(n) ⌋ <= h <= n
- The max height of a tree with N nodes is N
- The min height of a tree with N nodes is ⌊log2(N)⌋
-
-

Binary Tree ADT

Binary Tree

- Binary Tree ADT is a finite set of nodes which is
  - either empty
  - or consists of root and two disjoint binary trees
    - left and right subtrees of the root
- Nodes with no successors are called leaves.
- A child has one parent.
- A parent has at most two children.



Why Binary Trees

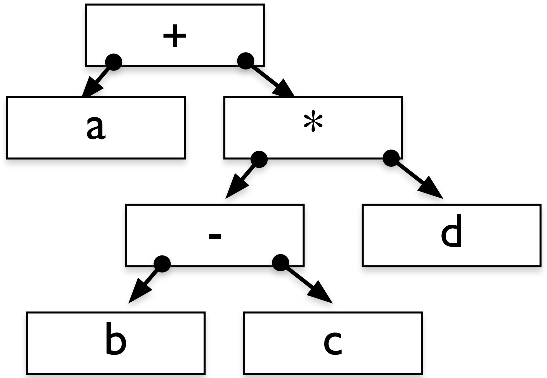
- Every Data structure has its own efficiency and limitations
  - Arrays
    - Good for element access
  - Linked Lists
    - Good for lot of insertions and deletions
  - Stacks
    - Where order is much important like mathematical expression evaluation
  - Queue
    - Again order matters, process scheduling on CPU

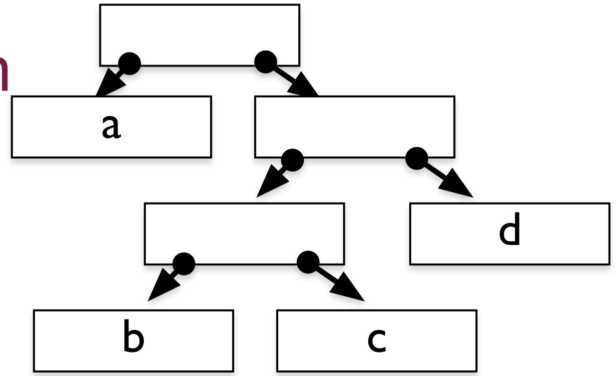
Why Binary Trees

- Tree
  - good where data has hierarchical relationship, like file system, parsers, sorting
  - One common issue with all linear data structures is in-efficient searching
    - takes O(n) time
  - Binary Trees can improve this time to an average case of O(log n)
  - Even though worst case can still leads to O(n) time but this can be improved using certain types of Trees.
- There are some variations of Binary Tree which guarantees O(log n) time for insertion, deletion and searching
  - Binary Search Tree (BST),
  - AVL Trees,
  - Heaps
  - etc

Applications of Binary Tree

- Binary Tree has different variations which are used for different purposes
  - Parse Tree
    - Expression trees (e.g., in compiler design) for checking that the expressions are well formed and for evaluating them.
      - leaf nodes are operands (e.g. constants)
      - non leaf nodes are operators
  - Huffman coding trees, for implementing data compression algorithms:
    - each leaf is a symbol in a given alphabet
    - code of the symbol constructed following the path from root to the leaf (left link is 0 and right link is 1)
  - Binary Search Tree (BST) →Ordered Tree
  - Binary Heap





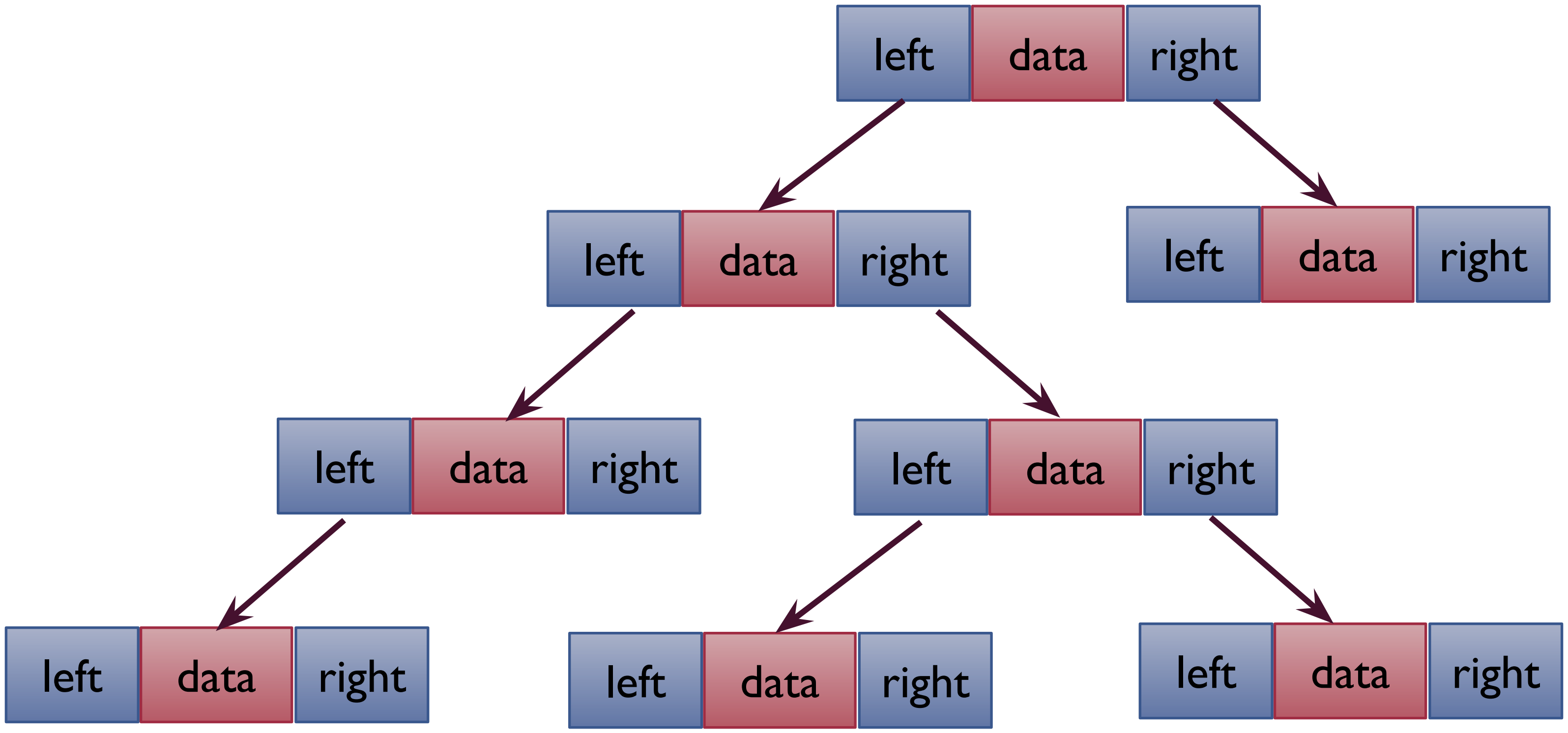
a = 0, b = 100, c = 101, d = 11

Linked-based Representation

Each node has two links left and right

- If root node is null, means tree is empty
- If node's left, right links are NULL, it means its leaf node
- Optionally, a parent field with a pointer to the parent node

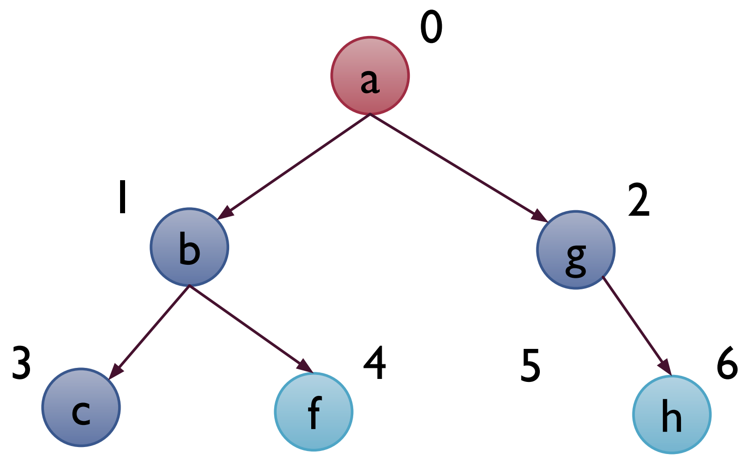
```
Struct Node{
    Data;
    Node*left;
    Node*right;
    };
```



Array-based Representation

- A fixed size tree can be represented using arrays.
- If we know the height of tree,
  - we can define size of array to hold maximum possible number of nodes → $2^{h+1}-1$
  - Root of tree → array[0]
  - Left child of root →array[1]
  - Right child of root →array[2]

  - Left child of node at index k →array[2k+1]
  - Right child of node at index k→array[2k+2]



0	1	2	3	4	5	6
a	b	g	c	f	NULL	h

Tree Traversal

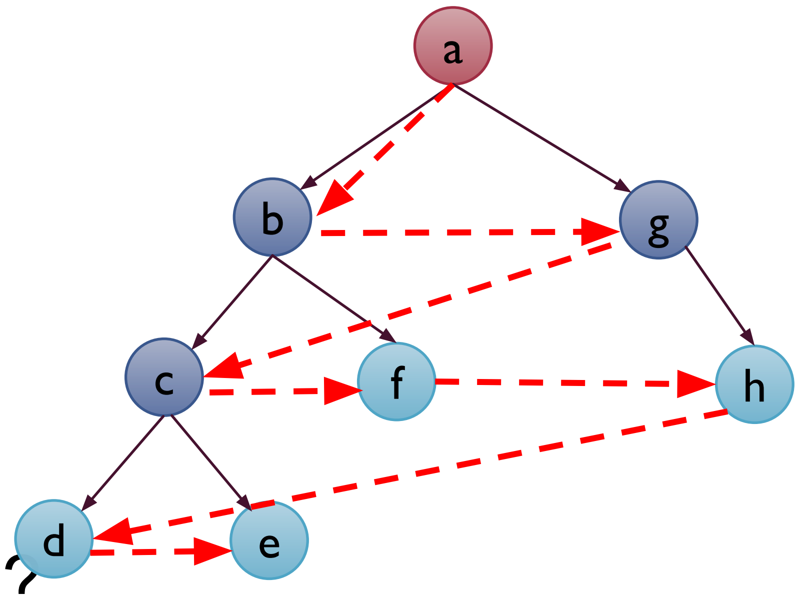
- A tree traversal means visiting each node of tree once.
- Due to non-linear structure of tree there is not a single way to traverse node:
  - Breadth First Search
  - Depth First Search
    - Pre-Order
    - In-Order
    - Post-Order

Breadth First Search (BFS)

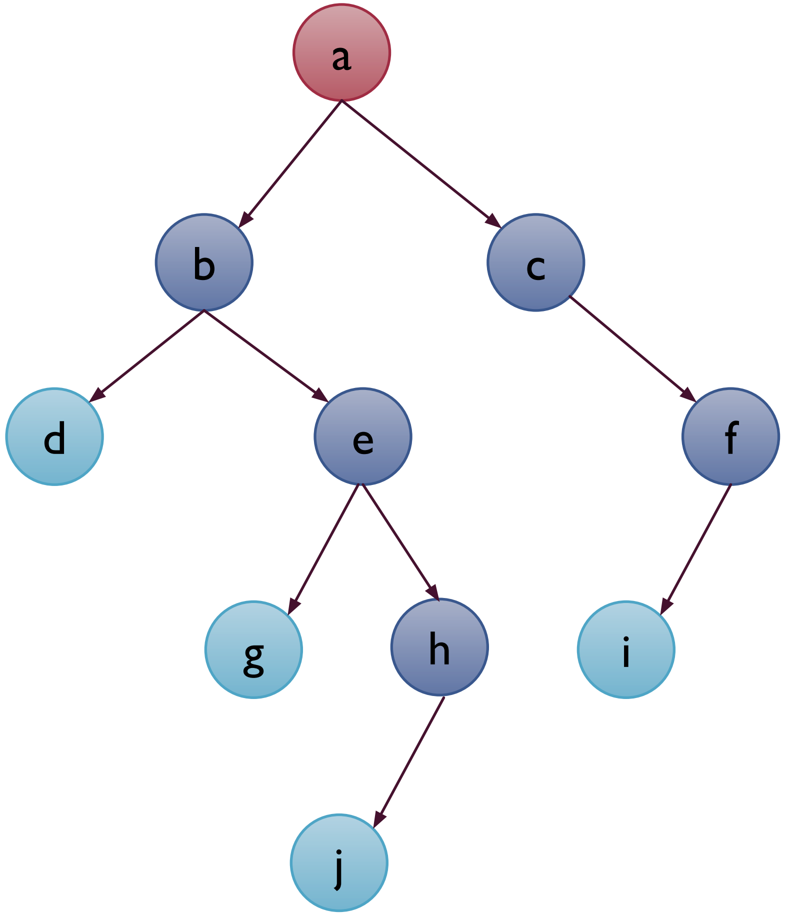
- Starting from root node, visit all of its children, all of its grandchildren and all of its great grandchildren
  - Order of nodes: a b g c f h d e
  - Nodes at same level must be visited first before nodes of next level

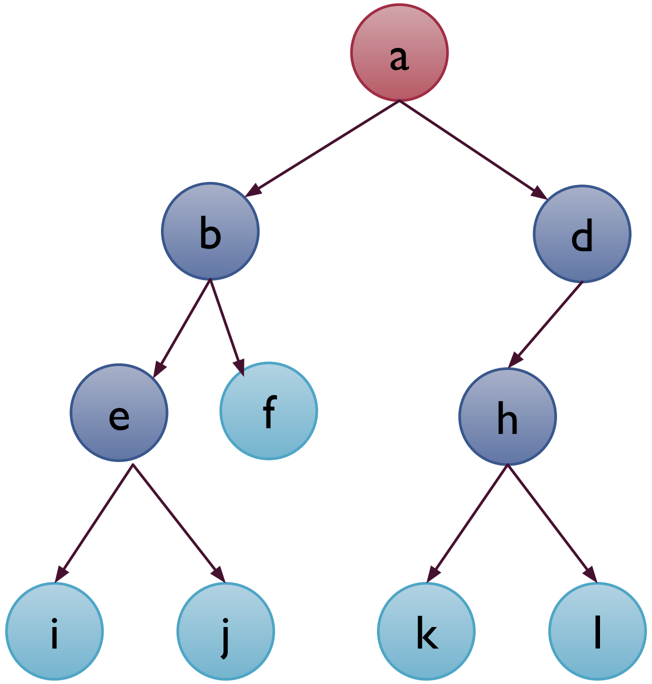
- Also known as level order traversal
- Implementation?
  - We should store nodes to keep track of them.
  - The sequence in which we store them effects the sequence in which we retrieve them back

- Which data structure can be used to store nodes?
  - array, stack or queue

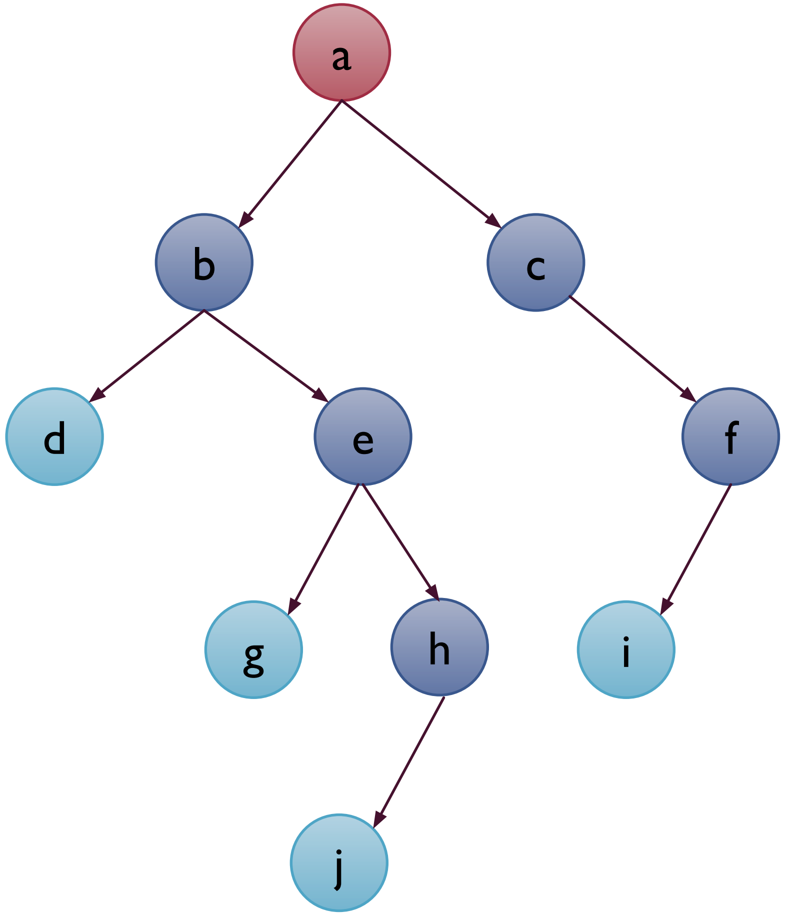


Breadth First Search (BFS)

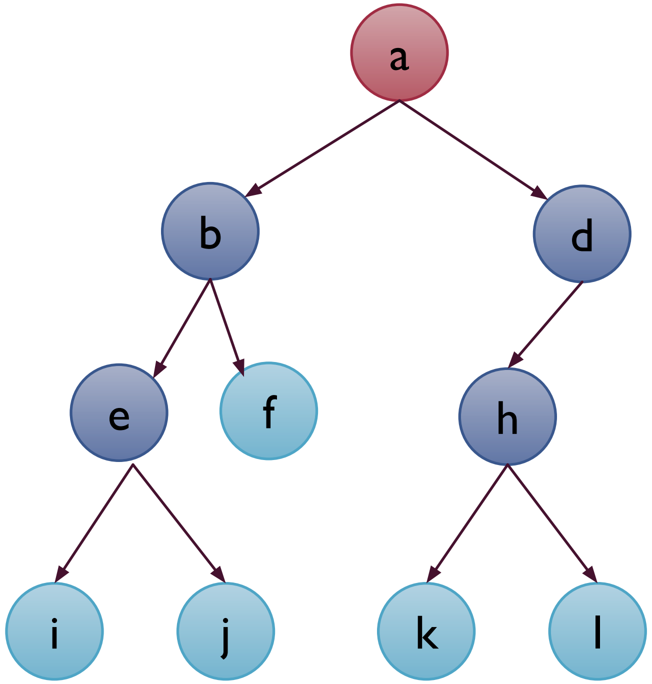




Breadth First Search (BFS)



a b c d e f g h i j

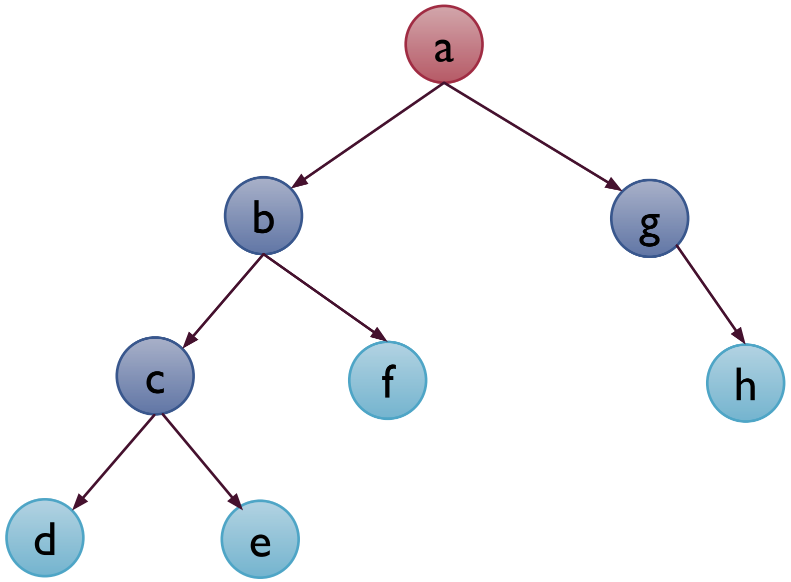


a b d e f h i j k l

Breadth First Search (BFS)

Algorithm: Iterative_BFS(node *root)

- Set current = root and Queue is empty initially
- if (current != NULL)
  - Enqueue (current)
  - while (Queue is not empty)
    - current = Dequeue()
    - visit current
    - if (current.left)   // if(IsLeft(current))
      - Enqueue (current.left)
    - if (current.right)   // if(IsRight(current))
    - Enqueue (current.right)
  - End While
- End if



a
b g
g c f
c f h
f h d e
h d e
d e
e

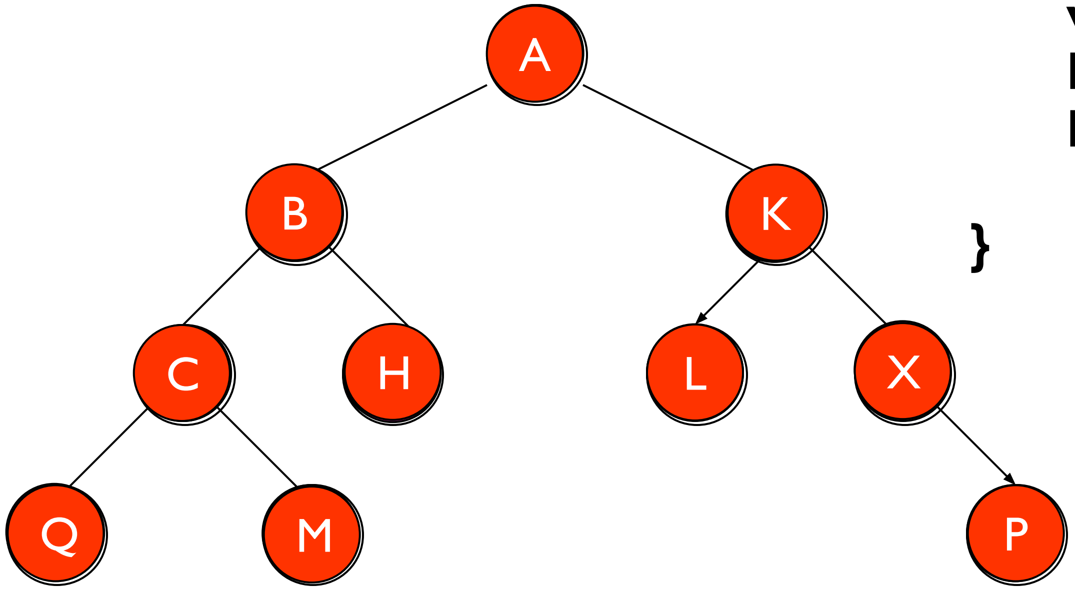
Output: a b g c f h d e

Breadth First Search (BFS)

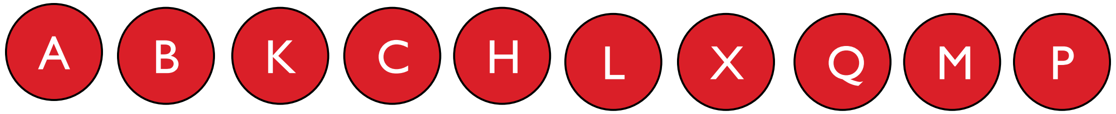
Recursive_BFS(Node *node)

- If (node != Null)
  - visit(node)
- If hasLeft(node)
  - enqueue(node->left)
- If hasRight(node)
  - enqueue(node->right)
- If Q is not Empty
  - Recursive_BFS(Dequeue())
- Else
  - return

Breadth First Search (BFS)

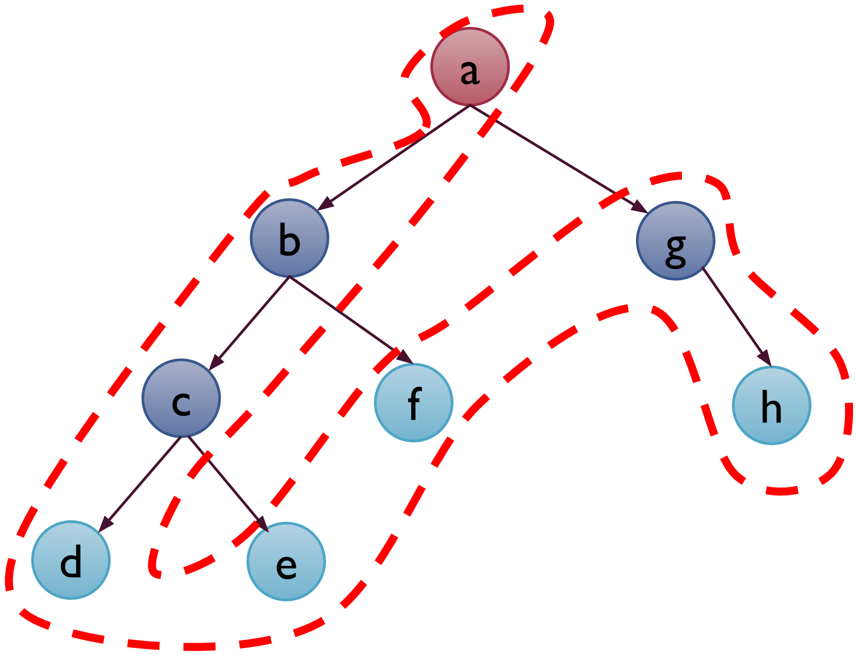



```
levelOrder (Queue queue,TreeNode * t)
{
      if(t == NULL) then return;
      visit t;
      Enqueue(t. left);
      Enqueue(t. right);
            levelOrder(queue, Dequeue());
}
```



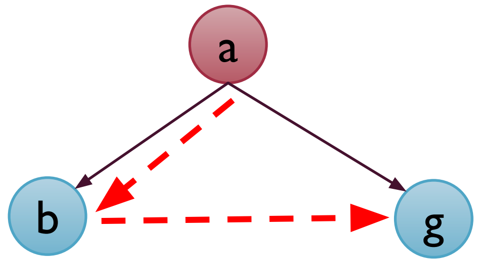
Depth First Search (DFS)

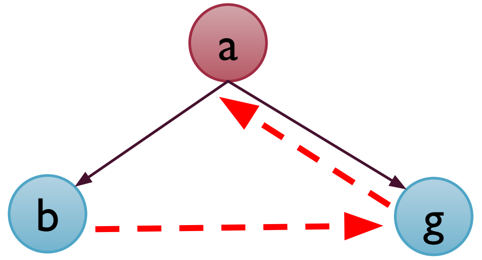
- Using the top-down view of the tree, starting from root, go to each subtree as far as possible, then back track
- Possible Orders
  - Left subtree and then right subtree
    - a b c d e f g h
  - right subtree and then left subtree
    - a g h b f c e d
- Implementation
  - Can we use a stack instead of queue

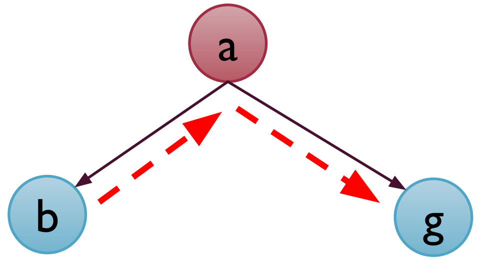


Depth First Search (DFS)

- Depth First Search can also be implemented with recursive approach.
- Depending upon the order in which we go in depth can bring different variations in order of node traversal.
  - Pre-Order (simple DFS)
    - Visit node
    - Visit left child of node
    - Visit right child of node
  - Post-Order
    - Visit left child of node
    - Visit right child of node
    - Visit node
  - In-Order
    - Visit left child of node
    - Visit node
    - Visit right child of node

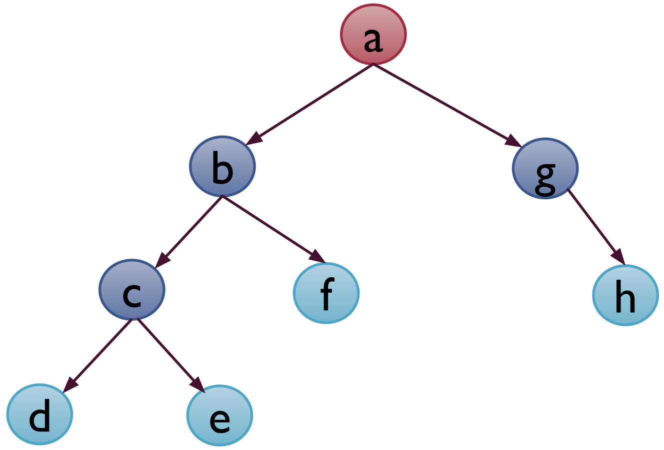






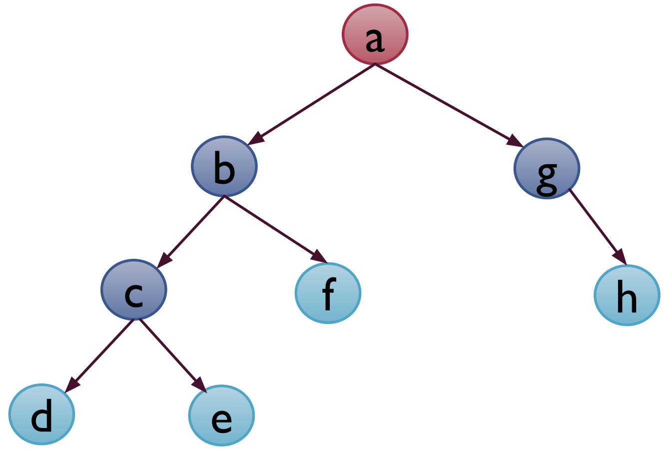
Inorder vs. Preorder vs. Post order

- Pre-order (node-left-right)
- Post-order (left-right-node)
- In-order (left-node-right)

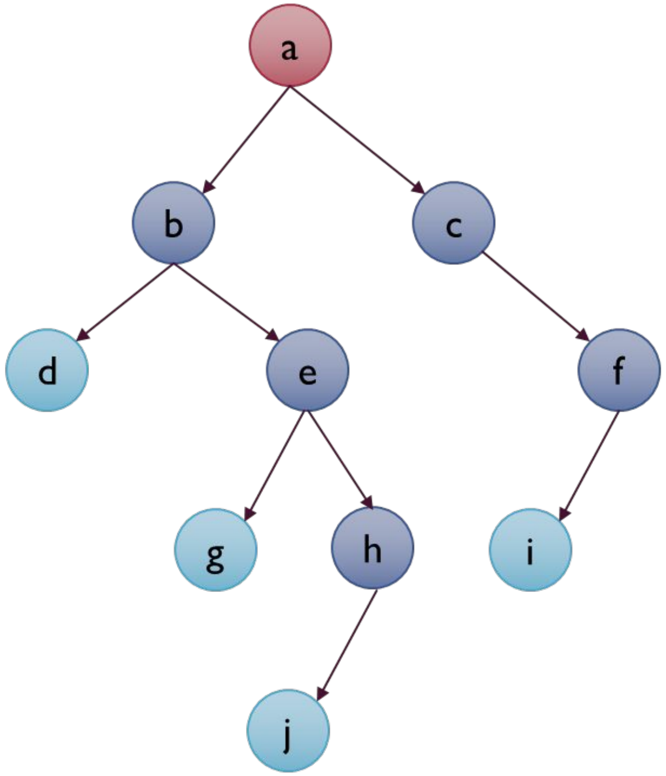


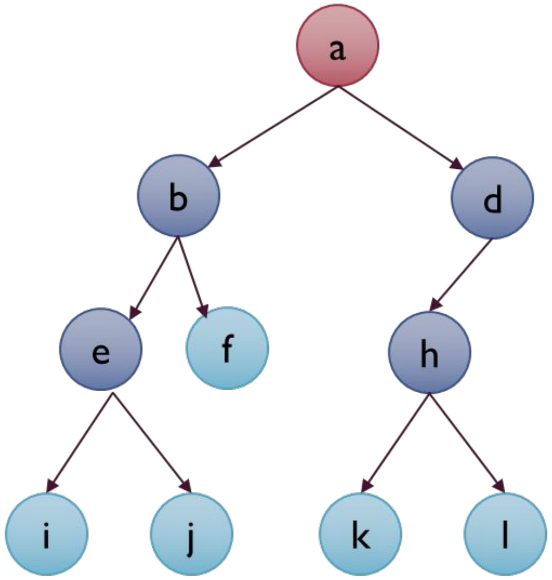
Inorder vs. Preorder vs. Postorder

- Pre-order (node-left-right)
  - a b c d e f g h
- Post-order (left-right-node)
  - d e c f b g h a
- In-order (left-node-right)
  - d c e b f a g h

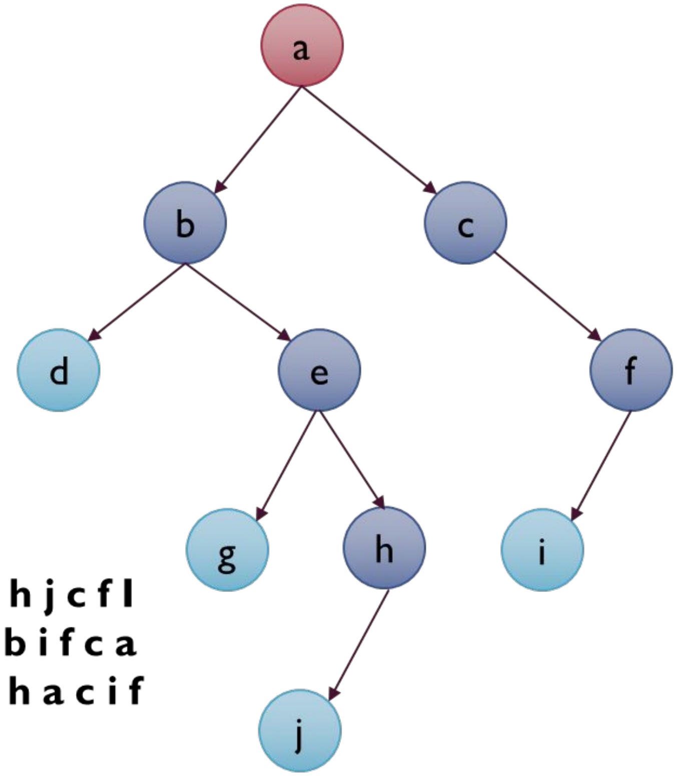


Inorder vs. Preorder vs. Postorder

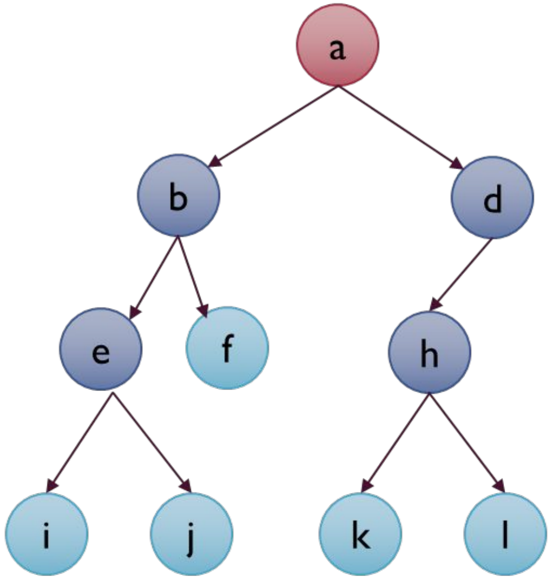




Inorder vs. Preorder vs. Postorder



Pre-Order: **a b d e g h j c f I**
Post-Order: **d g j h e b i f c a**
In-Order: **d b g e j h a c i f**



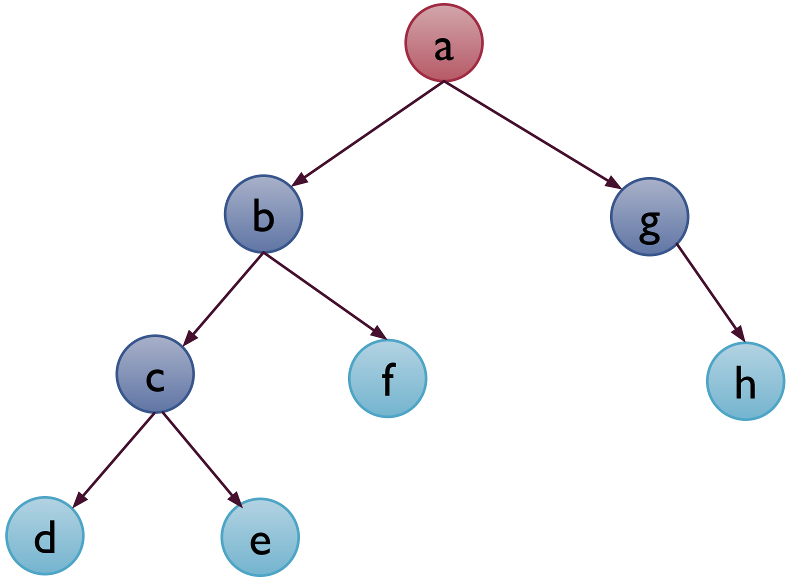
Pre-Order: **a b e i j f d h k l**
Post-Order: **i j e f b k l h d a**
In-Order: **i e j b f a k h l d**

Inorder Traversal

Algorithm: IterativeInOder(node * root)

Input: a pointer to root node of Tree.

- Set current = root and initially Stack is empty
- while(current ! =NULL || stack is nonempty)
  - if(current ! = NULL)
    - push (current)
    - current = current.left
  - else
    - current = pop ()
    - visit current
    - current = current.right
- End While



Output: d c e b f a g h

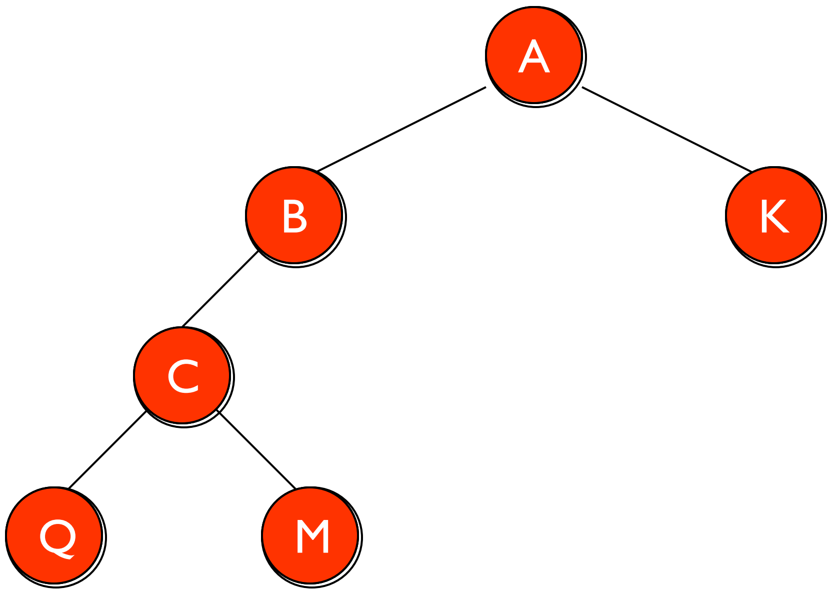
d								
c	c	e						
b	b	b	b		f			
a	a	a	a	a	a	a	g	h

Inorder Traversal

- Recursive_InOrder(Tree *node)
  - If node is not NULL
    - Recursive_InOrder(node.left)
    - visit(node)
    - Recursive_InOrder(node.right)
  - End If

Inorder Traversal (LNR)

```
if (t) {
    InOrder(t->left);
    visit(t);
    InOrder(t->right);
}
```



Q C M B A K

InOrder(NULL)

InOrder(M)

InOrder(M->left)
visit(M)
InOrder(M->right)

t=M

InOrder(NULL)

InOrder(C->left)
visit(C)
InOrder(C->right)

t=C

InOrder(K)

InOrder(K->left)
visit(K)
InOrder(K->right)

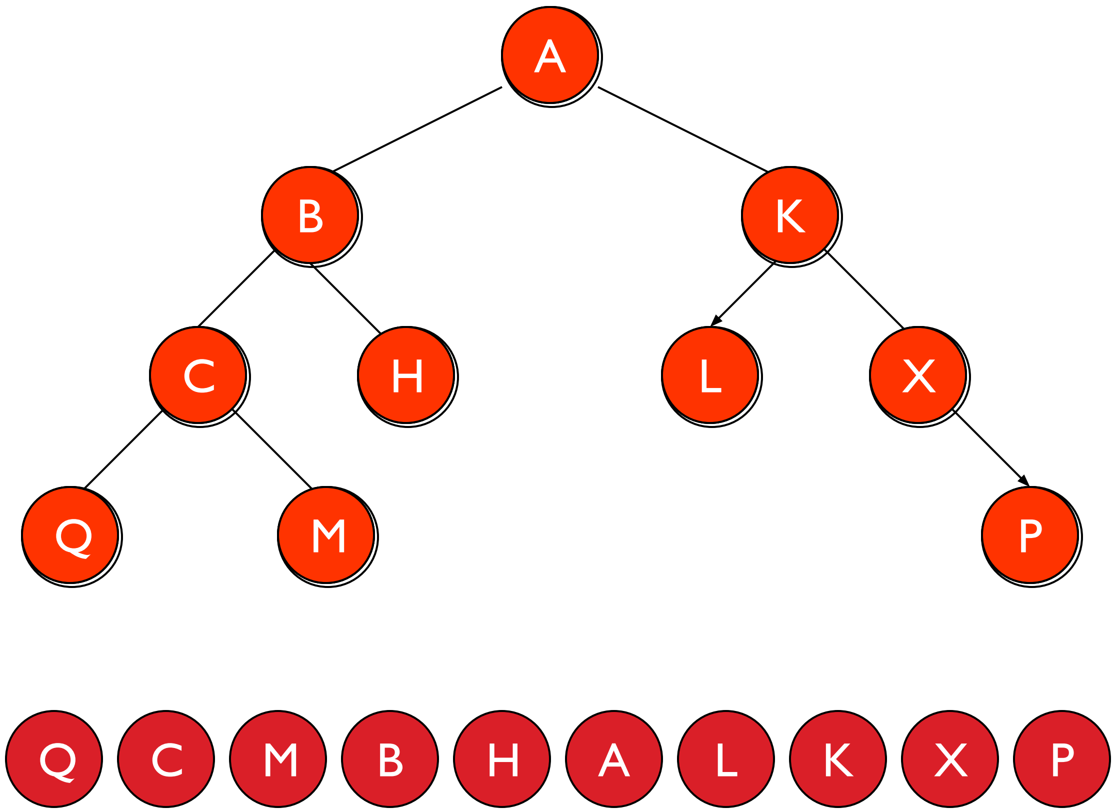
t=K

InOrder(A)

InOrder(A->left)
visit(A)
InOrder(A->right)

t=A

Inorder Traversal (LNR)



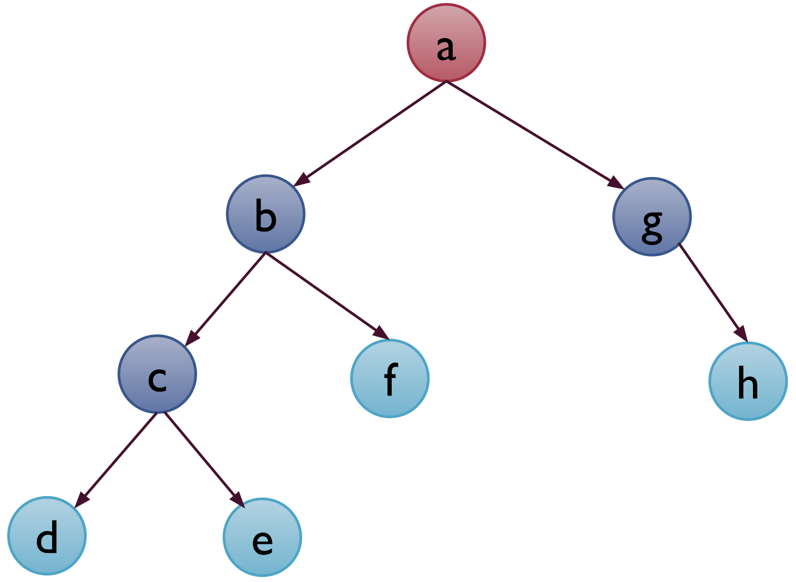

```
InOrder(TreeNode *t)
{
    if (t == Null ) then return;
    InOrder(t.left);
    visit(t);
    InOrder(t.right);

}
```

Preorder Traversal

Algorithm: IterativePreOder(node * root)

- Set current = root and initially stack is empty
- while(current ! = NULL || stack is nonempty)
  - if(current ! = NULL)
    - visit current
    - if (current.right ! = NULL)
      - push(current.right)
    - current = current.left
  - else
    - current = pop()
- End while



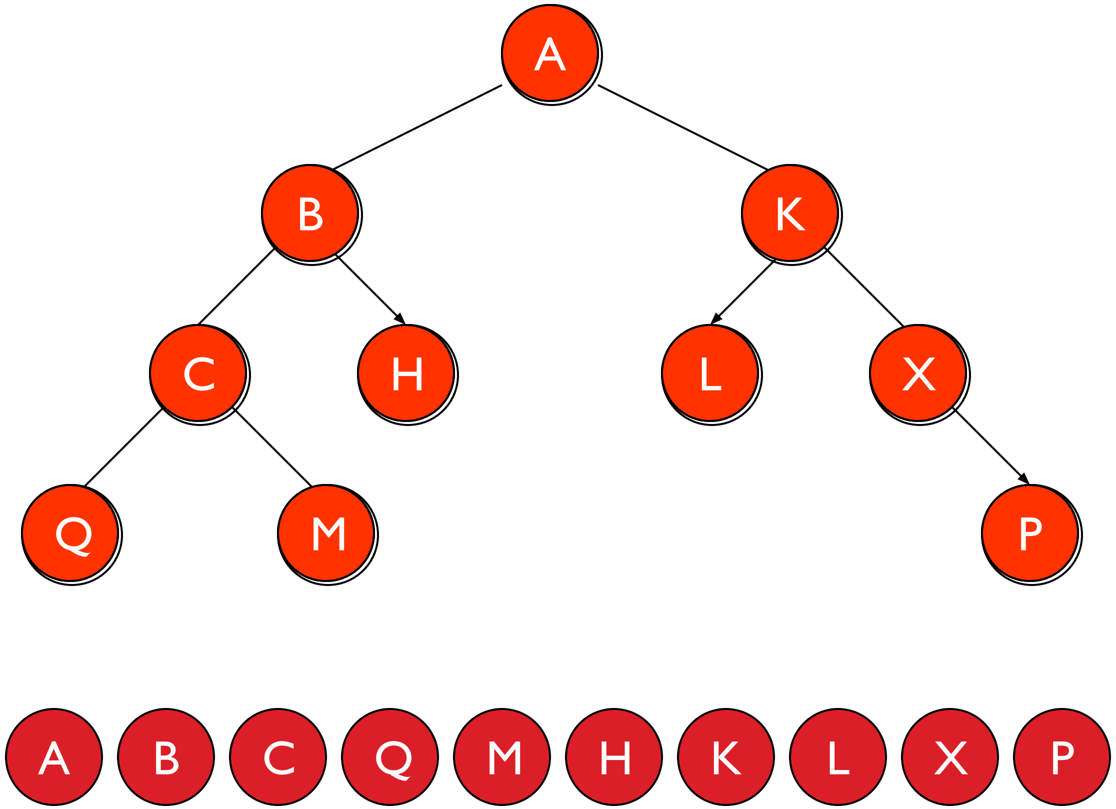
Output: a b c d e f g h

		e			
	f	f	f		
g	g	g	g	g	h

Preorder Traversal

- Recursive_PreOrder(Tree *node)
  - If node is not NULL
    - visit(node)
    - Recursive_PreOrder(node.left)
    - Recursive_PreOrder(node.right)
  - End If

Preorder Traversal (NLR)



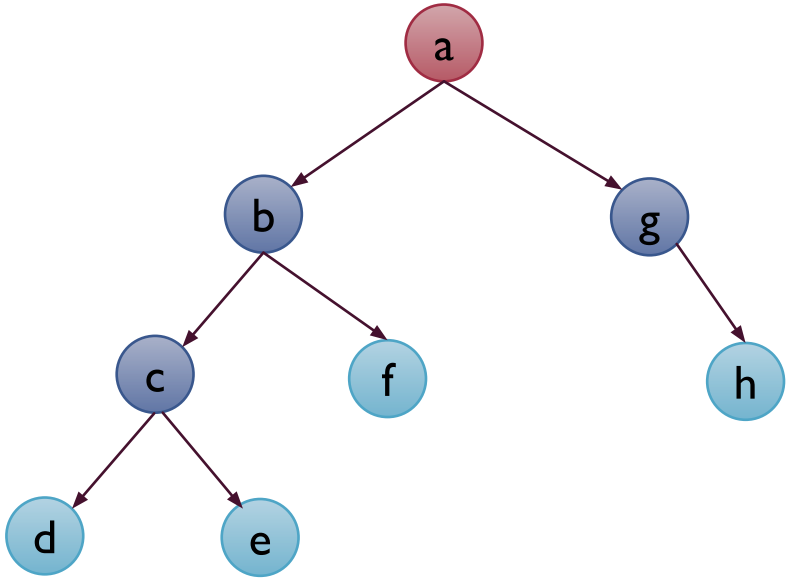

```
PreOrder(TreeNode *t)
{
        if (t == NULL) then return;
        visit(t);
        PreOrder(t.left);
        PreOrder(t.right);

}
```

PostOrder Traversal

Algorithm: IterativePostOder(node * root)

- Set current = root lastNodeVisited = NULL topNode = Null
- while (current ! = NULL || Stack is not empty)
  - if (current ! = NULL)
    - push(current)
    - current = current.left
  - else
    - topNode = top()
    - if (topNode.Right ! = NULL && lastNodeVisited ! = topNode.right)
      - current = topNode.Right
    - else
      - visit topNode
      - lastNodeVisited = pop()
- End While



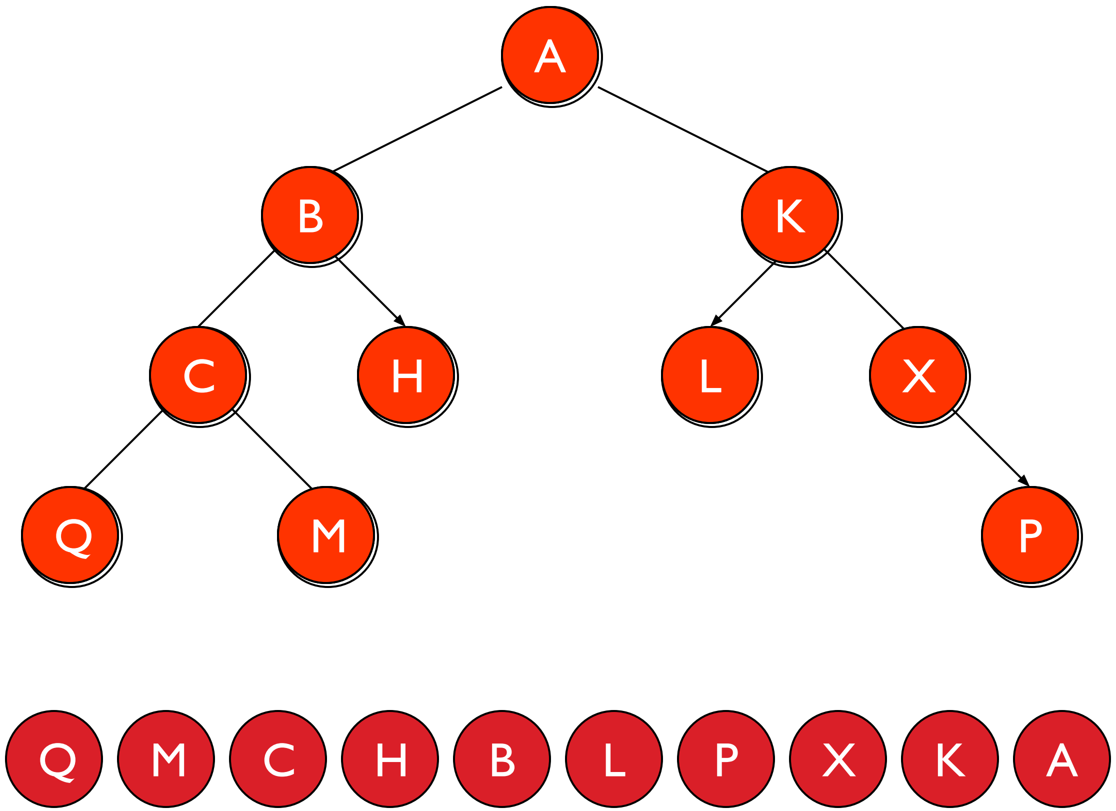
Output: **d e c f b h g a**

d	d		e								
c	c	c	c	c		f				h	
b	b	b	b	b	b	b	b		g	g	g
a	a	a	a	a	a	a	a	a	a	a	a

PostOrder Traversal

- Recursive_PostOrder(Tree *node)
  - If node is not NULL
    - Recursive_PostOrder(node->left)
    - Recursive_PostOrder(node->right)
    - visit(node)
  - End If

Postorder Traversal (LRN)



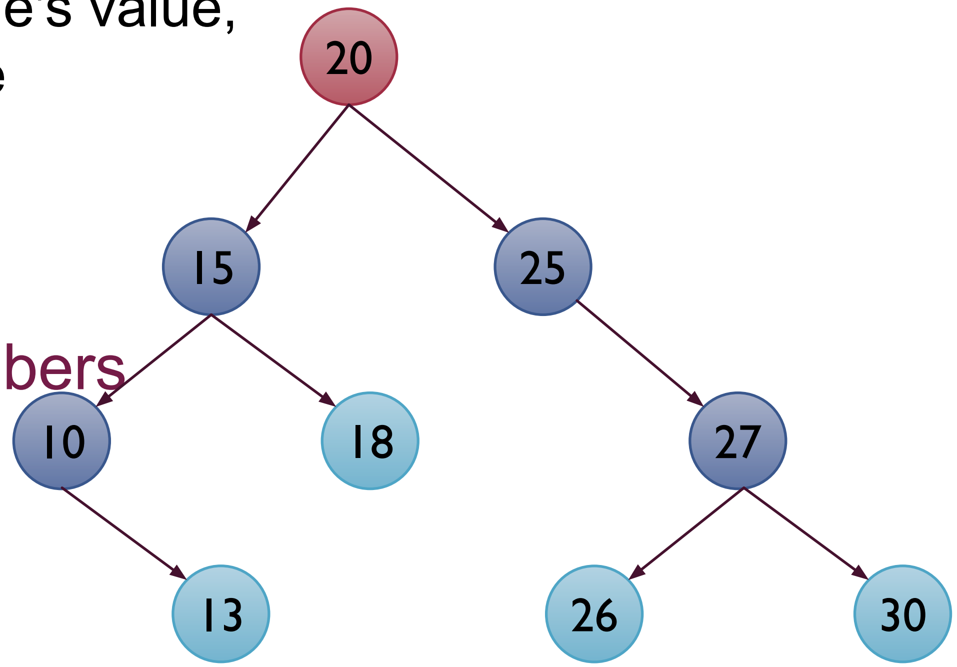

```
PostOrder(TreeNode *t)
{
        if (t == NULL) then return;
        PostOrder(t.left);
        PostOrder(t.right);
        visit(t);

}
```

Binary Search Tree (BST)

BST

- A Binary Search Tree is a binary tree with a special property
  - For each node in tree
    - Node’s left subtree holds values less than the node's value,
    - Node’s right subtree holds values greater than the node's value.



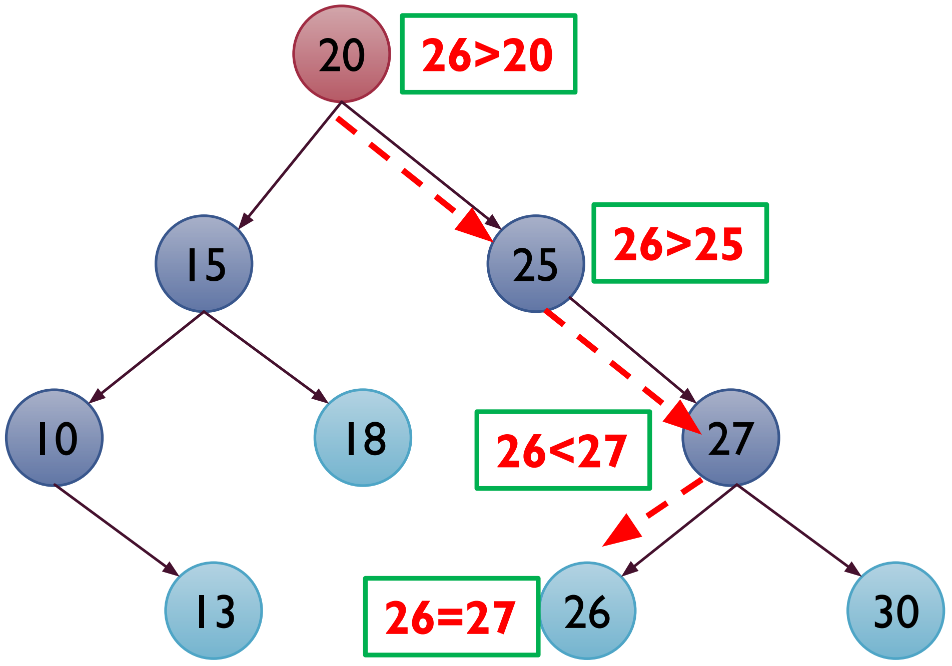
- BST are used to present sorted data
  - If you traverse tree in order, it will produce numbers
    - 10 13 15 18 20 25 26 27 30
- Operations
  - Search
  - Insertion
  - Deletion

Searching BST

BST_SEARCH(node, value)

- Input: root node of tree, value to be searched
- Output: node which contains value
- If node!= null
    - If value==node.data
        - return node
    - If value < node.data
        - return BST_SEARCH(node.left, value)
    - If value > node.data
        - return BST_SEARCH(node.right, value)
- End If

Let say we want to search 26
How much time will it take?
What would be the worst case scenario?

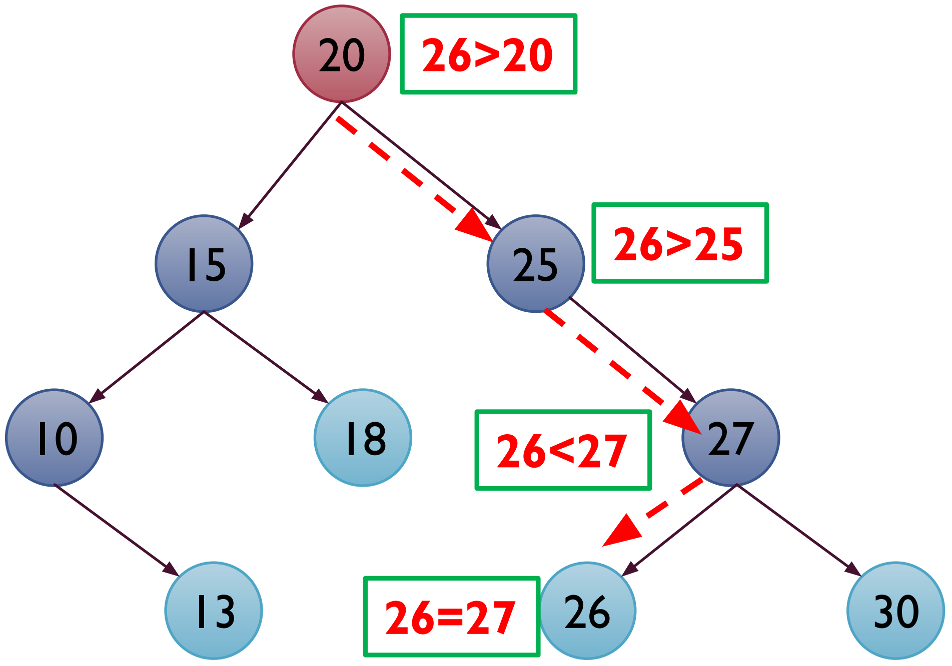


Searching BST

BST_Search_iterative(key, root):

- node = root
- while (node !=Null)
  - if (key == node.data)
    - return node
  - else if (key < node.data)
    - node = node.left
  - else   // key > node.data
    - node = node.right
- End while
- return Null

Let say we want to search 26
How much time will it take?
What would be the worst case scenario?



Sorting BST

- BST is already sorted.

- We only have to select our traversing strategy.
  - In-order traversal of a binary search tree always gives a sorted sequence of the values.

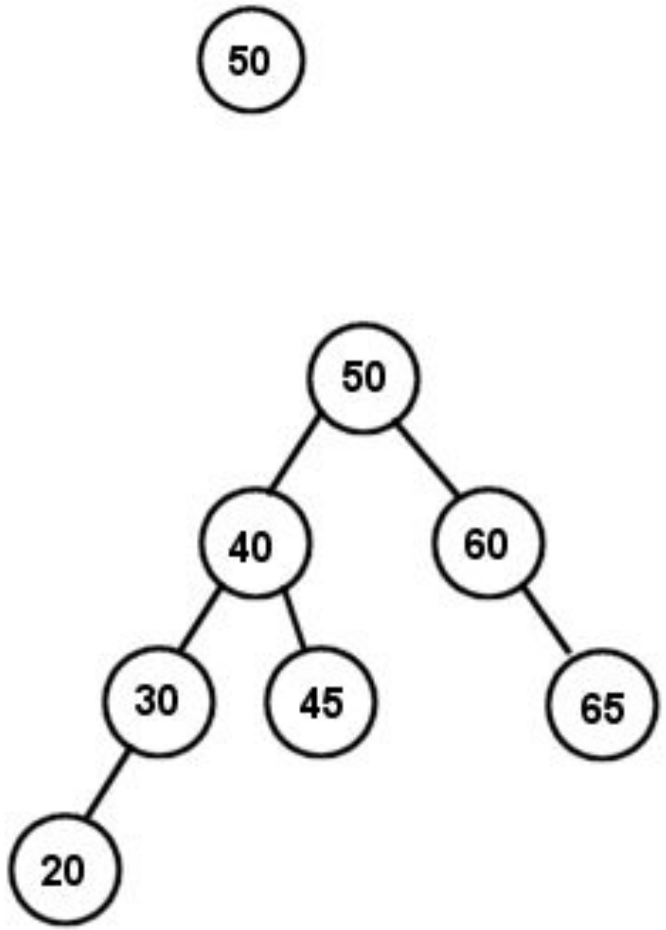
- This is a direct consequence of the BST property.

BST Insertion

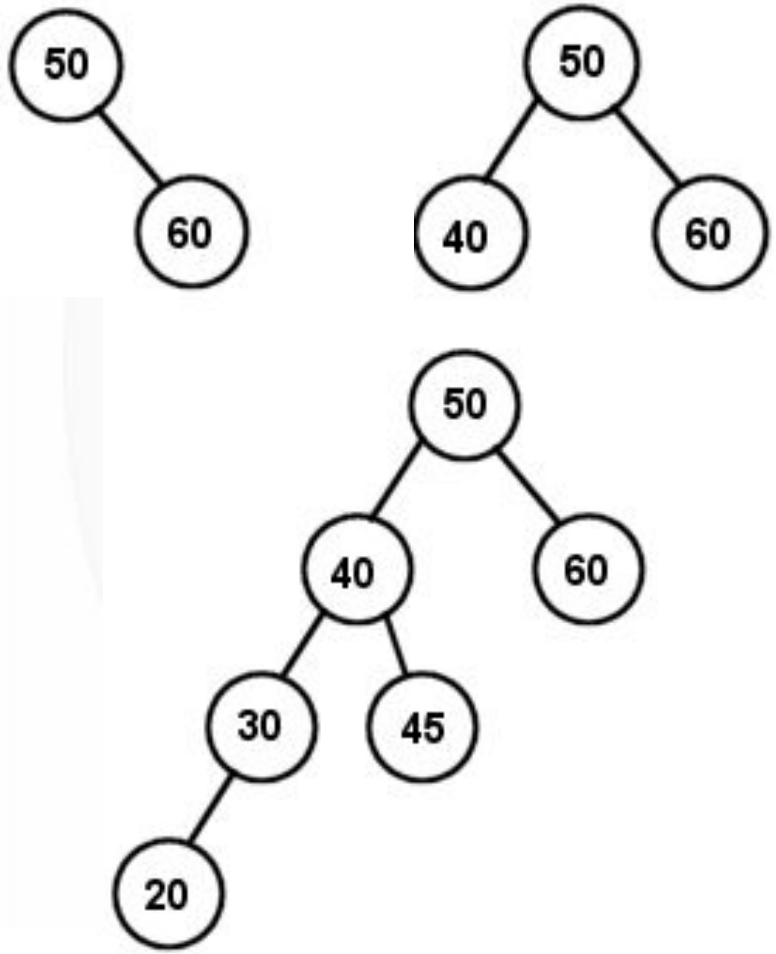
- Three Case
  - Tree is empty
    - Node is Root now
  - Node is less than to Root
    - Traverse left Subtree to find a suitable null space
  - Node is greater than Root
    - Traverse right Subtree to find a suitable null space
- Point to Remember
  - Node is always inserted as leaf if tree is non-empty

BST Insertion

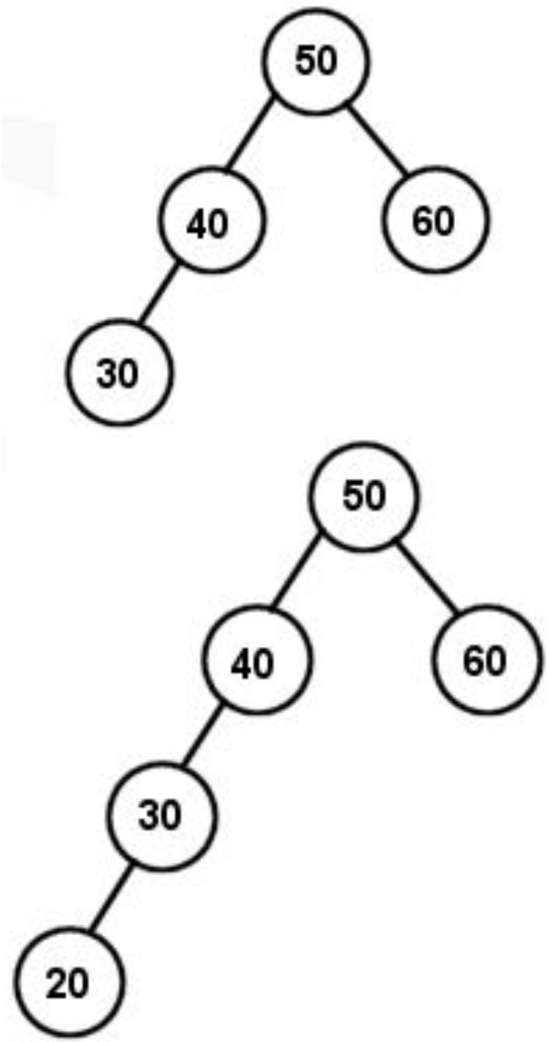
BST– empty



BST– Root only

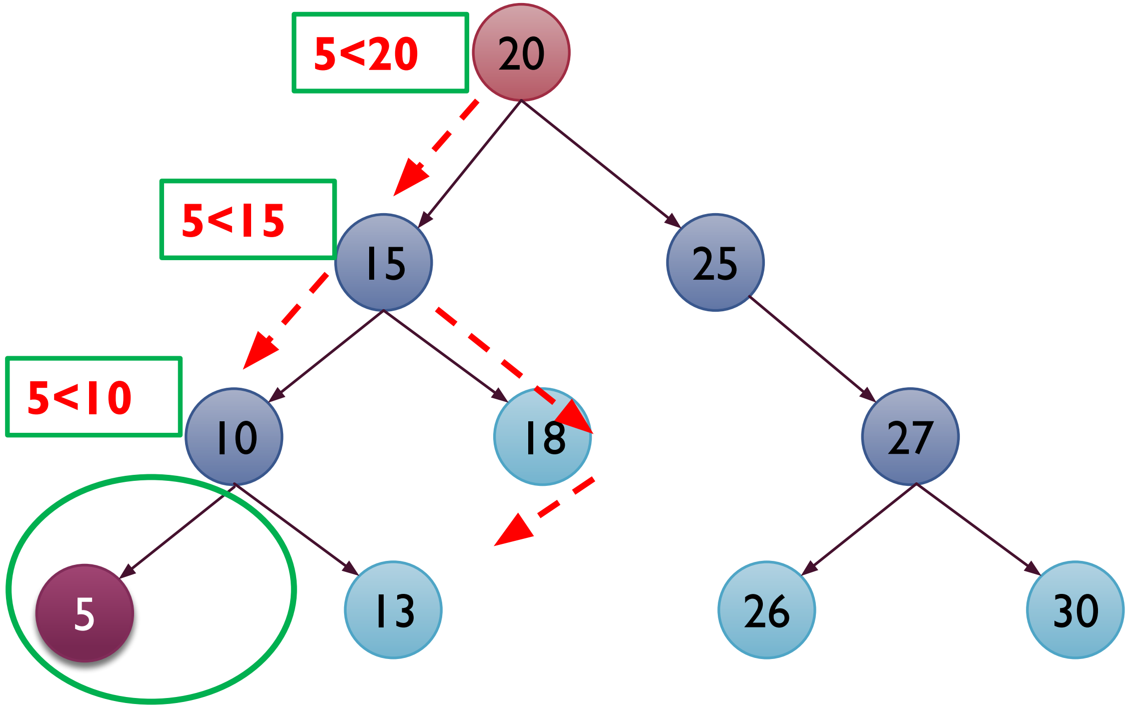


BST– With Multiple Nodes



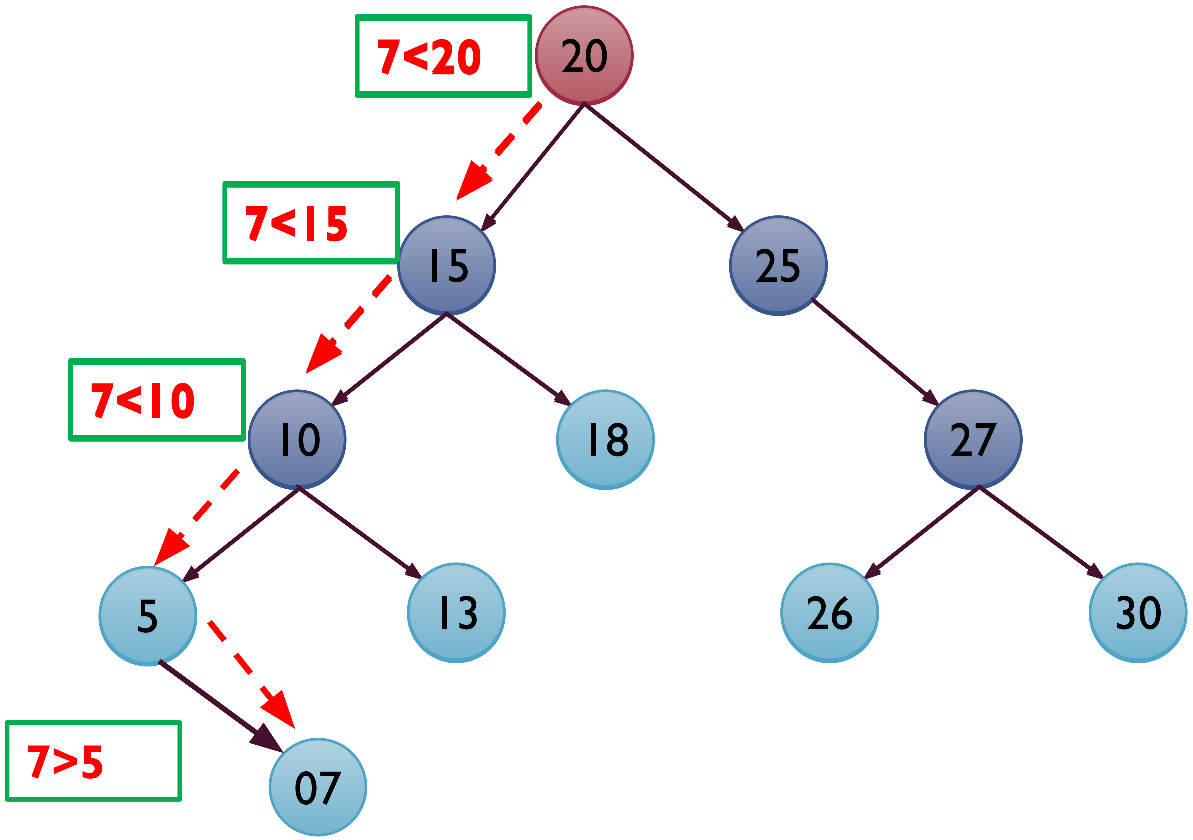
BST Insertion

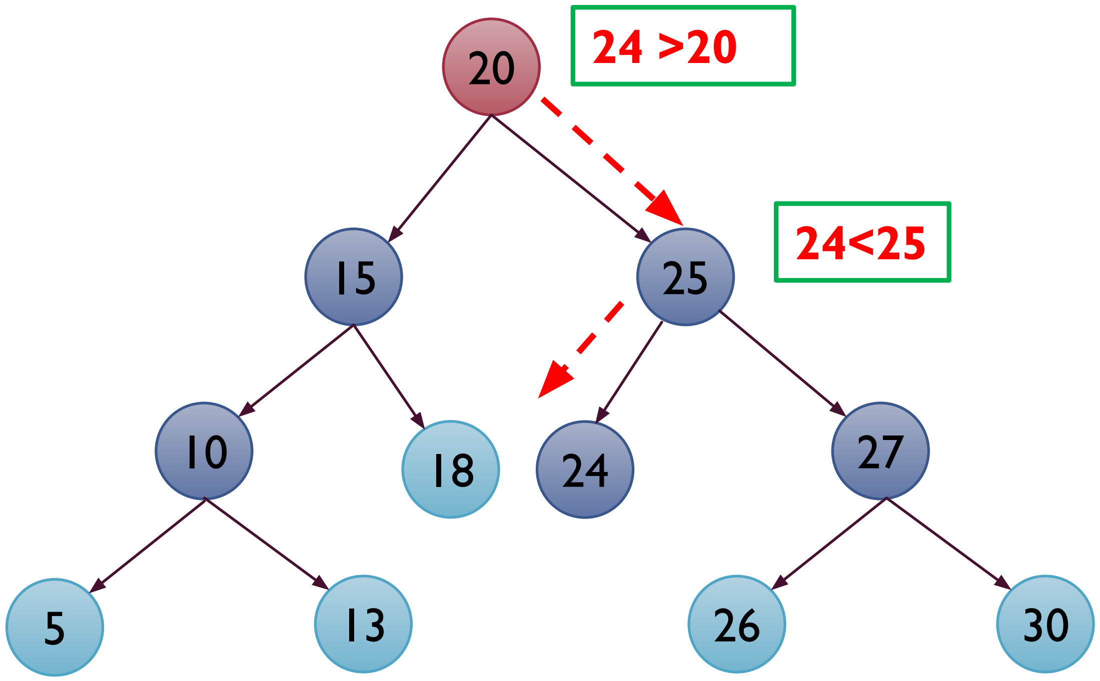
- Point to Remember
  - Node is always inserted as leaf if tree is non-empty
- Let say we want to insert 5,
  - We have reached at a node which is > 5
  - It’s right child >5
  - It does not has left child
  - Make 5 it’s left child



BST Insertion

What will happen if we want to insert 7 & 24





BST Insertion

BSTInsert_Recursive(node , newNode)

- Input: root node of tree for 1st call & new node
- Output: updated tree with newNode
- if(node == NULL)
  - node = newNode
- else if (newNode.data < node.data)
  - BSTInsert(node.Left, newNode)
- else if (newNode.data > node.data)
  - BSTInsert(node.Right, newNode)
- Return node

BST Insertion

BSTInsert_Iterative(node , newNode)

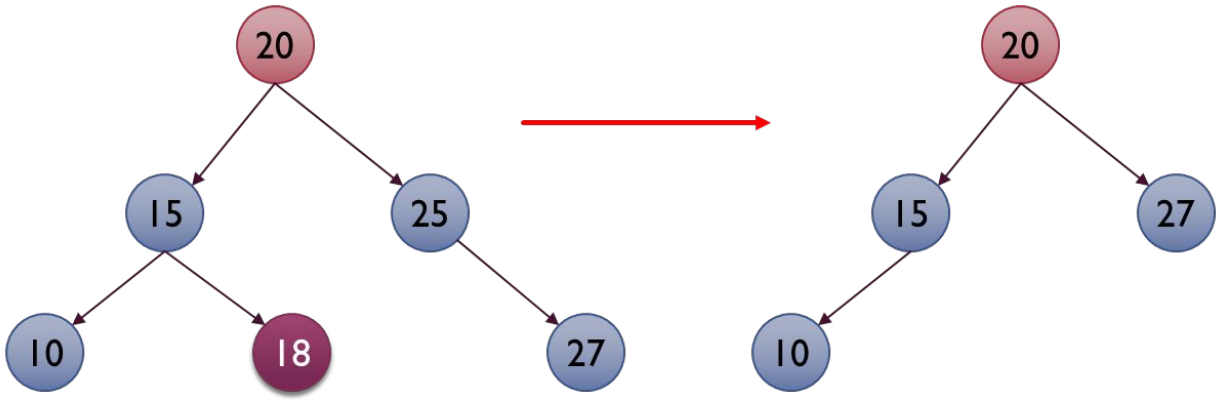
- Parent = Null
- if(node == NULL)
  - node = newNode
- else
  - while (node != Null)
    - Parent = node
    - if (newNode.data < node.data)
      - node = node.left
    - else if (newNode.data > node.data)
      - node = node.right
    - else
      - return
  - End while
- if newNode.data < Parent.data)
  - Parent.left = newNode
- else
  - Parent.right = newNode

BST Deletion

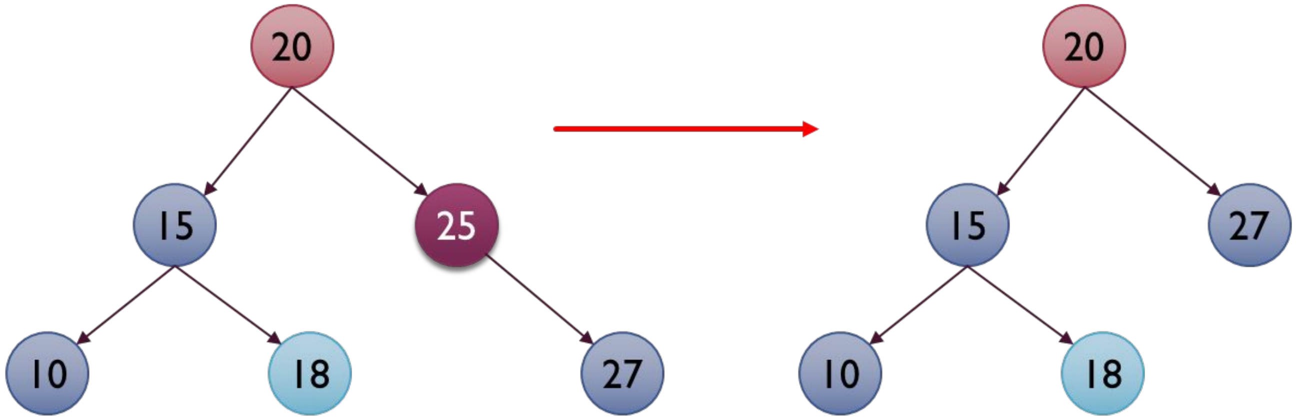
- There are three possible cases to consider:
  - Deleting a leaf (node with no children)
    - Deleting a leaf is easy, as we can simply remove it from the tree.
  - Deleting a node with one child
    - Delete it and replace it with its child.
  - Deleting a node with two children
    - Call the node to be deleted N.
      - Do not delete N.
    - Instead, choose either its inorder successor or its inorder predecessor, R.
    - Copy the value of R to N, then recursively call delete on R until reaching one of the first two cases.

BST Deletion

- If node is leaf node
  - Delete node
- Let Say: Delete 18

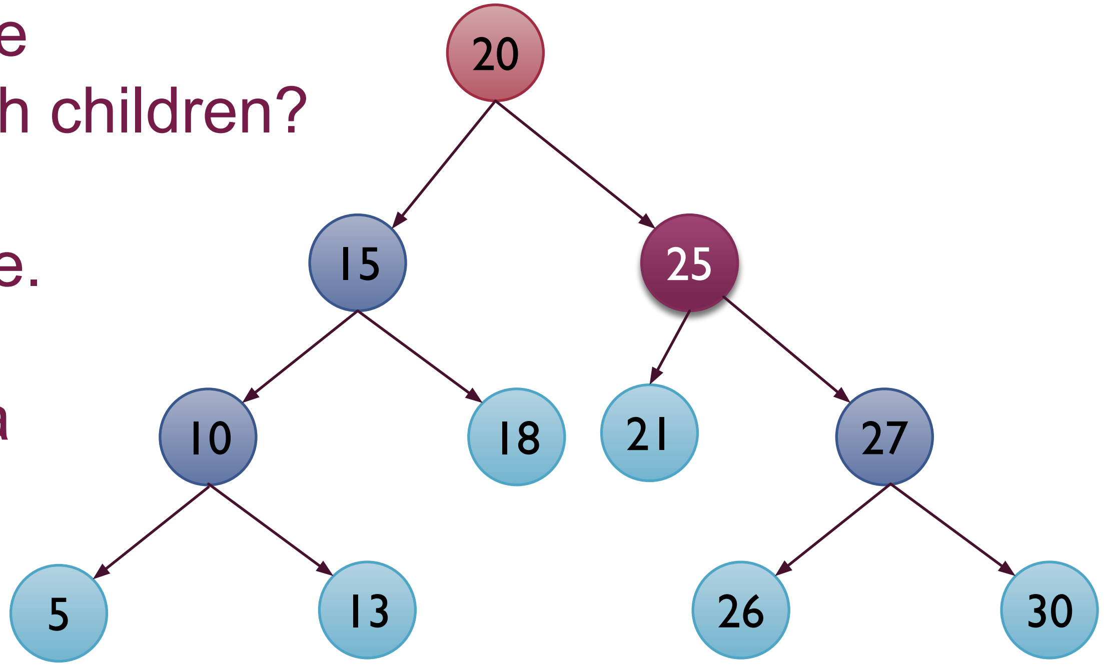


- If node has one child
  - Link parent of node to child of node
  - Delete node
- Let say: Delete 25



BST Deletion

- If Node has two child
  - Consider node 25 in following modified tree
  - Now which node will come above from both children?
  - So that BST property remains in hold.
  - That child node is called successor of node.
  - Find inorder successor of node
  - Replace node's data with successor's data
  - Delete successor



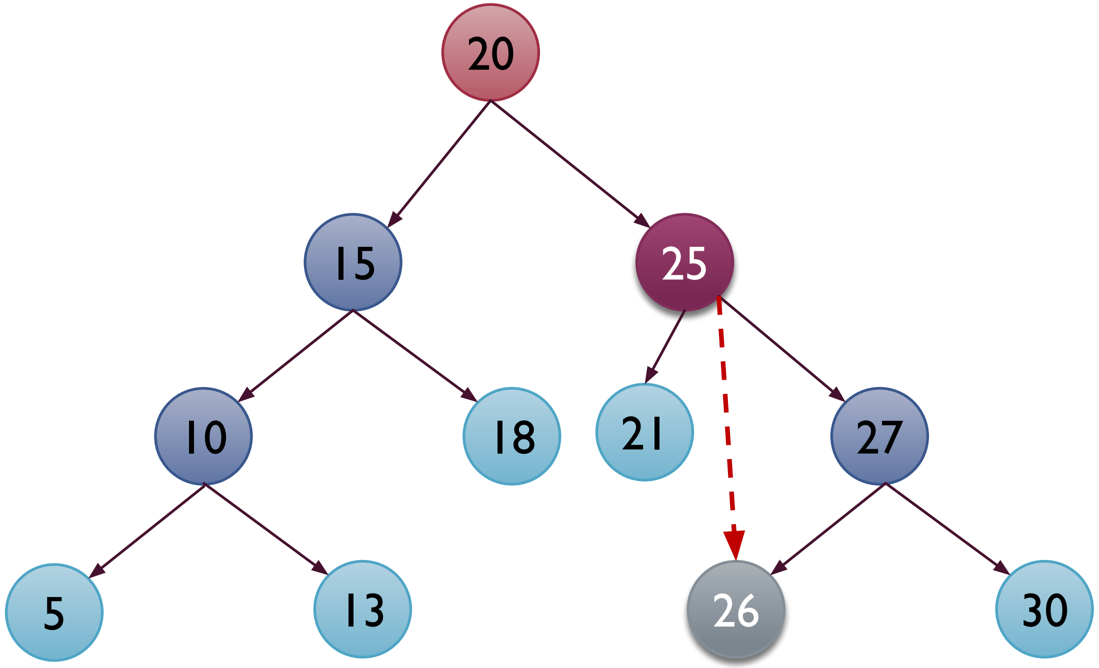
- Inorder Successor of a node is a node
  - Which comes next in inorder traversal.
  - In inorder traversal which node comes after 25?
  - That is 26

BST Deletion

- If Node has two child
  - Find inorder successor of node?
  - Should we go to right subtree to?
  - Or left sub tree?
- What if 26 has left child?
  - It can't

INORDER_SUCCESSOR(node)

- Set curr= node.right
- While(curr.left!=null)
  - curr=curr.left
- End While
- Return curr

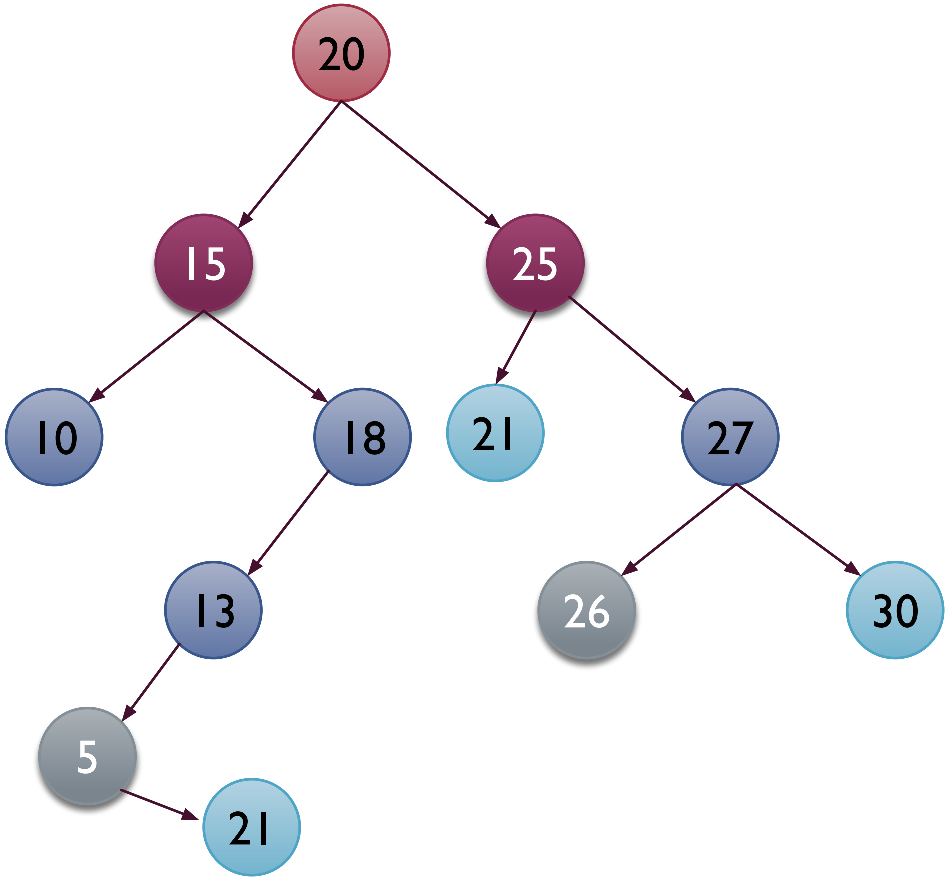


Finding Successor is also called Splicing out

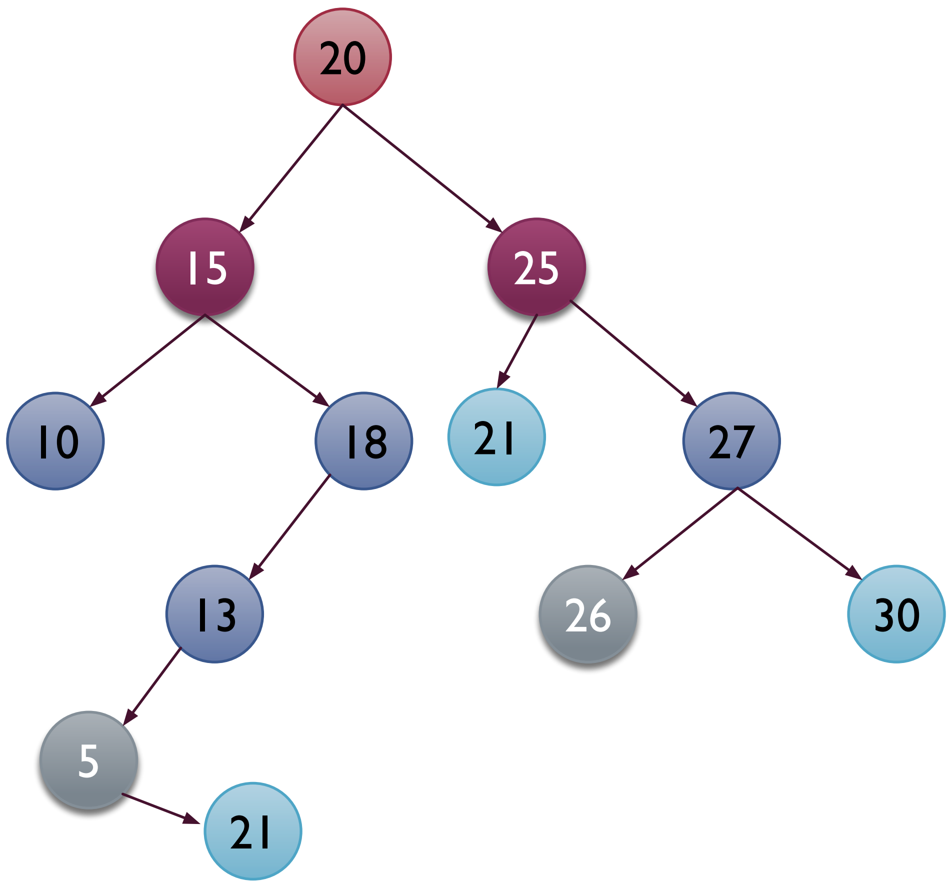
Inorder Successor

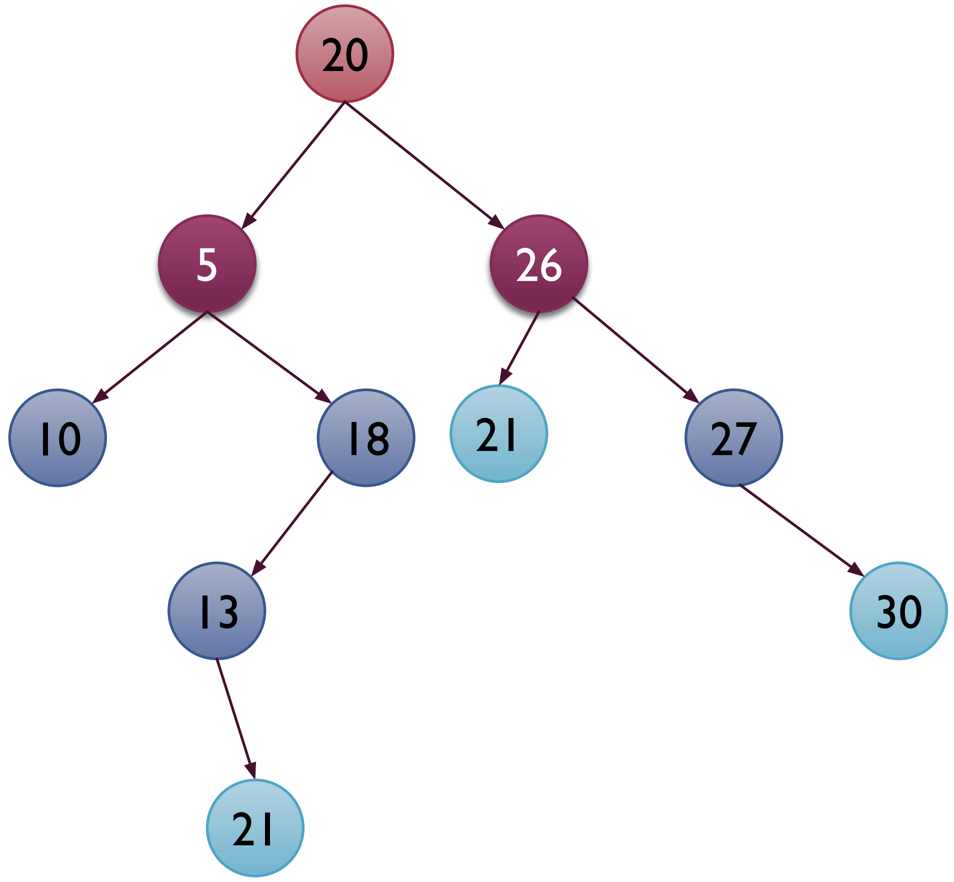
- 5 is inorder successor of 15
- 26 is inorder successor of 25

- If Node has two child
  - Find inorder successor of node?
    - Use INORDER_SUCCESSOR(node)
  - Replace node's data with successor's data
    - Node.data= successor.data
  - Delete successor
    - Again this can be case 1 or 2 (Recursive call)



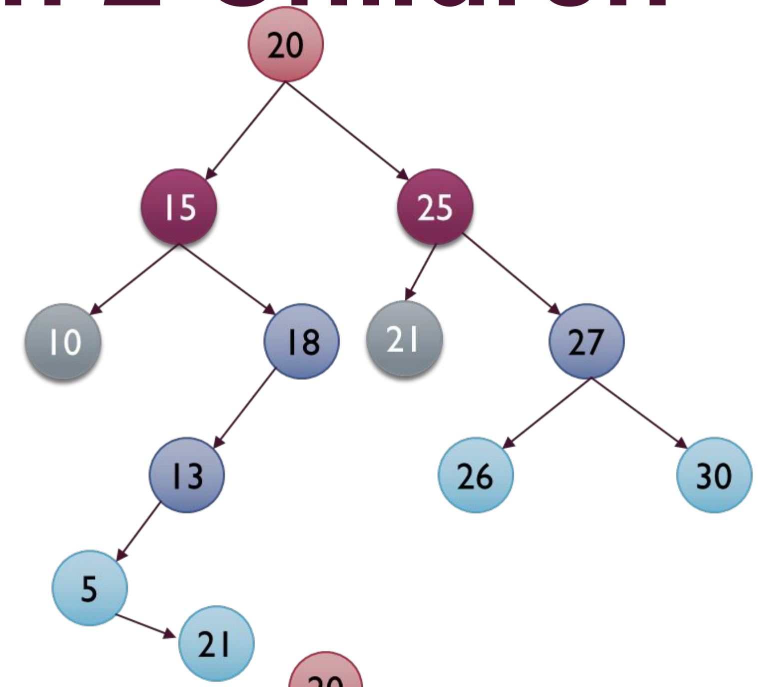
BST Deletion: Internal node with 2 Childern

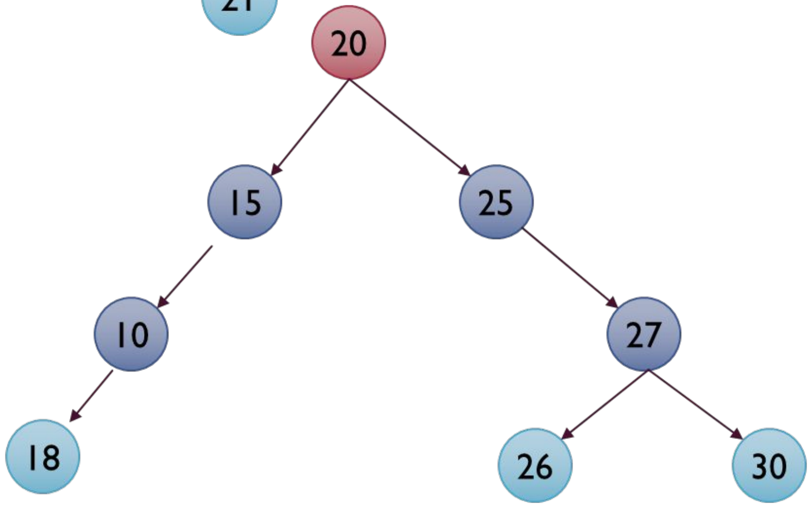




BST Deletion: Internal node with 2 Children

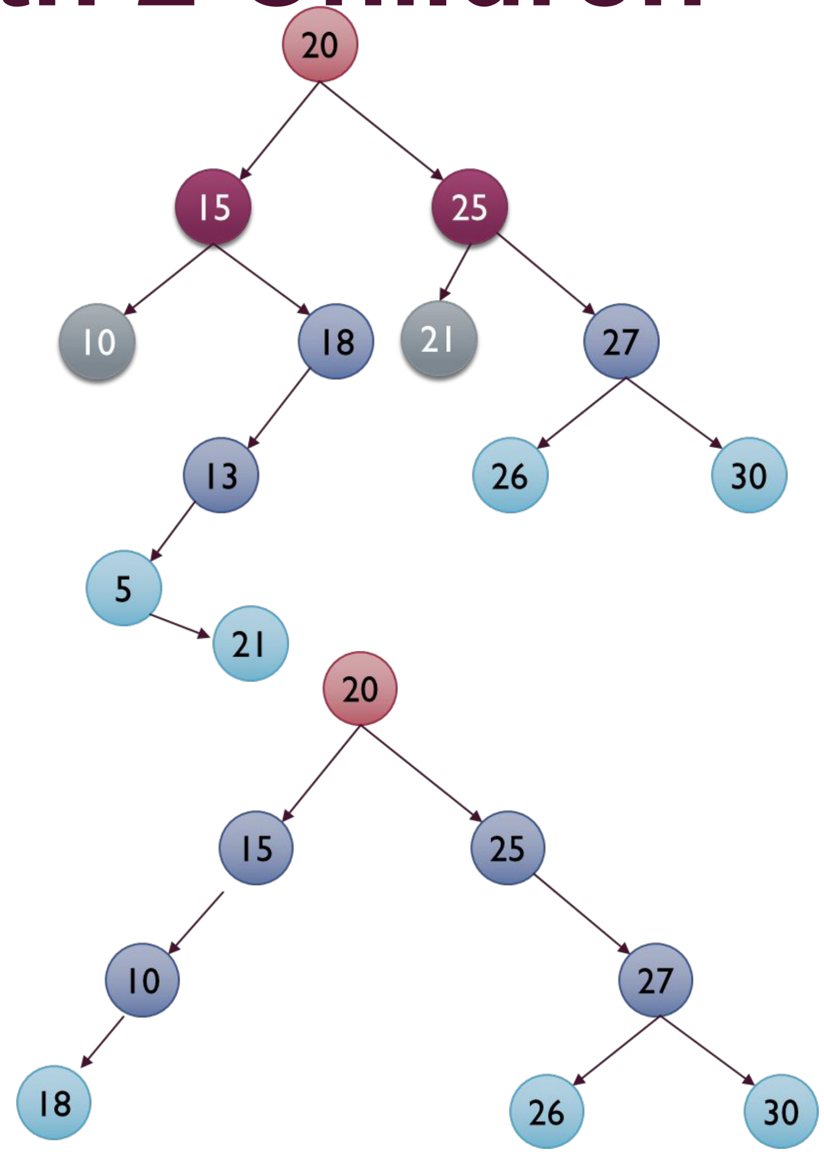
- If Node has two child
  - What if we replace node with its inorder predecessor?
  - Inorder Predecessor of a node is node that comes **before** that node in inorder traversal
  - It is actually **largest node** in left subtree of node
  - Even though it does not affect BST property if you choose successor or predecessor
  - But it is good to **choose randomly** between them to maintain tree balance





BST Deletion: Internal node with 2 Children

- Consider the example
  - if you always pick from one half of tree, eventually tree will become skewed.
  - One side will become extremely short than other



BST Deletion

BSTDelete(node, value)

- Input: root node of tree for 1st call, value to be deleted
- Output: updated tree without node that contains key

- if (value < node.data)
  - BSTDelete(node.left, value)
- else if (value > node.data)
  - BSTDelete(node.right, value)
- else
  - if (node.left && node.right) // if both children are present
    - replacement = **FindMin**(node.right)
    - node.data = replacement.data
    - BSTDelete(node.right, replacement.data)

  - else if (node.left)
    - ReplaceNode(node, node.left) // if 1 child
  - else if (node.right)
    - ReplaceNode(node, node.right) // if 1 child
  - else
    - ReplaceNode(node, NULL) // if 0 child

BST Deletion

BSTDelete(node, value)

- Input: root node of tree for 1st call, value to be deleted
- Output: updated tree without node that contains key

- if (value < node.data)
  - BSTDelete(node.left, value)
- else if (value > node.data)
  - BSTDelete(node.right, value)
- else
  - if (node.left && node.right) // if both children are present
    - replacement = **FindMax**(node.left)
    - node.data = replacement.data
    - BSTDelete(node.left, replacement.data)

  - else if (node.left)
    - ReplaceNode(node, node.left) // if 1 child
  - else if (node.right)
    - ReplaceNode(node, node.right) // if 1 child
  - else
    - ReplaceNode(node, NULL) // if 0 child

BST Deletion

node FindMin(node)

//Gets minimum node (leftmost leaf) in Right subtree

- Set current = node
- while (current.left)
  - current = current.left
- return current

BST Deletion

node FindMax(node)

//Gets maximum node (rightmost leaf) in Left subtree

- Set current = node
- while (current.right)
  - current = current.right
- return current

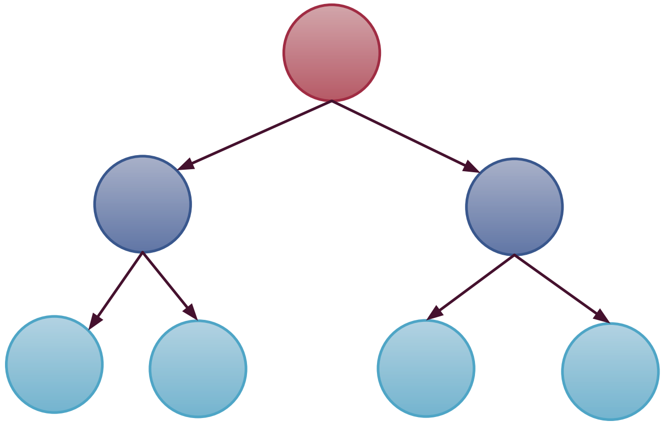
BST Deletion

ReplaceNode (node, child)

- if (child == NULL)
  - delete node
- else
  - node.data = child.data
  - if (child == node.left)
    - node.left = NULL
  - else
    - node.right = NULL
  - delete child

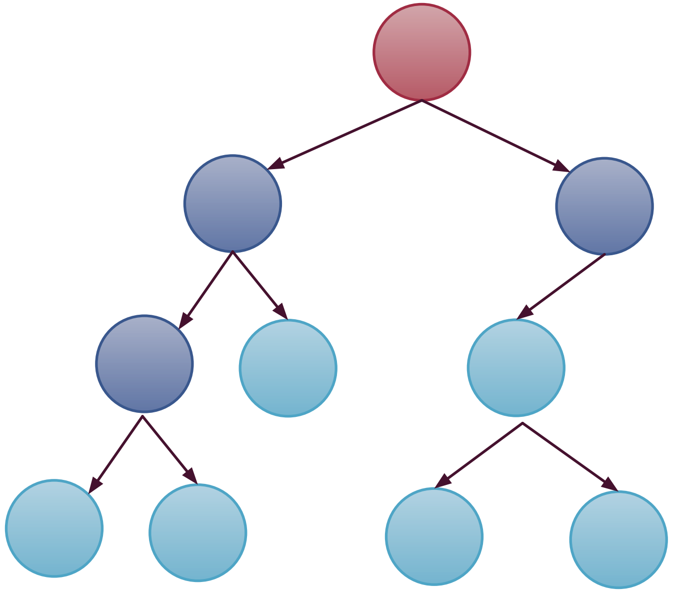
BST Time Complexity

- Depends upon height
- Big-Oh→h



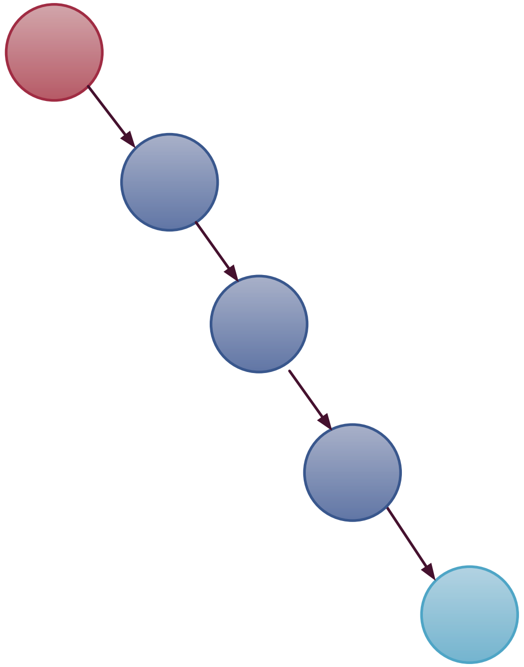
Best

h=log n



Average

h=almost log n



Worst

h=n

BST Time Complexity

Binary Search Tree is a type of balanced tree.

- Linked Structure (size is flexible)
  - Data is stored in a sorted fashion
    - Search: O (log n)
    - Insertion: O (log n)
    - Deletion: O (log n)

- Sorted Array (Fixed Size)
  - Need to know the size of the largest data set
  - Space wastage
    - Search: O (log n)
    - Insertion: O (n)
    - Deletion: O (n)