بسم الله الرحمن الرحيم

Data Structure

Chapter 6

Lecture 12:
The Tree Data Structure

In this lecture, we will cover:

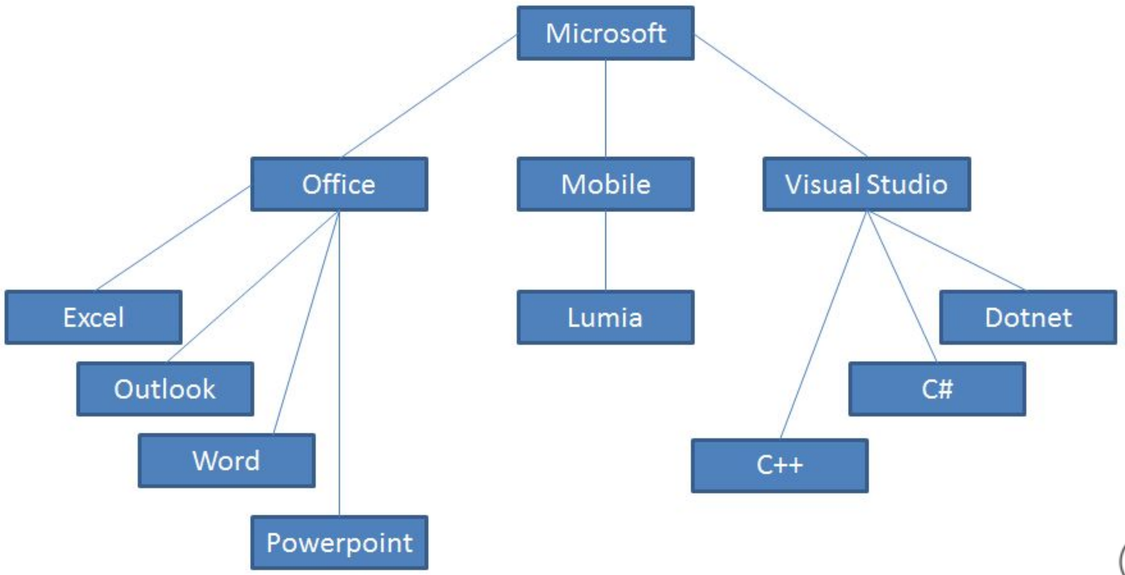
- Definition of a tree data structure and its components
- Concepts of:
  - Root, internal, and leaf nodes
  - Parents, children, and siblings
  - Paths, path length, height, and depth
  - Subtrees
  - Types of trees
  - Tree traversals
- Examples

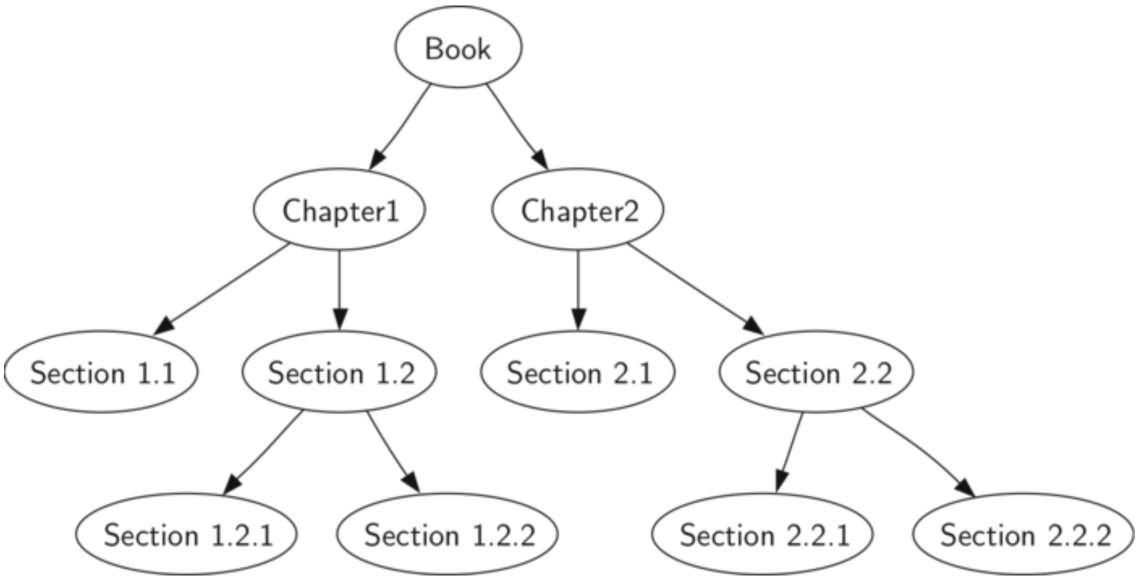
Trees

- Linear lists are useful for ordered data
  - $(e_0, e_1, e_2, \ldots, e_{n-1})$
  - Days of week
  - Months in a year
  - Students in this class

- Trees are useful for ordered data
  - **Employees of a corporation**
    - President, vice presidents, managers, and so on …
  - **Java's classes**
    - Object is at the top of the hierarchy
    - Subclasses of Object are next, and so on

Examples of Trees

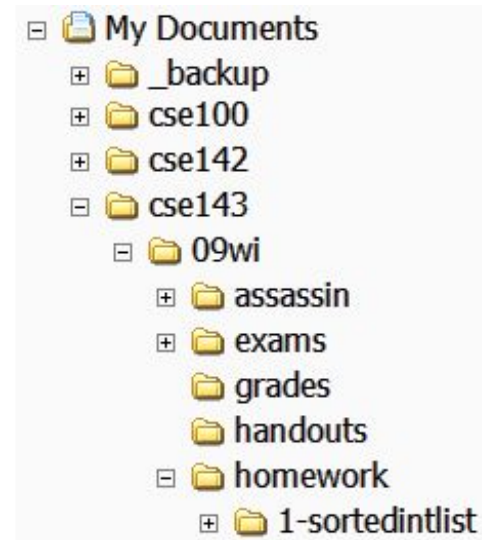


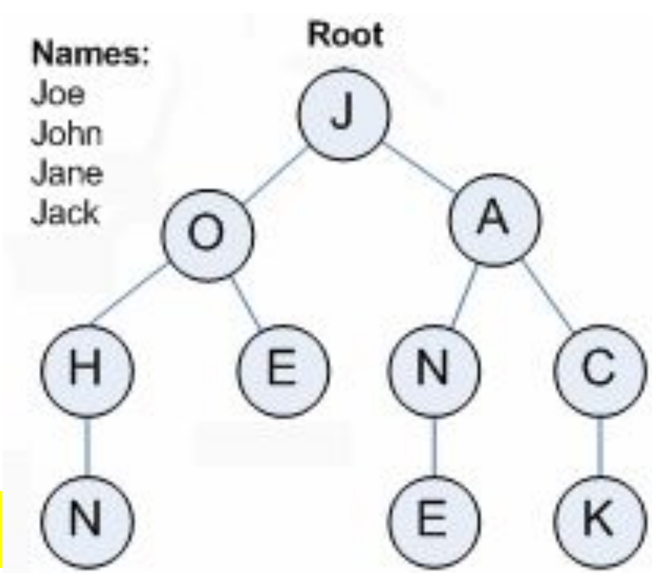


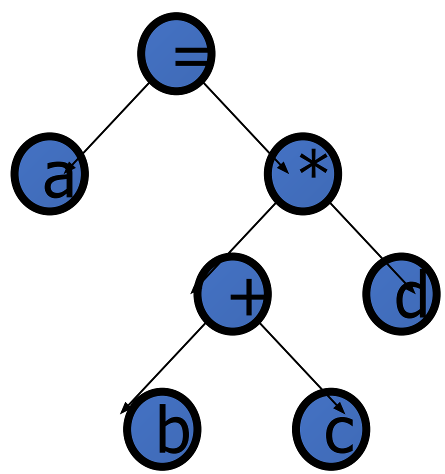
Trees in Computer Science

- **Folders/files** on a computer
- **Family genealogy** organizational charts
- **AI** decision trees
- **Compilers** with parsing trees

  a = (b + c) * d;
- Cell phone **T9**



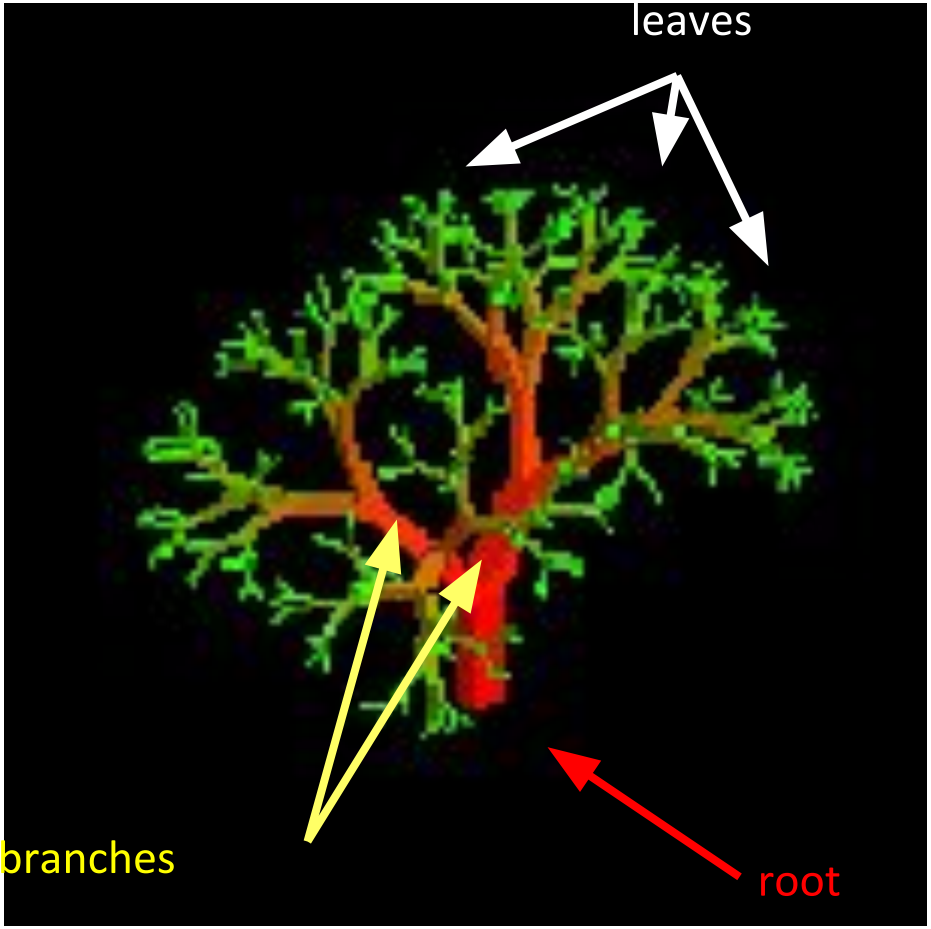




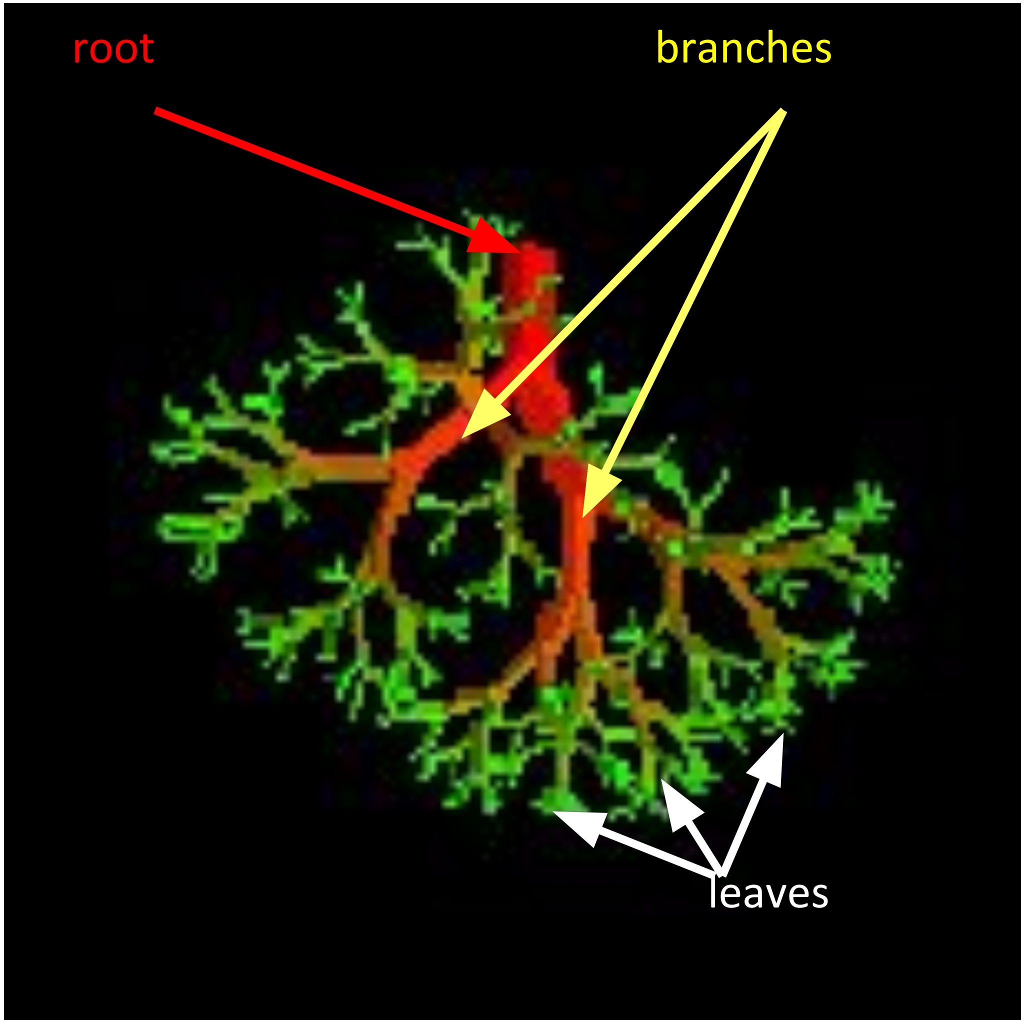
Applications of Trees

1. Store hierarchical data, like folder structure, organization structure
2. Binary Search Tree is a tree that allows fast search, insert, delete on a sorted data. It also allows finding closest item
3. Heap is a tree data structure which is implemented using arrays and used to implement priority queues.
4. B-Tree and B+ Tree : They are used to implement indexing in databases.
5. Syntax Tree: Used in Compilers.
6. Spanning Trees and shortest path trees are used in routers and bridges respectively in computer networks

Nature Lover's View of a Tree



Computer Scientist's View of a Tree



What Are Trees?

A *Tree* structure means that the data are organized so that items of information are related by branches

Definition of Tree

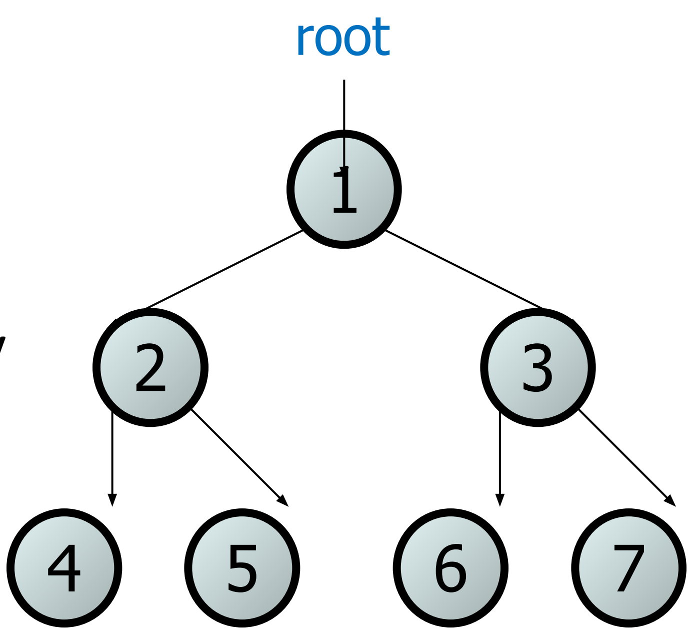
- Tree is a finite set of one or more nodes such that:
  - There is a specially designated node called the root
  - The remaining nodes are partitioned into n>=0 disjoint sets T1, ..., Tn, where each of these sets is a tree
- We call T1, ..., Tn the subtrees of the root

Trees

- A **Tree** is a directed, acyclic structure of linked nodes
  - **directed**: Has one-way links between nodes
  - **acyclic**: No path wraps back around to the same node twice

- **Binary Tree:** One where each node has **at most two children**

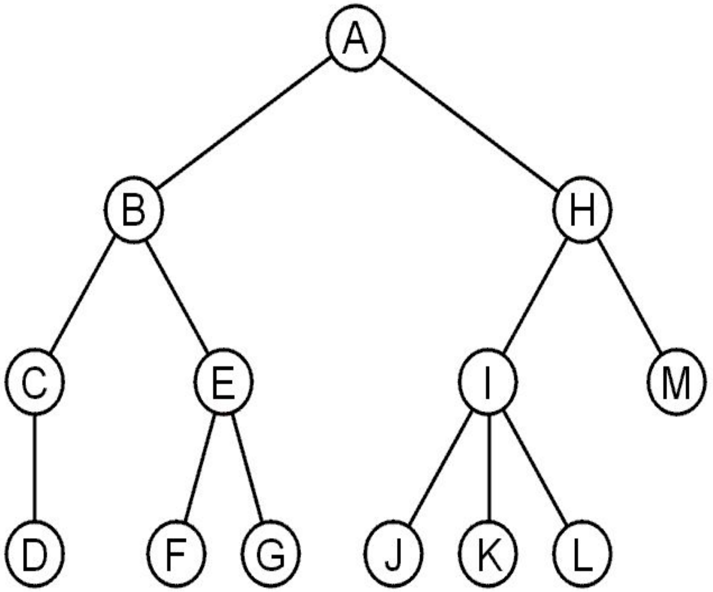
- A **Binary Tree** can be defined as either:
  - empty (`null`), or
  - a **root** node that contains:
    - **data**
    - a **left subtree** and a **right subtree**
    - either (or both) subtrees could be empty



The Tree Data Structure

A rooted tree data structure stores information in nodes

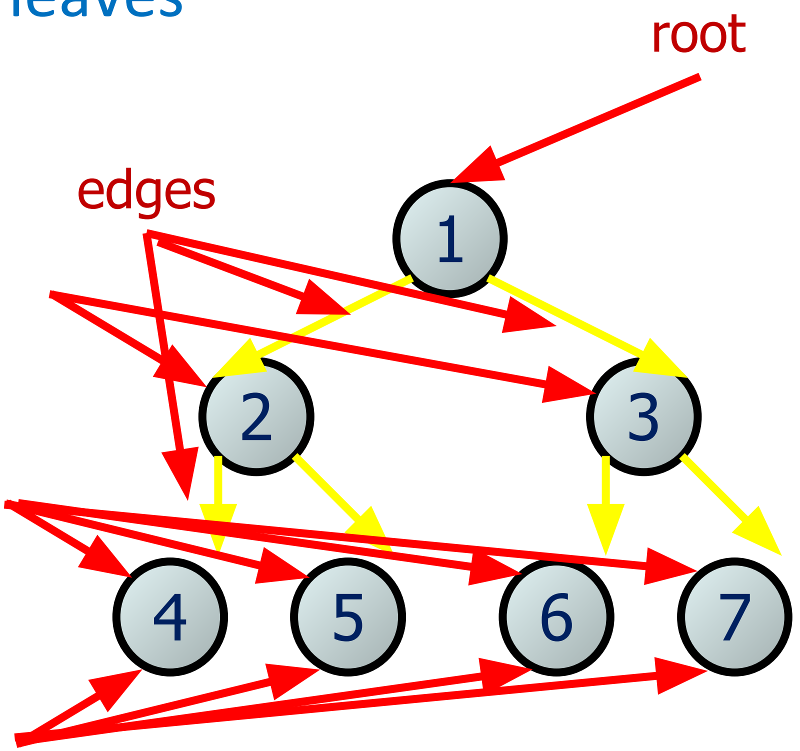
- Tree
  - Similar to linked lists:
    - There is a first node, or *root*
    - Each node has variable (not just one) number of references to successors
    - Each node, other than the root, has exactly one node pointing to it



Tree Terminology

- 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 that have no children are leaves
- Nodes can be called Vertices
- Connections between vertices

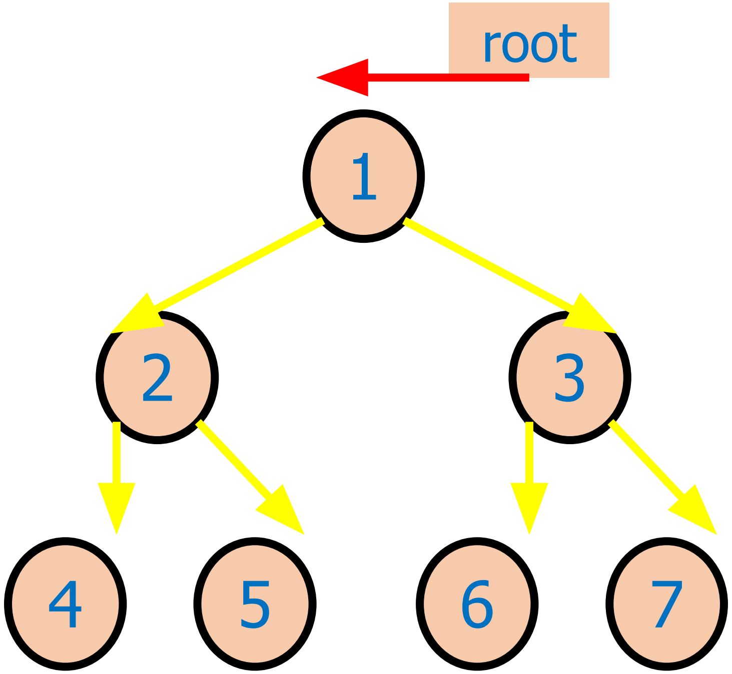
  can be called edges



Tree Terminology

- **Node/Vertix**: an object containing a data value (parent and/or child)

- **Edge:** connection between nodes

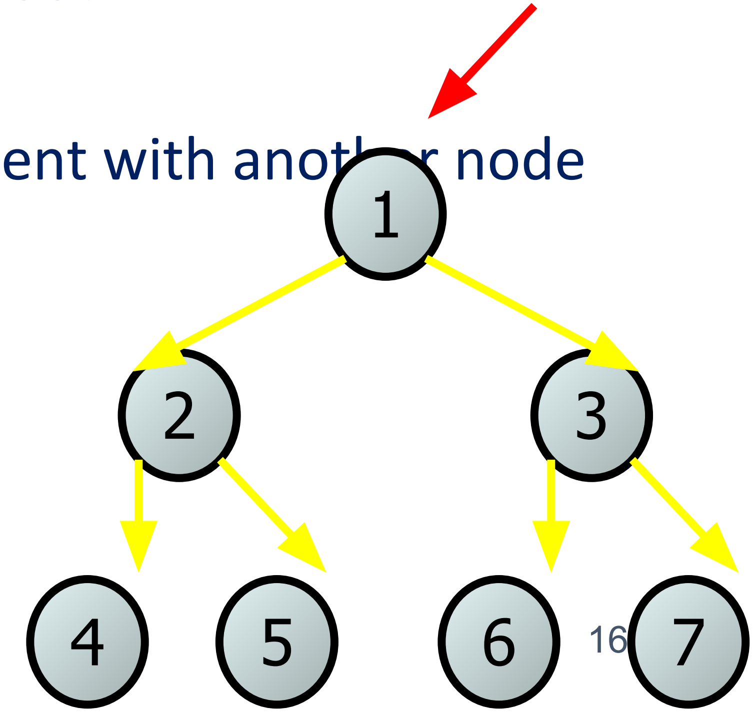


Tree Terminology cont'd

- **Parent node**: Exactly one node that refers by a directed edge to the current node (Does not apply to the root)

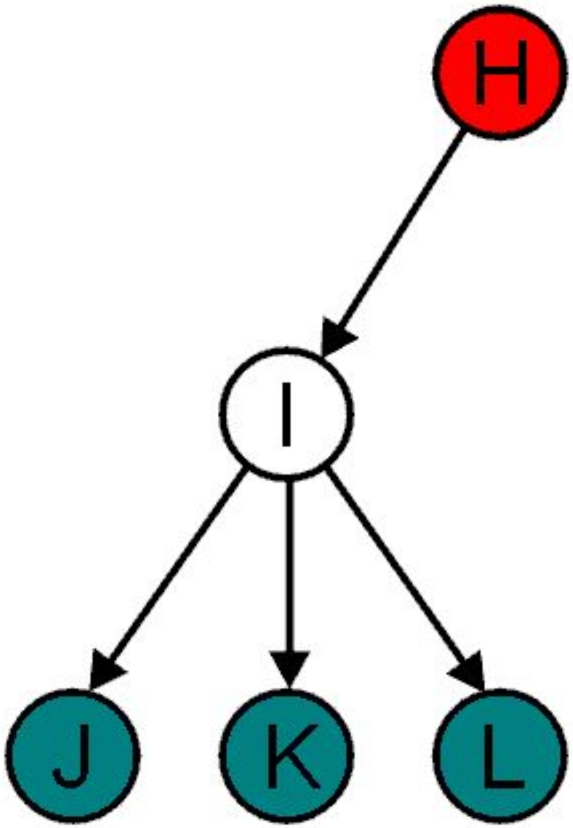
- **Child node**: a node directly connected to another node that this node refers to when moving away from the root.

- **Sibling node**: a node with common parent with another node



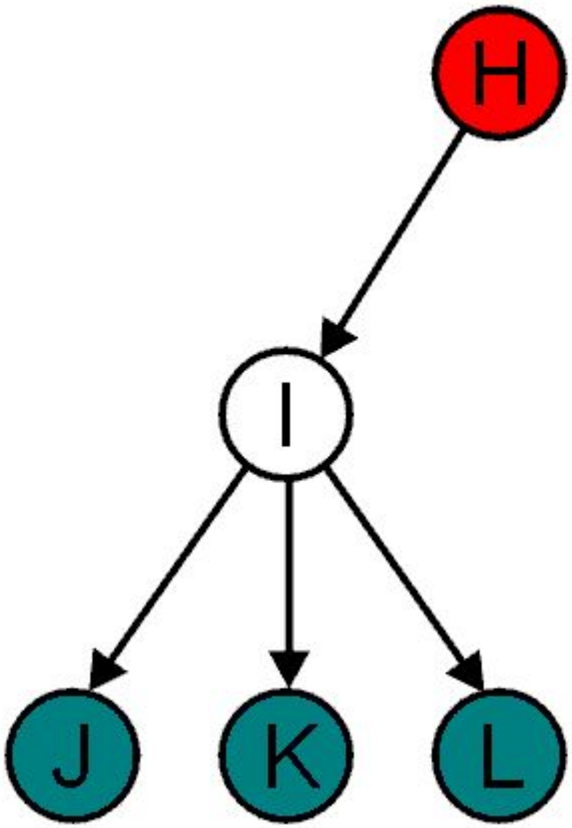
Tree Terminology cont'd

- All nodes will have zero or more child nodes or *children*
  - **I** has three children: J, K and L
- For all nodes other than the root node, there is one parent node
  - H is the parent I



Tree Terminology cont'd

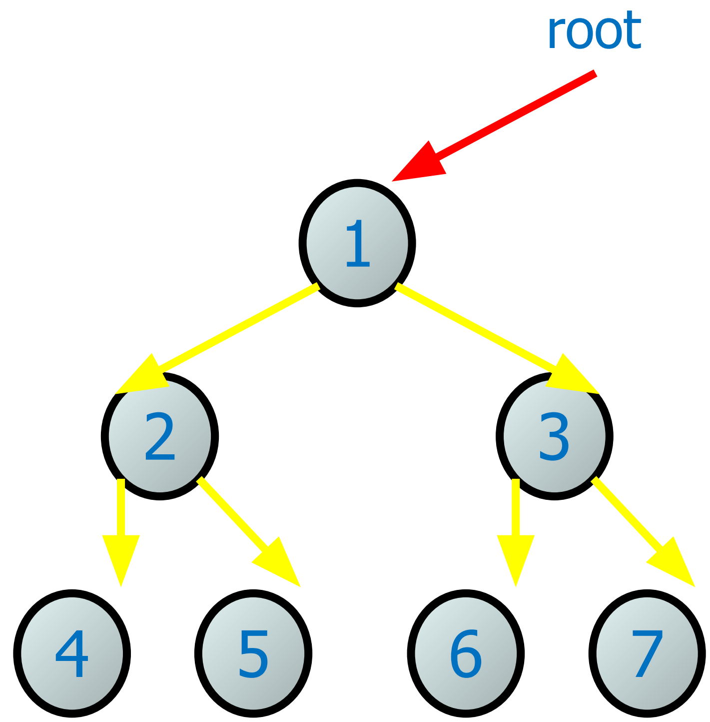
- Nodes with the same parent are *siblings*
  - J, K and L are siblings
- Ancestors of a node are all nodes along the path from the root to the node
  - I is the ancestor of J, K and L



Tree Terminology cont'd

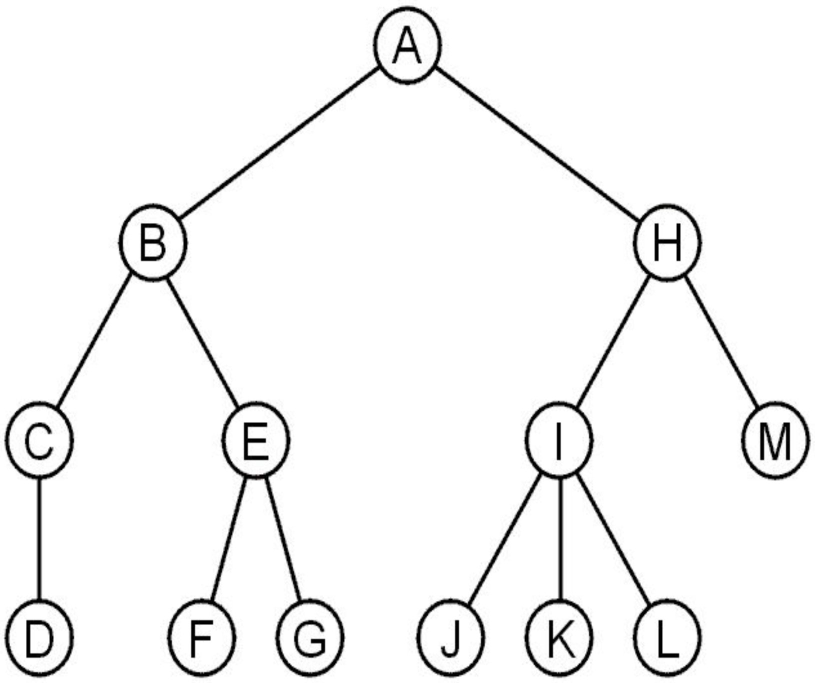
- **Root node**: topmost node of a tree
- **Leaf/External node**: a node that has no children
- **Branch/Internal node:** node with at least one child

  (neither the root nor a leaf)



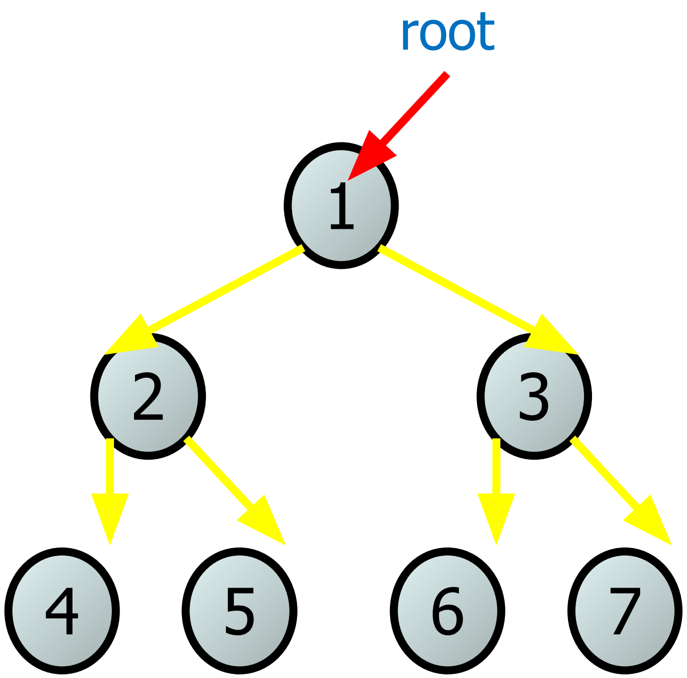
Tree Terminology cont'd

- Nodes with no children are called *leaf nodes*
- All other nodes are said to be *internal nodes*, that is, they are internal to the tree



Tree Terminology cont'd

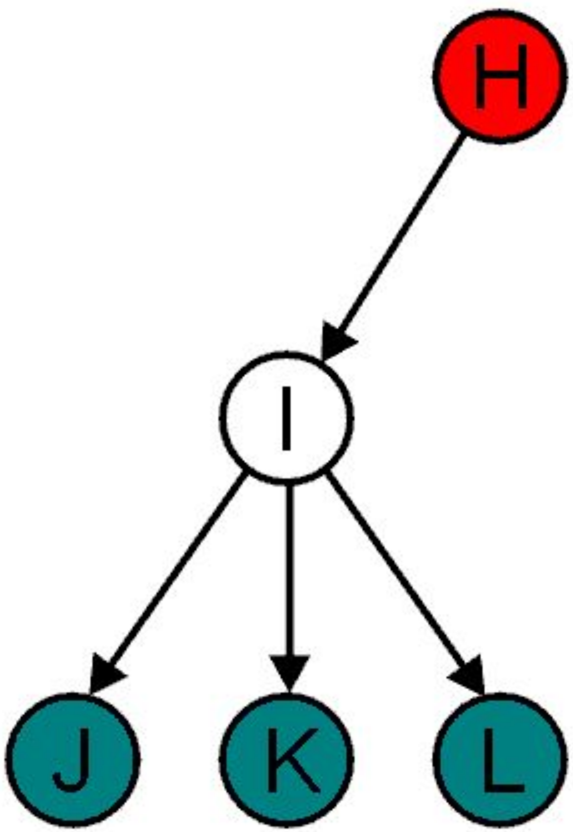
- **Degree of a node:** Number of its children
- **Depth of a node:** Number of edges from root to the node
- **Height of a node:** Number of edges from the node to the deepest leaf
- **Height of a tree:** is the height of its root



Tree Terminology cont'd

- The ***degree*** of a node is defined as the number of its children

  deg(I) = 3

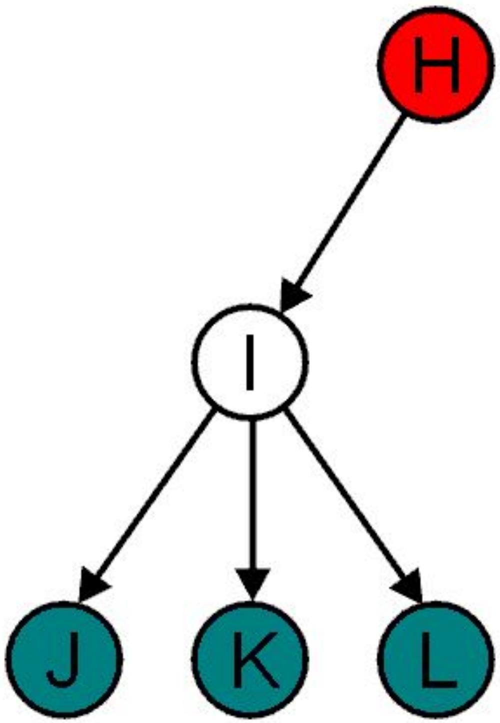


Tree Terminology cont'd

- The *height* of a tree is defined as the maximum depth of any node within the tree (H is the root)

  Height (Tree) = 2

- The height of a tree with one node (Just the root node) is 0
- For convenience, we define the height of the empty tree to be –1

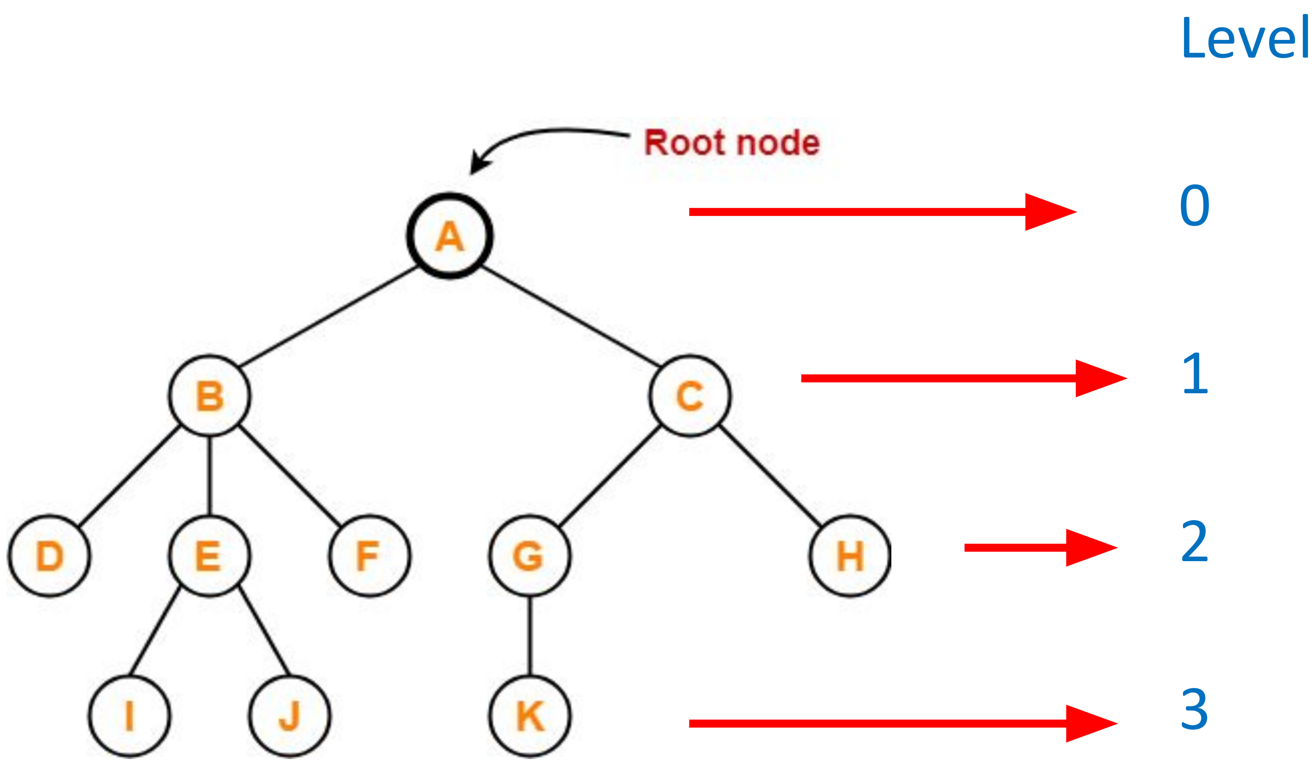


Level and Depth

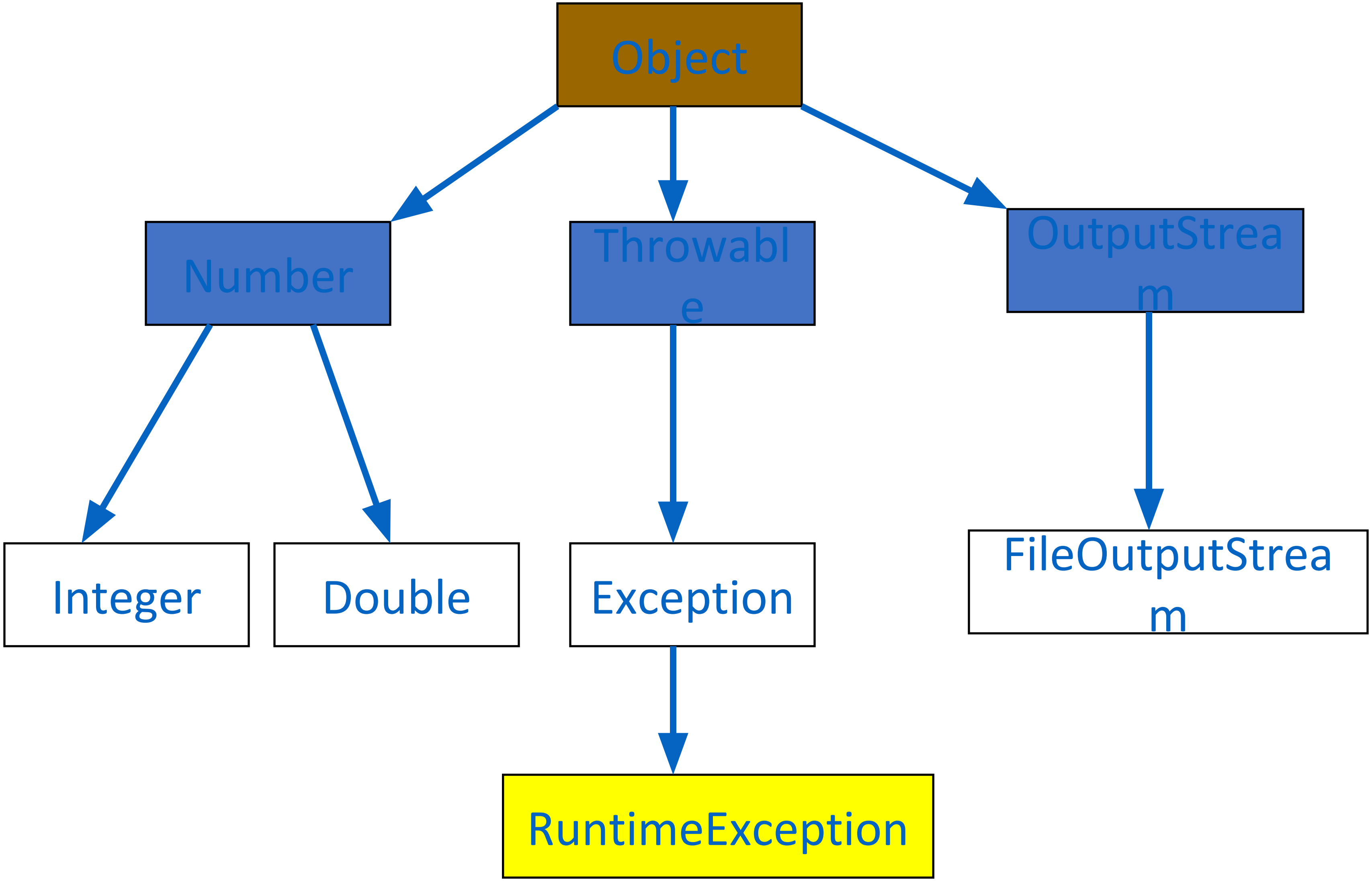
Node = 11

Degree of a tree = 3

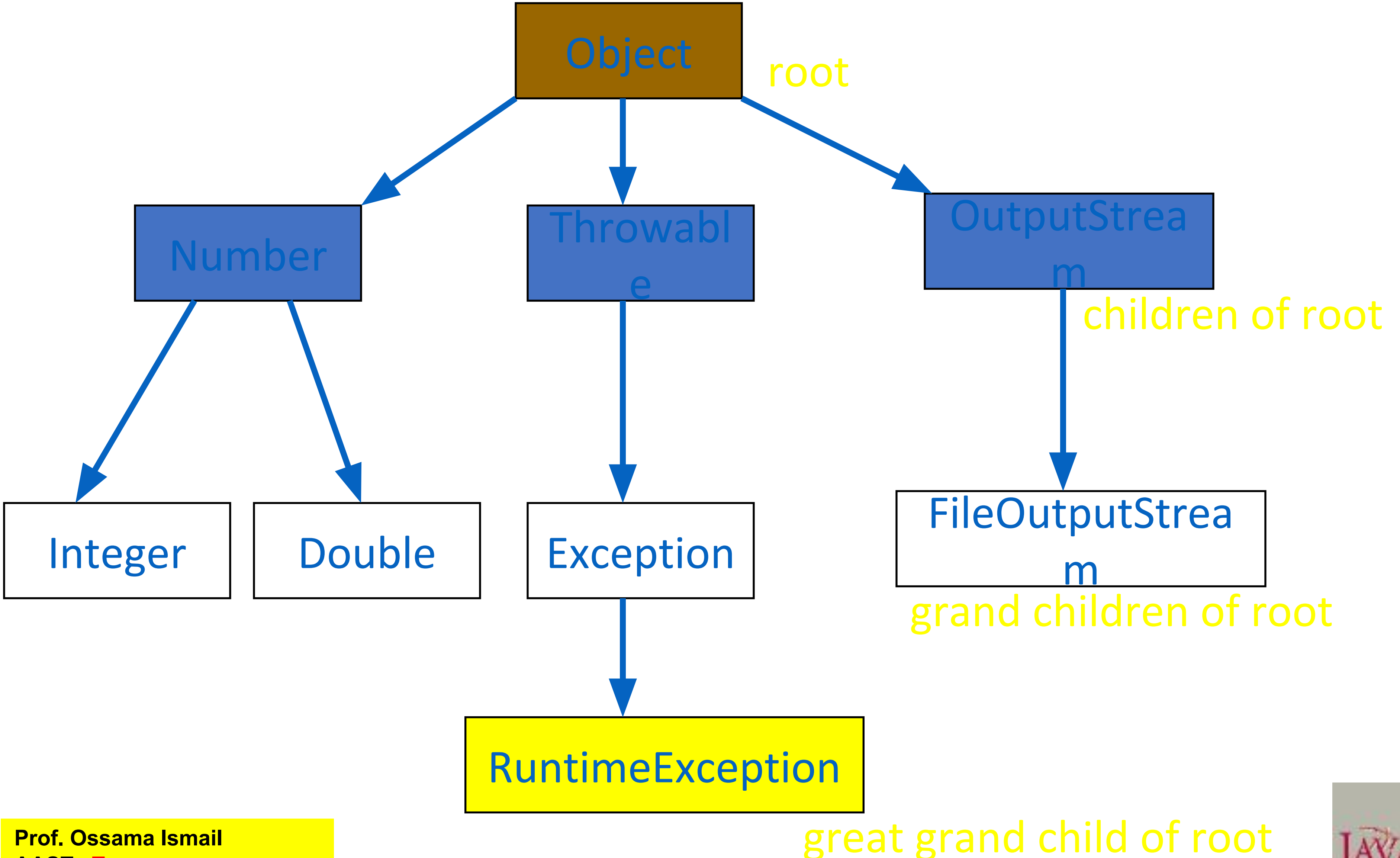
Height of a tree = 3



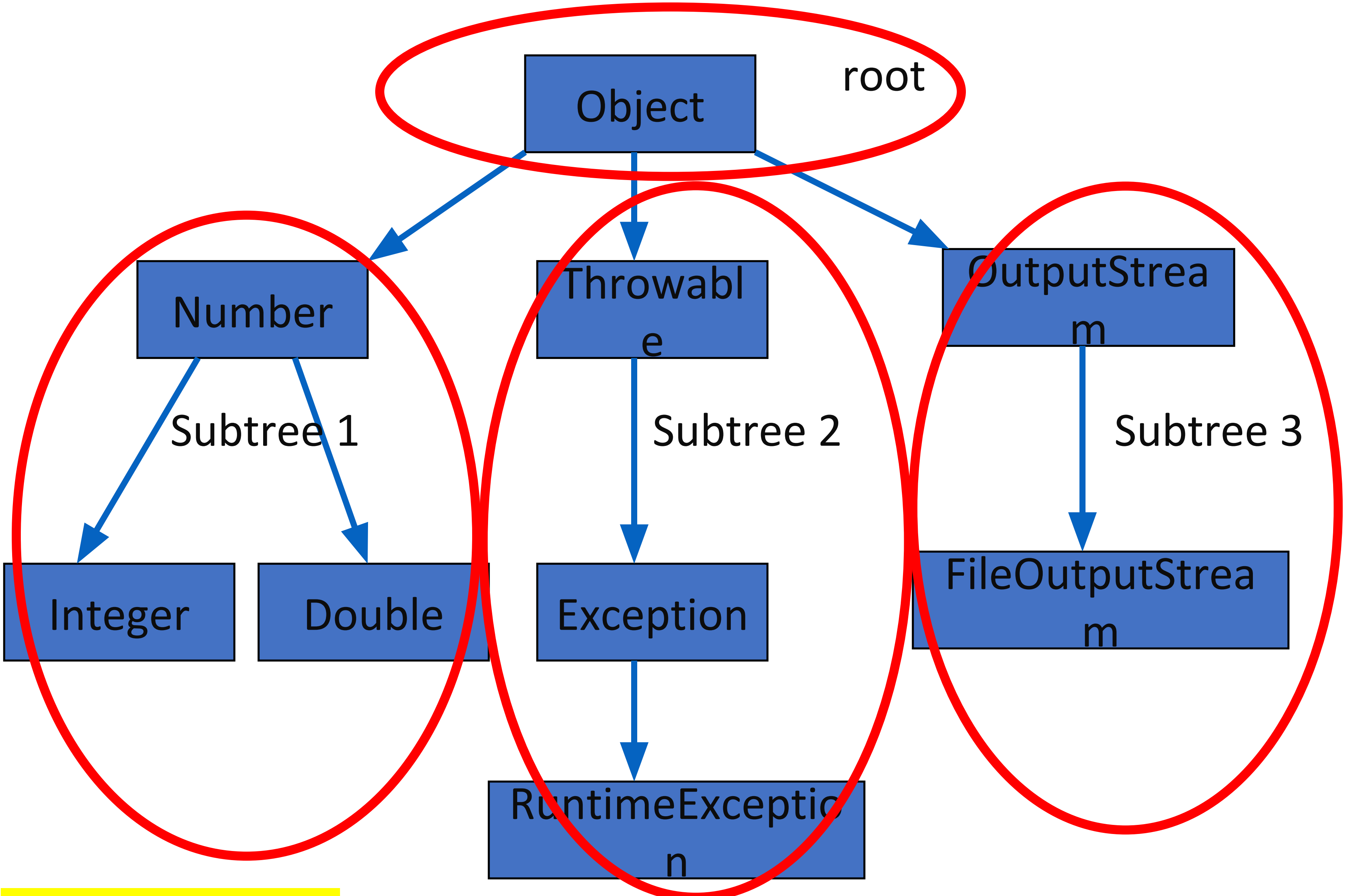
Examples: Java's Classes



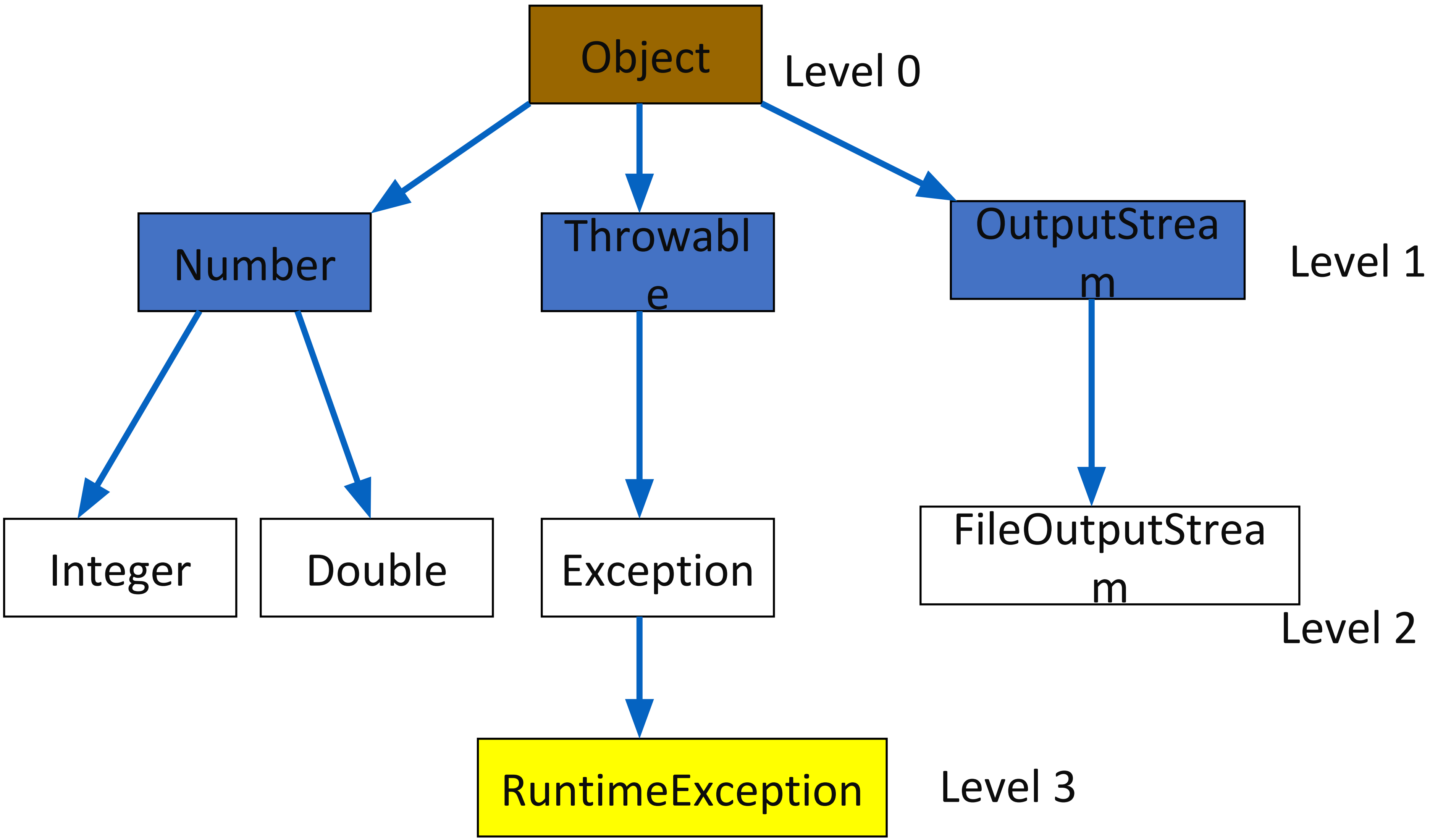
Examples: Children, Grand Children, …



Examples: Subtrees

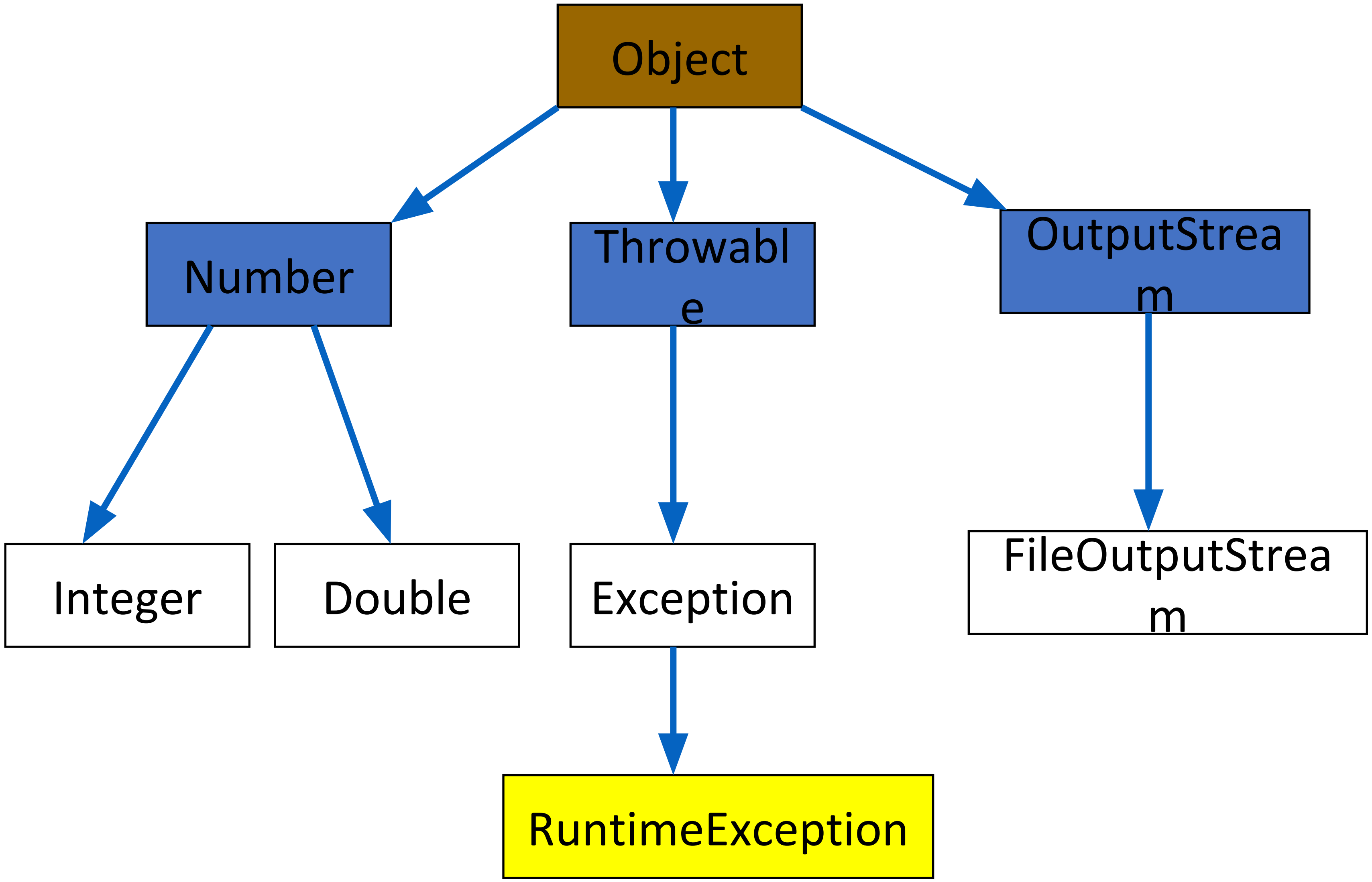


Examples: Levels

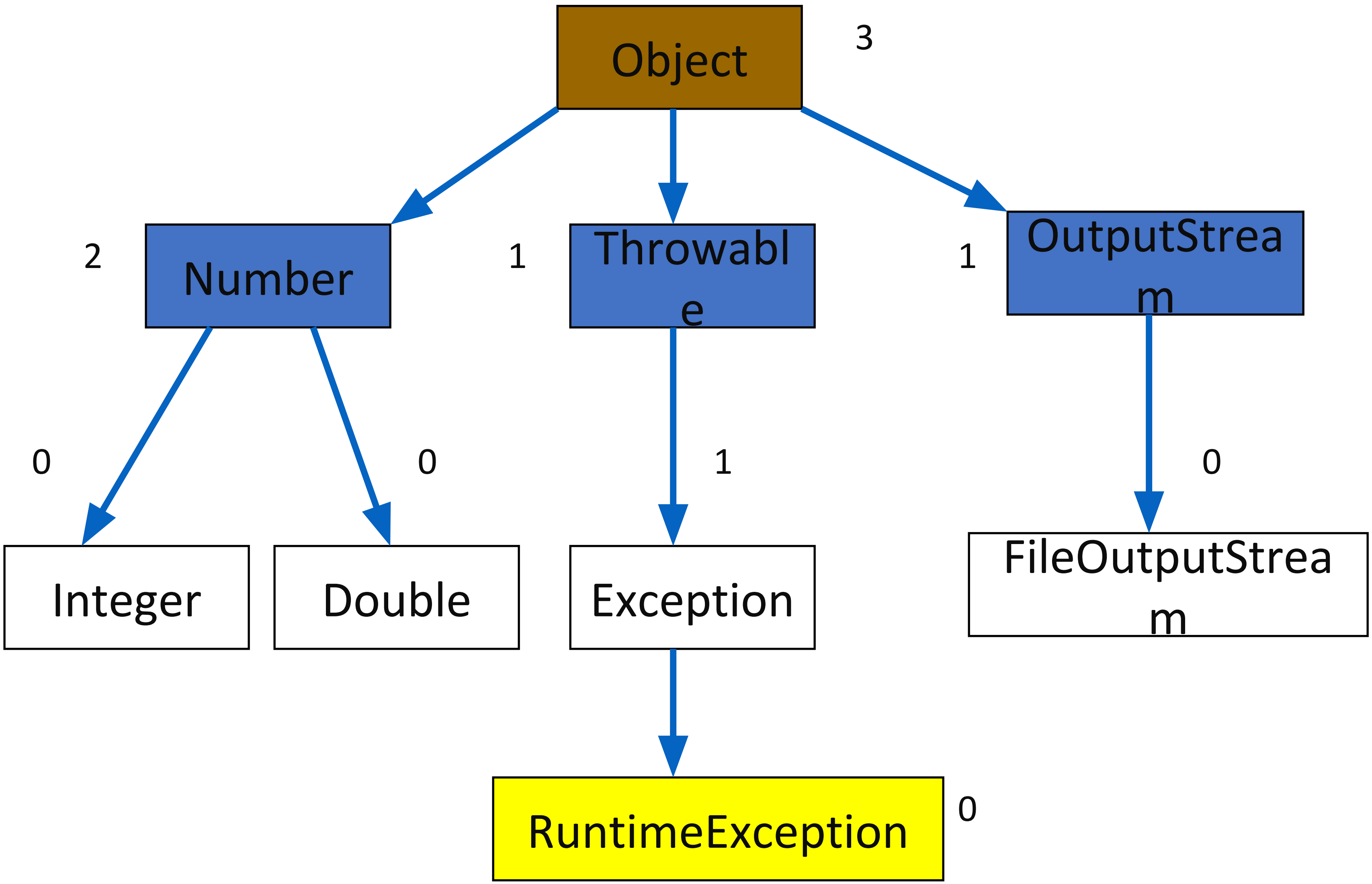


Examples:

Height = Depth = Number of Levels

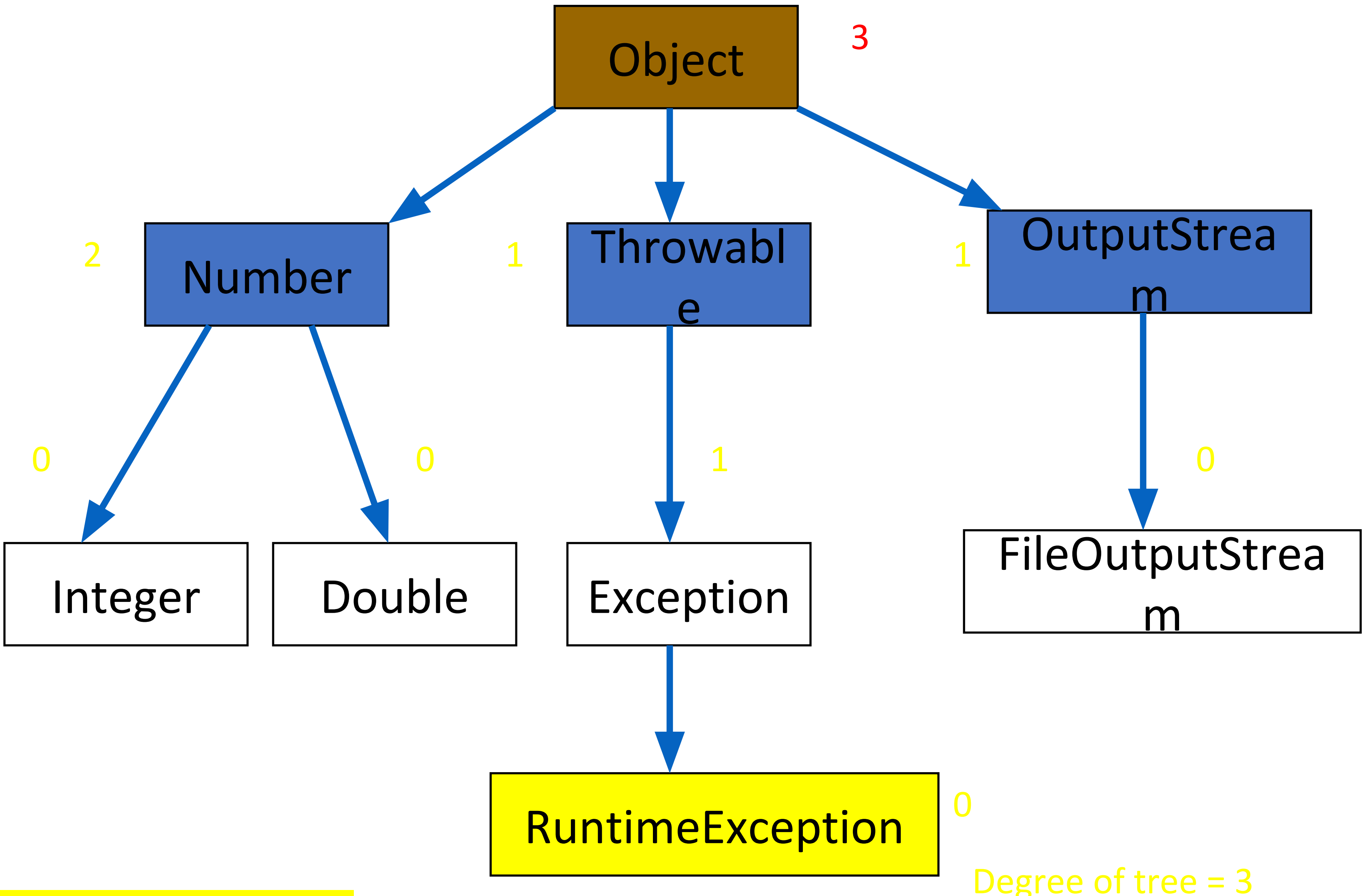


Examples:
Node Degree = Number Of Children



Examples:

Tree Degree = Max Node Degree

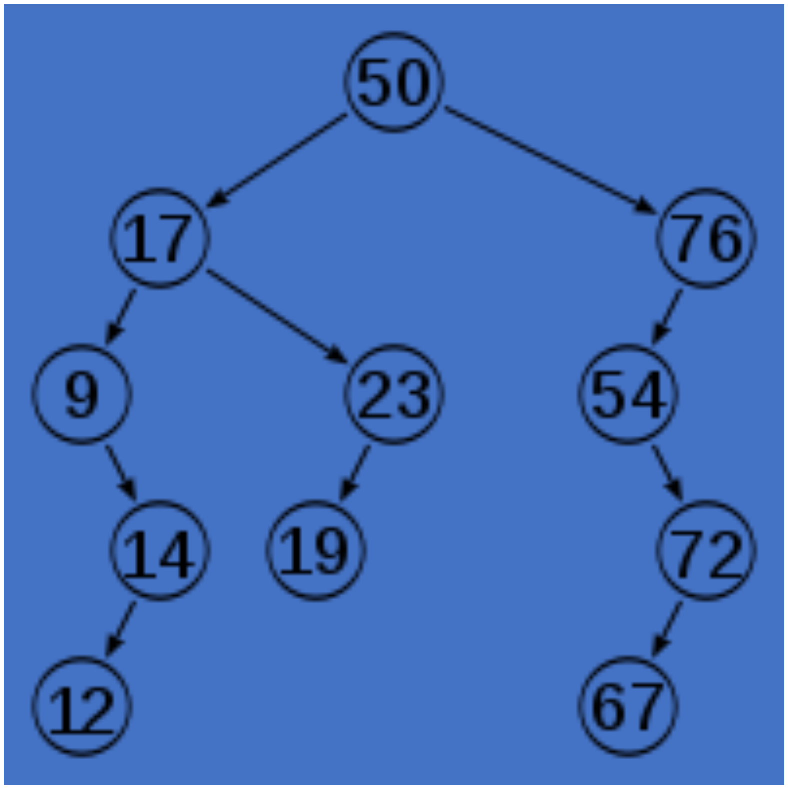


- Some text books start level numbers at 1 rather than at 0
- Root is at level 1
- Its children are at level 2
- The grand children of the root are at level 3
- And so on …

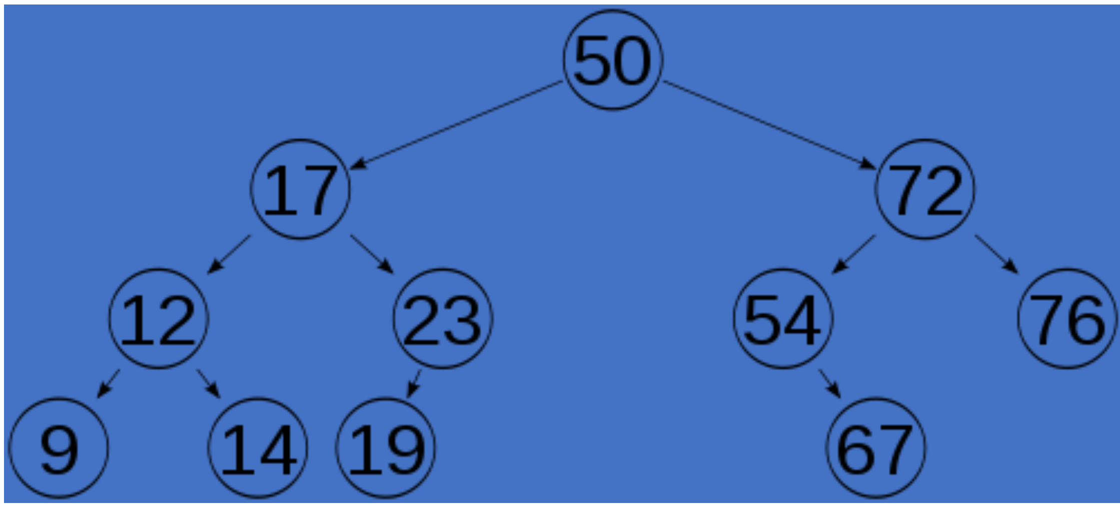
We shall number levels with the root at level 0

Balanced Tree

- **Balanced Tree:** a tree in which
  heights of subtrees are approximately equal



unbalanced tree

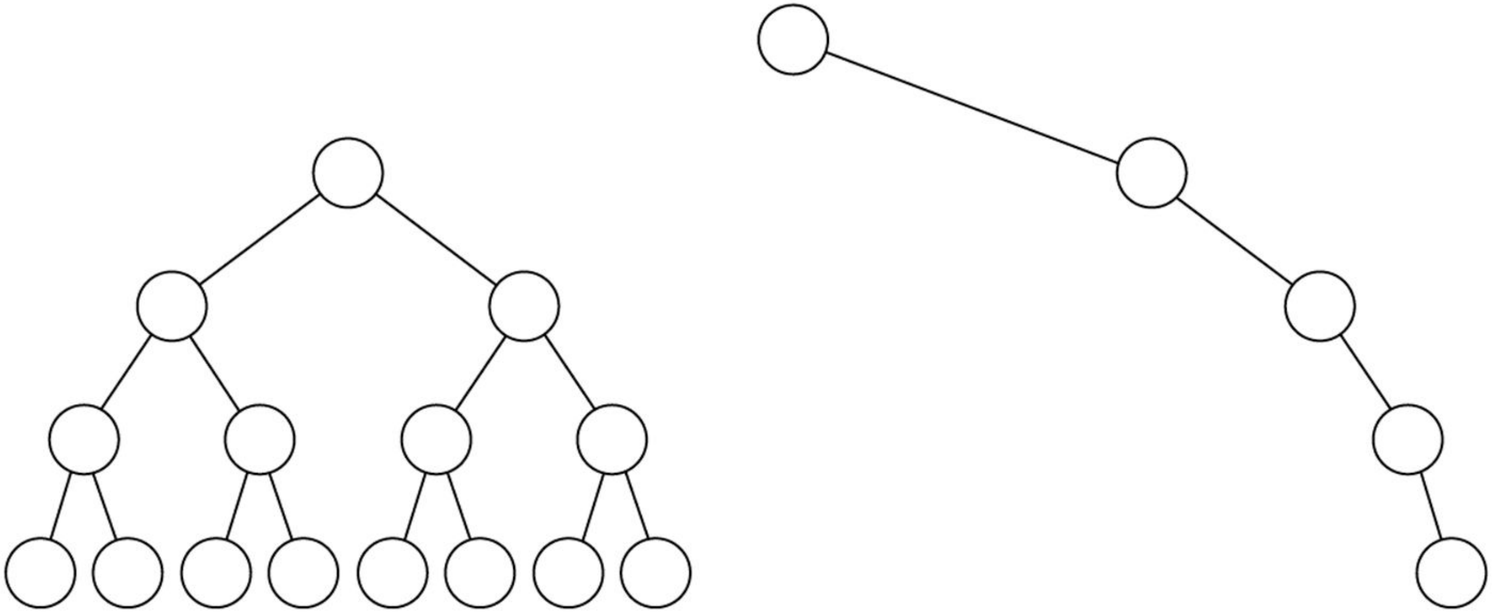


balanced tree

Tree Balance and Height

The balanced tree (a) has a height of: ___________

The unbalanced tree (b) has a height of: ___________



(a)

(b)

Types of Trees cont'd

- Regular trees are great for storing hierarchical data
- Their power can be heightened when we change how we store data in trees
- Rules and restrictions on:
  - What type of data can be stored
  - Where to store the data

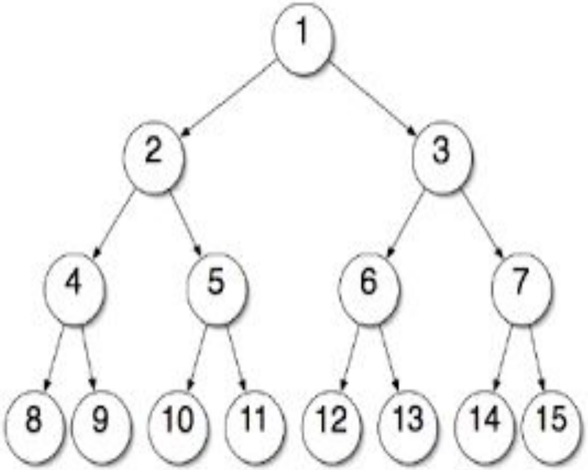
Types of Trees cont'd

- Types of Trees
  - Binary Trees (BT) (maximum two children per node)
  - Binary Search Trees (BST)
    (BT, Left child<=Node<=Right child, No same value)
  - AVL Trees
  - Red-Black Trees
  - Heap
  - N-ary Trees
  - …

Types of Trees:
Binary Tree
(BT)

Binary Tree

- is a tree data structure in which each node has at most two children, which are referred to as the *left child* and the *right child*
- is a finite set of nodes that is either
  - empty or
  - consists of a root and two disjoint binary trees called *the left subtree* and *the right subtree.*
- Any tree can be transformed into binary tree.
  - by left child-right sibling representation



Complete Binary Tree

Maximum Number of Nodes in BT

- The maximum number of nodes on level i of a binary tree is 2^{i-1}, $i>=1$

- The maximum nubmer of nodes in a binary tree of depth k is 2^k-1, $k>=1$

Proof by induction:

$$\sum_{i=1}^{k} 2^{i-1} = 2^k - 1$$

Binary Tree Properties

- Finite (possibly empty) collection of elements
- A nonempty binary tree has a root element
- The remaining elements (if any) are partitioned into two binary trees
- These are called the left subtree and right subtree of the binary tree

Differences Between Tree & Binary Tree

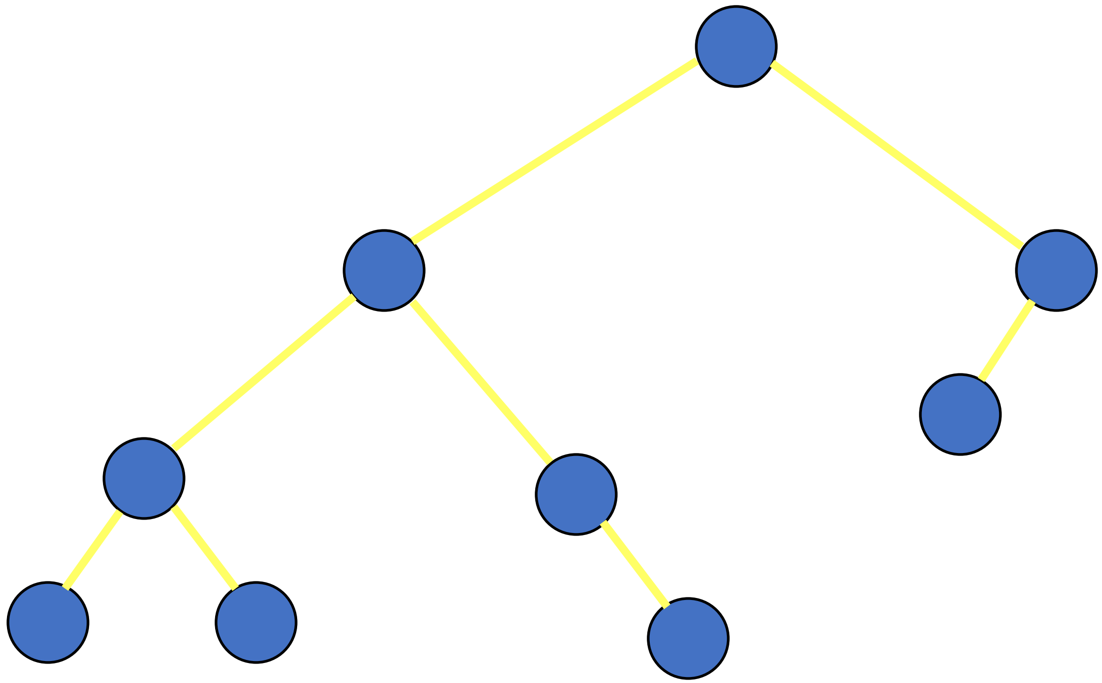
- No node in a binary tree may have a degree more than 2 (maximum 2 children), whereas there is no limit on the degree of a node in a tree
- A binary tree may be empty, whereas a tree cannot be empty

Differences Between Tree & Binary Tree

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

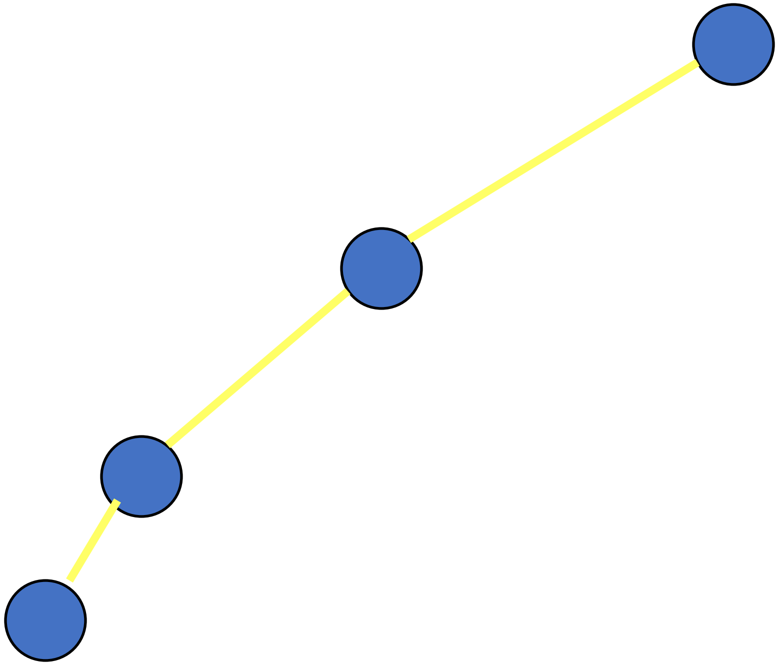
- Are different when viewed as binary trees
- Are the same when viewed as trees

Binary Tree Properties & Representation



Minimum Number Of Nodes

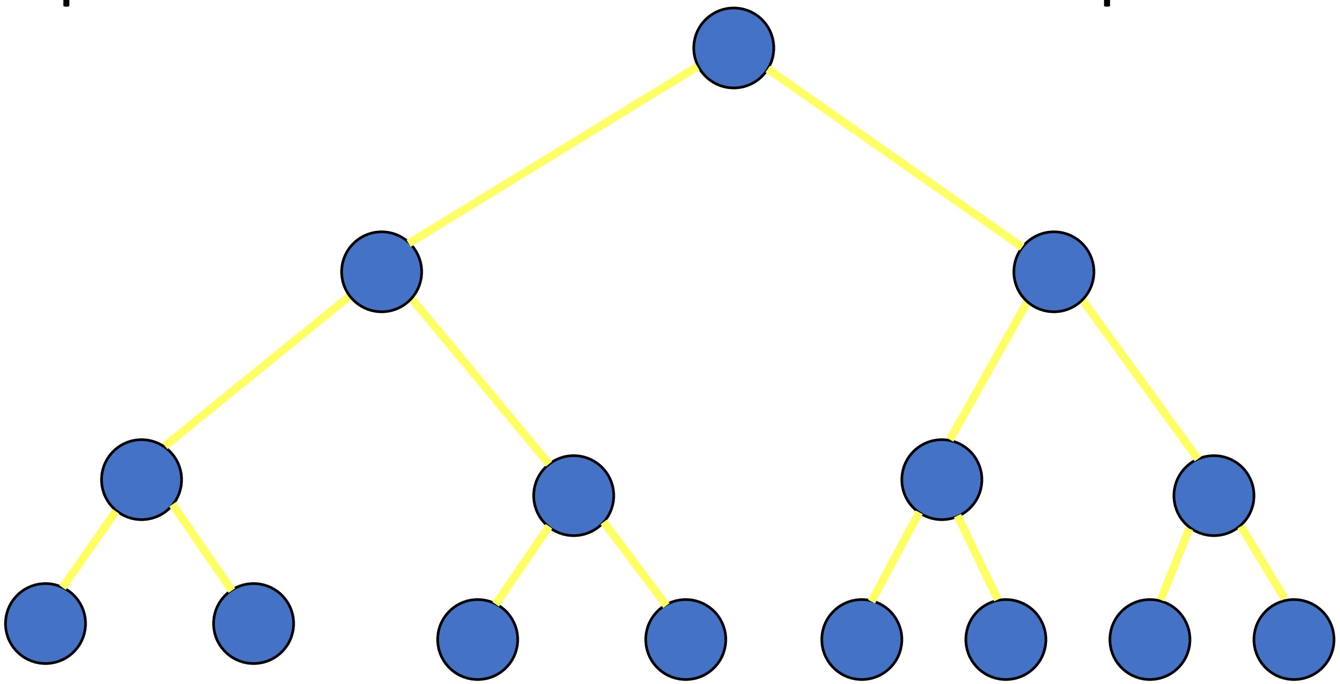
- Minimum number of nodes in a binary tree whose height is h
- At least one node at each of first h levels



minimum number of nodes is h

Maximum Number Of Nodes

- All possible nodes at first h levels are present.



Maximum number of nodes

$= 1 + 2 + 4 + 8 + \ldots + 2^{h-1}$

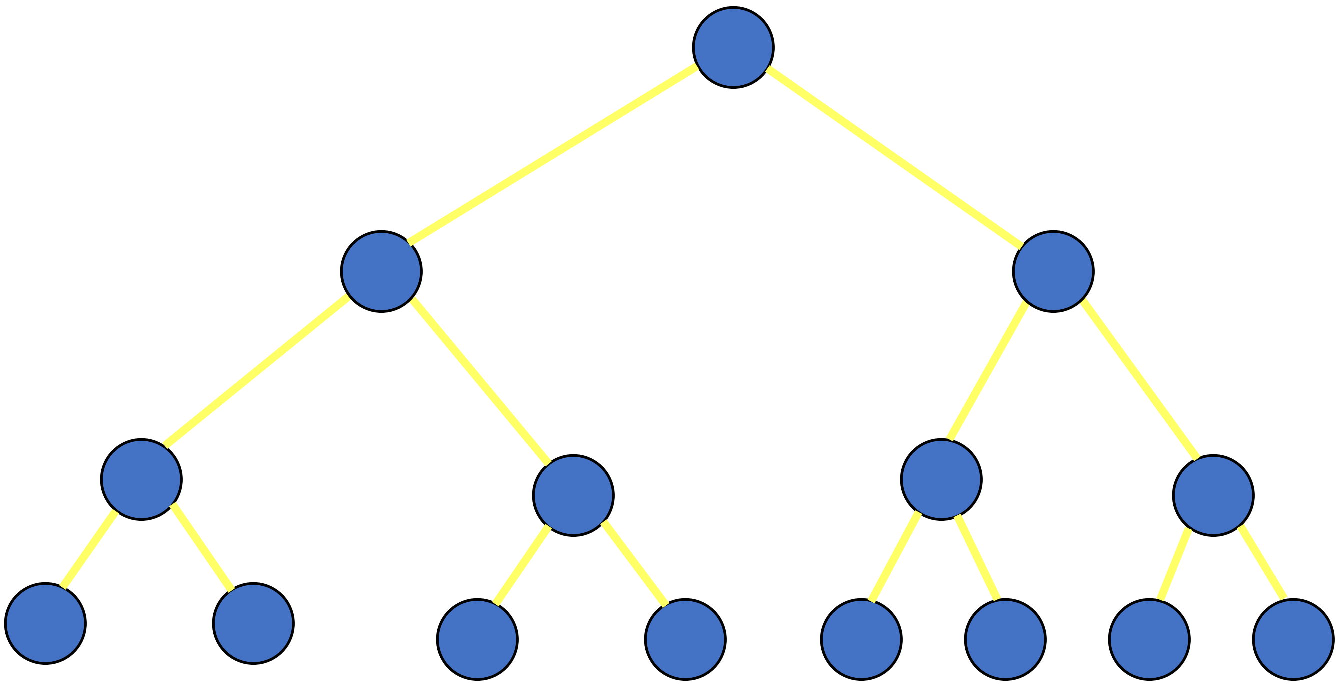
$= 2^h - 1$

Number Of Nodes & Height

- Let n be the number of nodes in a binary tree whose height is h
- $h <= n <= 2^h - 1$
- $\log_2(n+1) <= h <= n$

Full Binary Tree

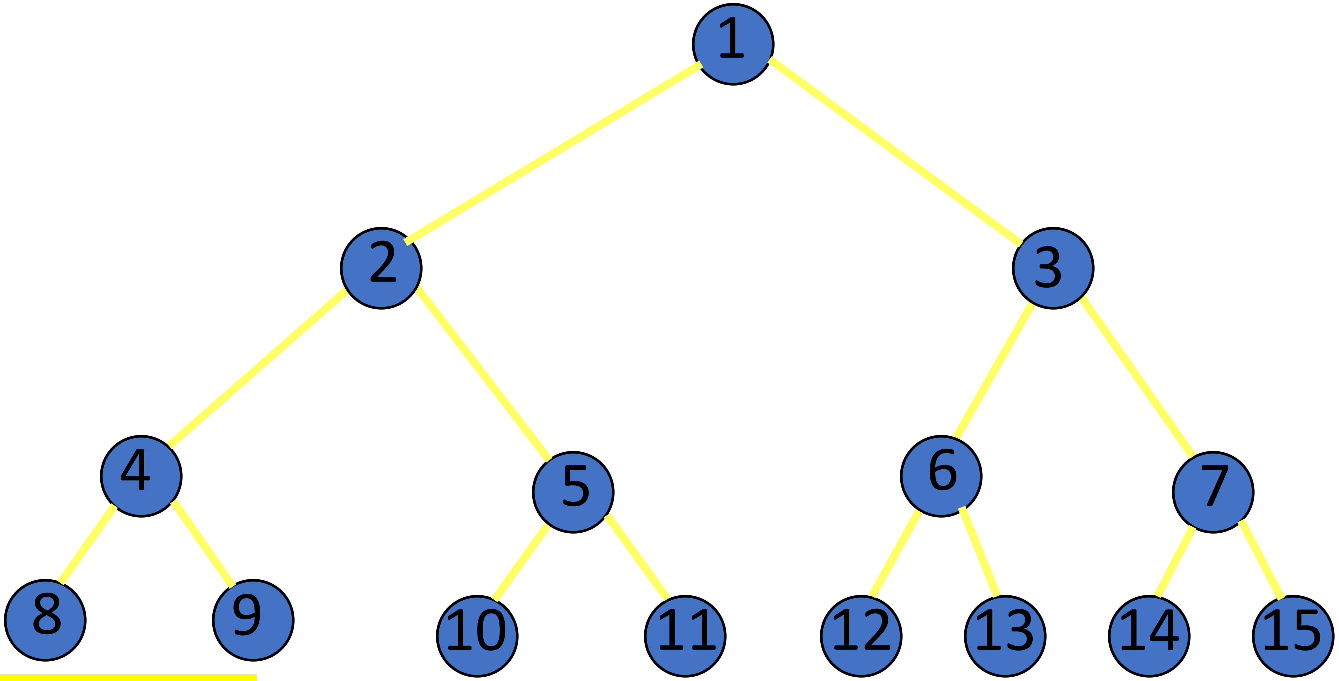
- A full binary tree of a given height h has $2^h - 1$ nodes



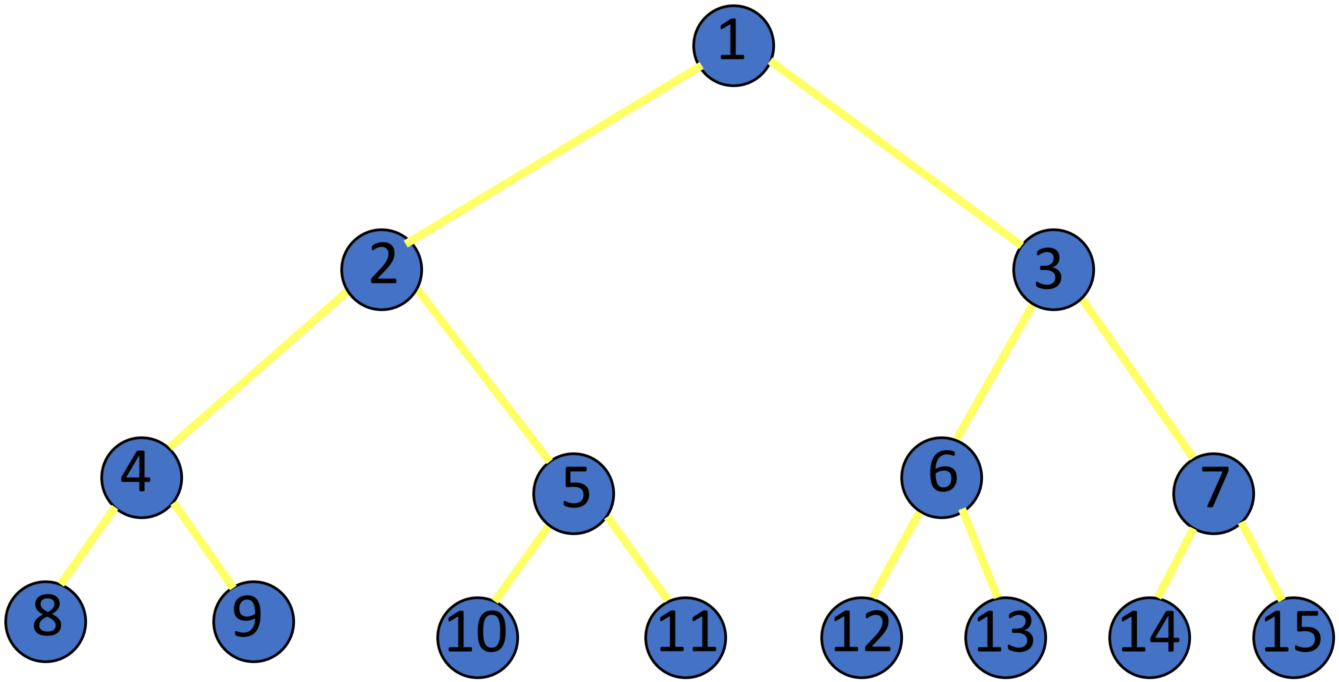
Height 4 full binary tree.

Numbering Nodes In A Full Binary Tree

- Number the nodes 1 through $2^h - 1$
- Number by levels from top to bottom
- Within a level number from left to right

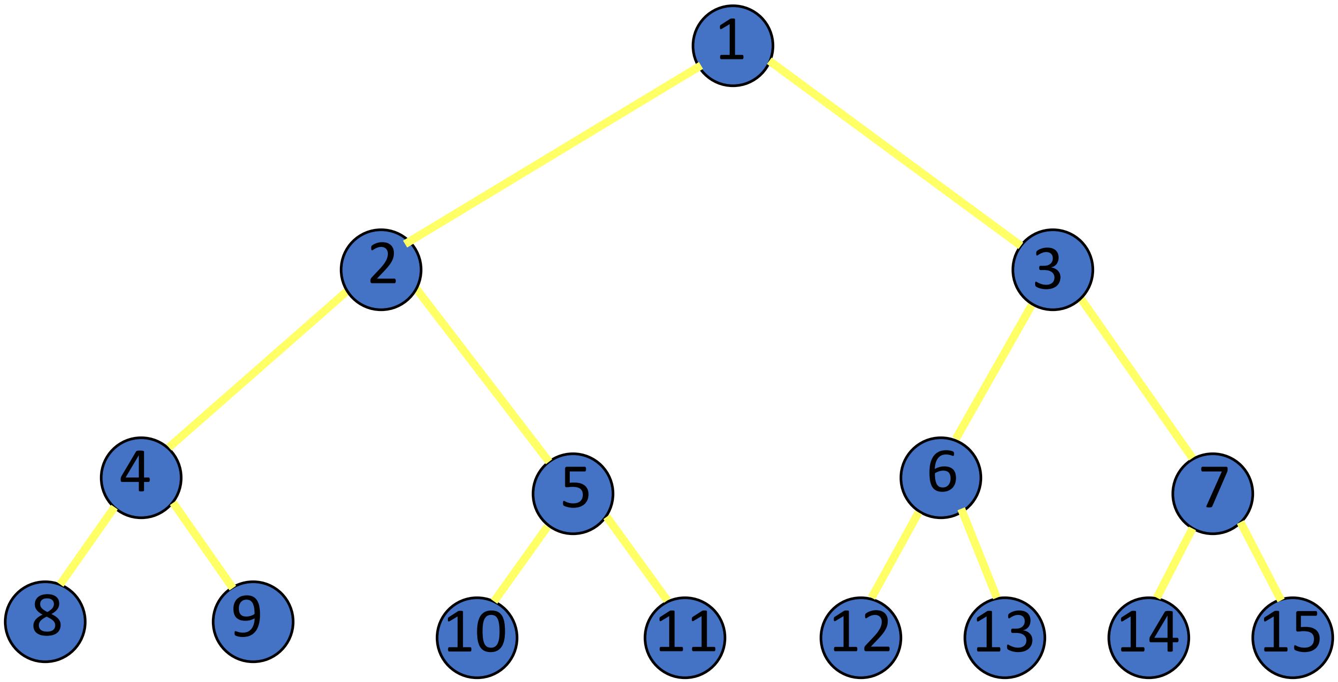


Node Number Properties



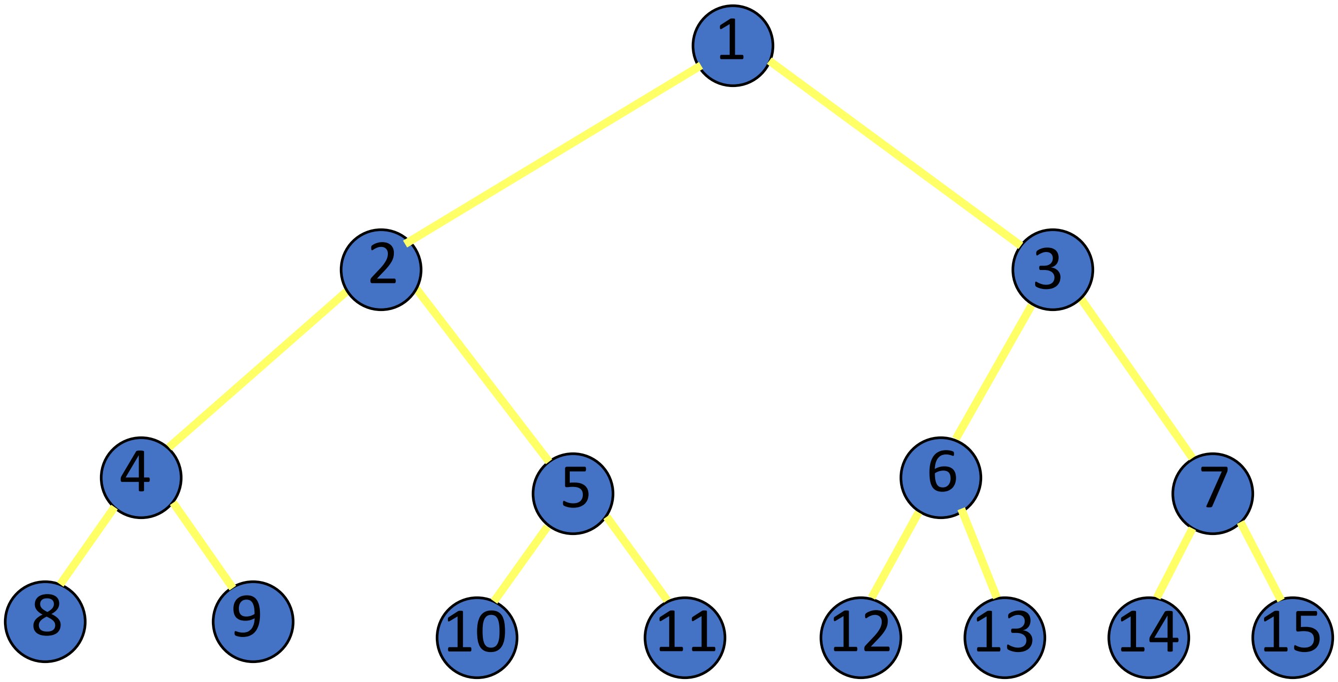
- Parent of node i is node i / 2, unless i = 1
- Node 1 is the root and has no parent

Node Number Properties



- **Left child** of node i is node $2*i$, unless $2*i > n$, where n is the number of nodes
- If $2*i > n$, node i has no left child

Node Number Properties



- **Right child** of node i is node 2*i+1, unless 2*i+1 > n, where n is the number of nodes
- If 2*i+1 > n, node i has no right child

Complete Binary Tree With n Nodes

- Start with a full binary tree that has at least n nodes

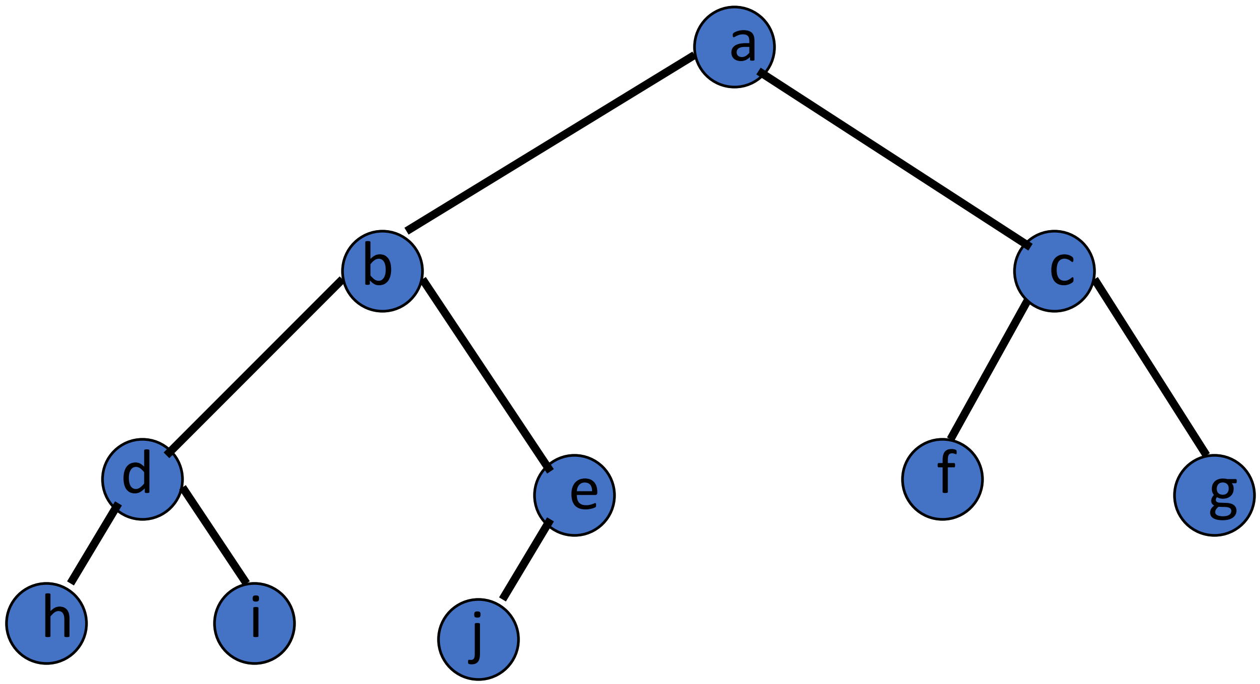
- Number the nodes as described earlier
- The binary tree defined by the nodes numbered 1 through n is the unique n node complete binary tree

Binary Tree Representation

- Array representation
- Linked-List representation

1. Array Representation

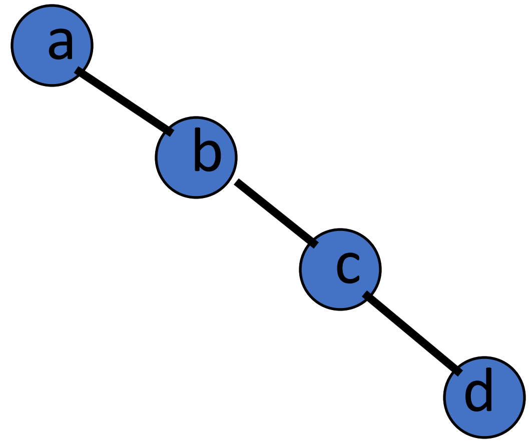
- Number the nodes using the numbering scheme for a full binary tree. The node that is numbered i is stored in tree[i].



tree[]

	a	b	c	d	e	f	g	h	i	j

Right-Skewed Binary Tree



tree[]

	a	-	b	-	-	-	c	-	-	-	-	-	-	-	d

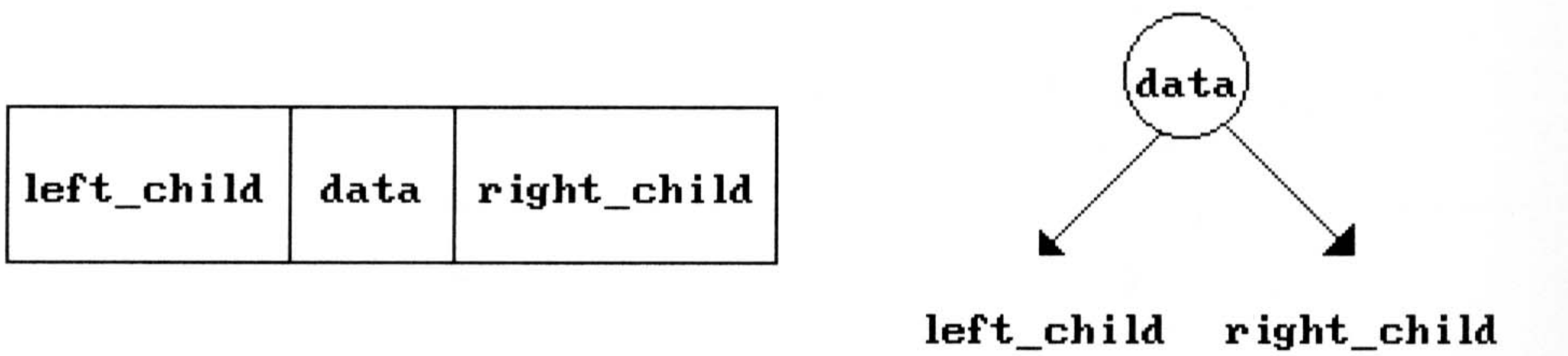
- An n node binary tree needs an array whose length is between $n+1$ and 2^n.

2. Linked-List Representation

- Each binary tree node is represented as an object whose data type is
- The space required by an   node binary tree is

Binary Tree Representations (using link)

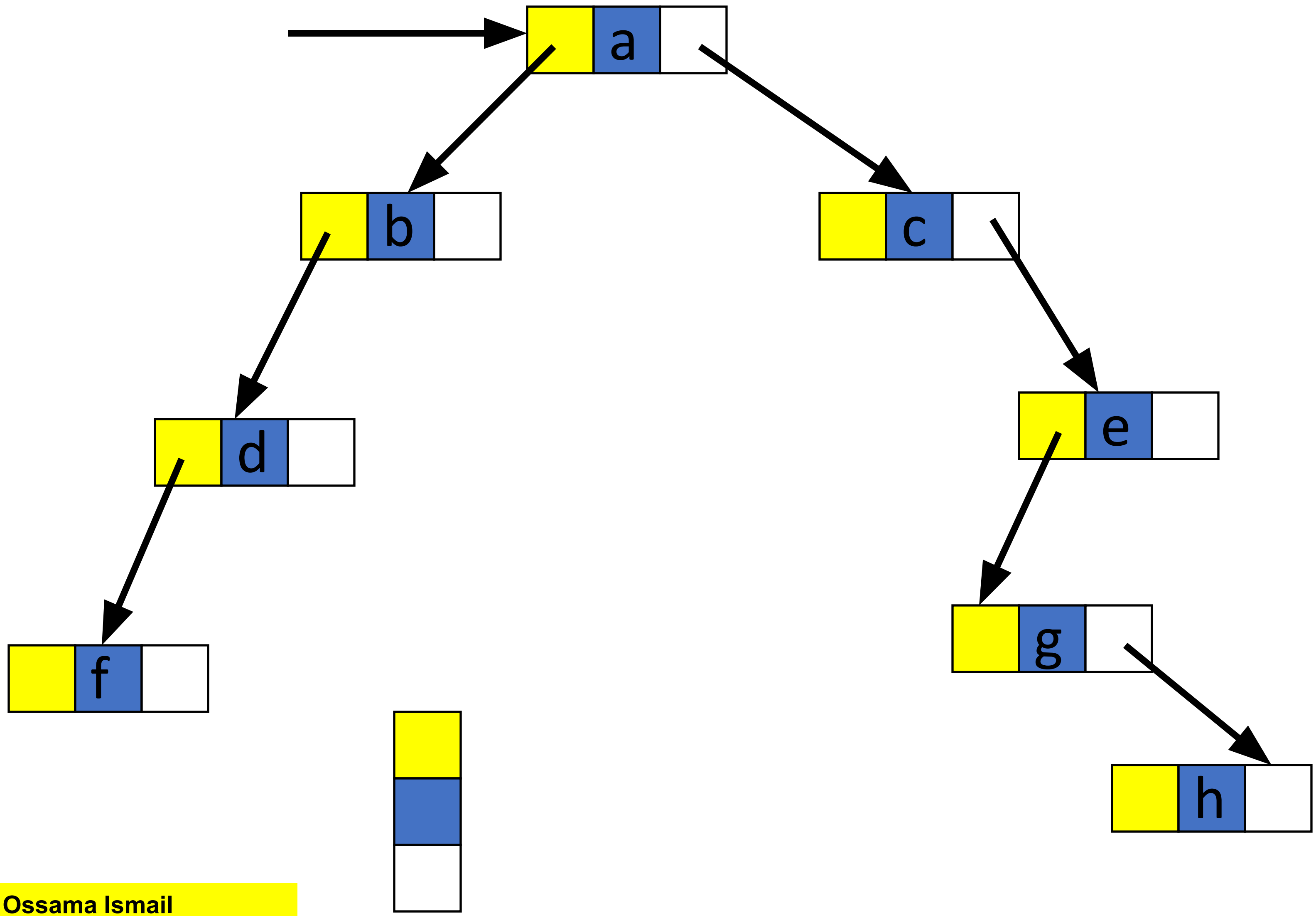
```
typedef struct node *tree_pointer;
typedef struct node {
        int data;
        tree_pointer left_child, right_child;
        };
```



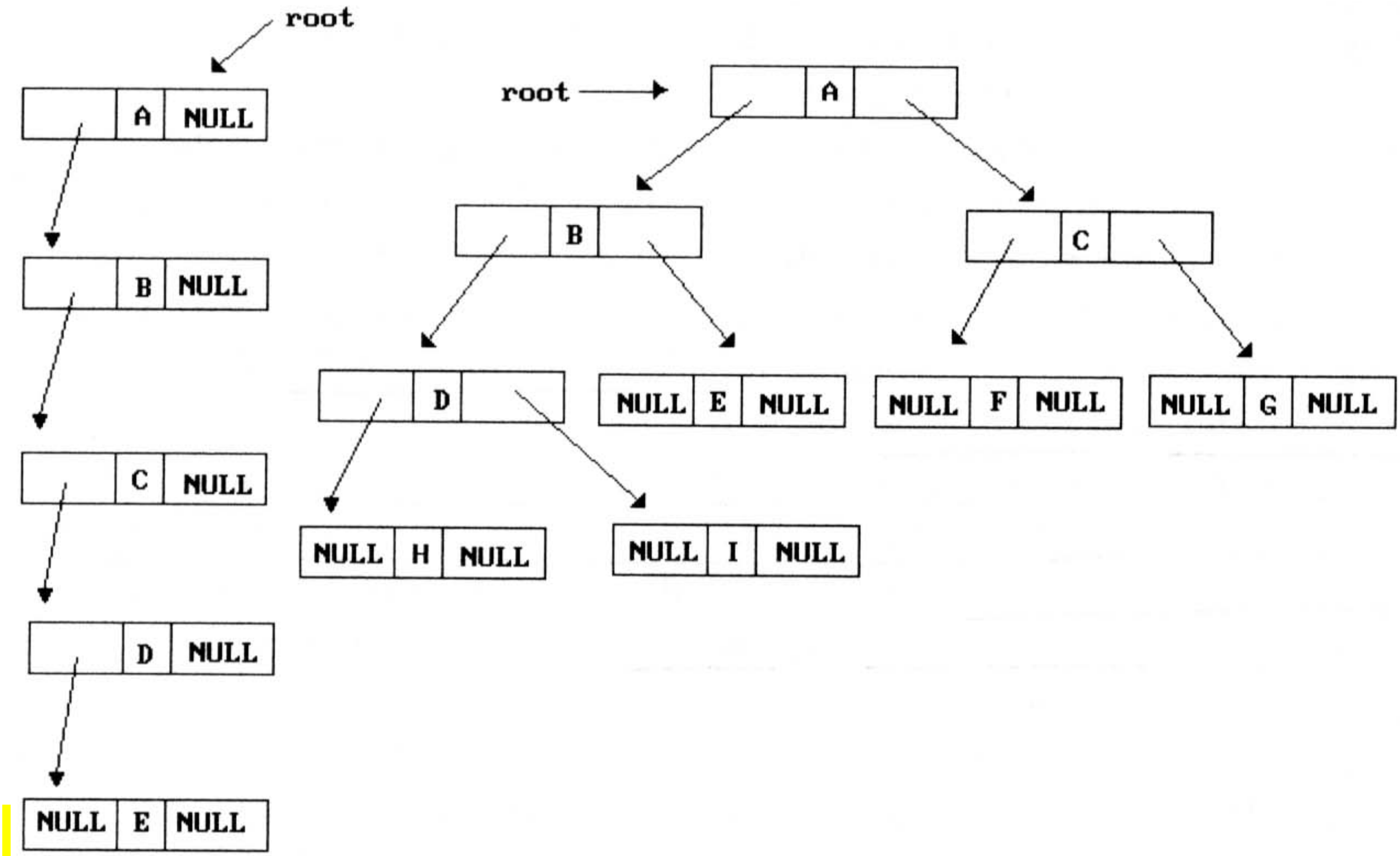
The Class BinaryTreeNode

```
          dataStructures;
              BinaryTreeNode
{
  Object element;
  BinaryTreeNode leftChild; // left subtree
  BinaryTreeNode rightChild; // right subtree
  // constructors and any other methods
  // come here
}
```

Linked Representation Example



Binary Tree Representations (using link)



Binary Tree Operations

Some Binary Tree Operations

- Determine the height
- Determine the number of nodes
- Make a clone
- Determine if two binary trees are clones
- Display the binary tree
- Evaluate the arithmetic expression represented by a binary tree
  - Obtain the infix form of an expression
  - Obtain the prefix form of an expression
  - Obtain the postfix form of an expression

Programming with Binary Trees

- Many tree algorithms are **recursive**
  - Process current node, recurse on subtrees
  - Base case is usually empty tree (`null`)

- Traversal: An examination of the elements of a tree
  - A pattern used in many tree algorithms and methods

- Common orderings for traversals:
  - **pre-order**: process root node, then its left/right subtrees
  - **in-order**: process left subtree, then root node, then right
  - **post-order**: process left/right subtrees, then root node

Binary Tree Traversal

Binary Tree Traversal

- Many binary tree operations are done by performing a traversal of the binary tree
- In a traversal, each element of the binary tree is visited exactly once
- During the visit of an element, all action (make a clone, display, evaluate the operator, etc…) with respect to this element is taken

Binary Tree Traversal

- A traversal is where each node in a tree is visited and visited once
- For a tree of n nodes there are n! traversals
- There are two very common traversals
  - Breadth First
  - Depth First

A - Breadth First

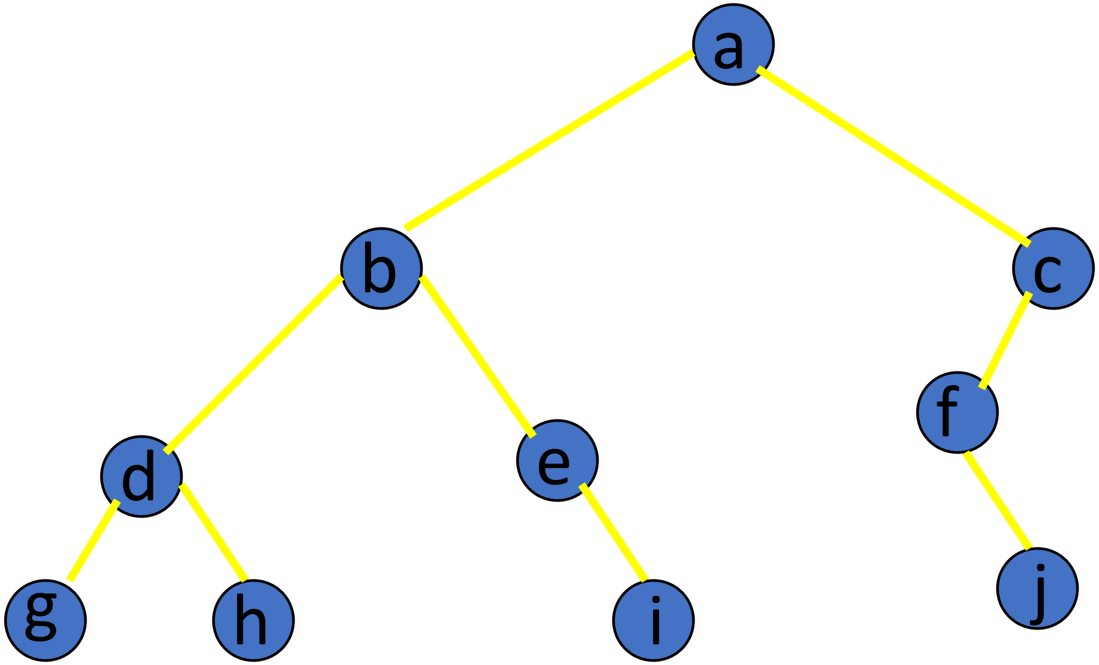
- In a breadth first traversal all of the nodes on a given level are visited and then all of the nodes on the next level are visited
- Usually in a left to right fashion
- This is implemented with a queue
- Sometimes called Level Order

Level Order

Let t be the tree root

```
(t != null)
{
   visit t and put its children on a FIFO queue;
   remove a node from the FIFO queue and call it t;
   // remove returns null when queue is empty
}
```

Level-Order Example (visit = print)



a b c d e f g h i j

Time complexity

– Let *n* be the number of nodes in the tree

– Time complexity: O(*n*)

– Space complexity: O(*n*)

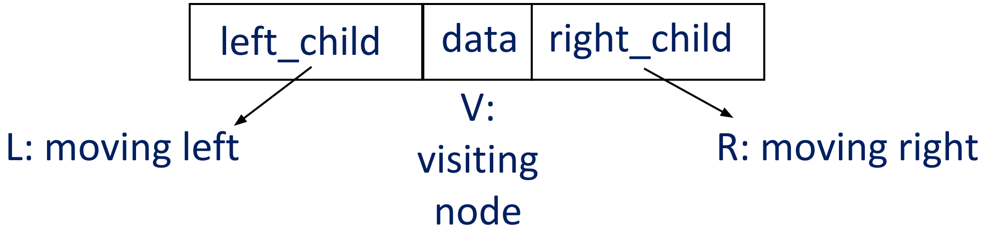
equal to the depth of the tree
(skewed tree is the worst case)

B - Depth First

- In a depth first traversal all the nodes on a branch are visited before any others are visited
- There are three common depth first traversals
  - Inorder
  - Preorder
  - Postorder
- Each type has its use and specific application

Binary Tree Traversals

- How to traverse a tree or visit each node in the tree exactly once?
- Let L, V, and R stand for moving left, visiting the node and moving right.
- There are six possible combinations of traversal
  - LVR, LRV, VLR, VRL, RVL, RLV
- Adopt convention that we traverse left before right, only 3 traversals remain : LVR, LRV, VLR (inorder, postorder, preorder)
  - LVR (inorder), LRV (postorder), VLR (preorder)



Binary Tree Traversals

Depth-First traversals

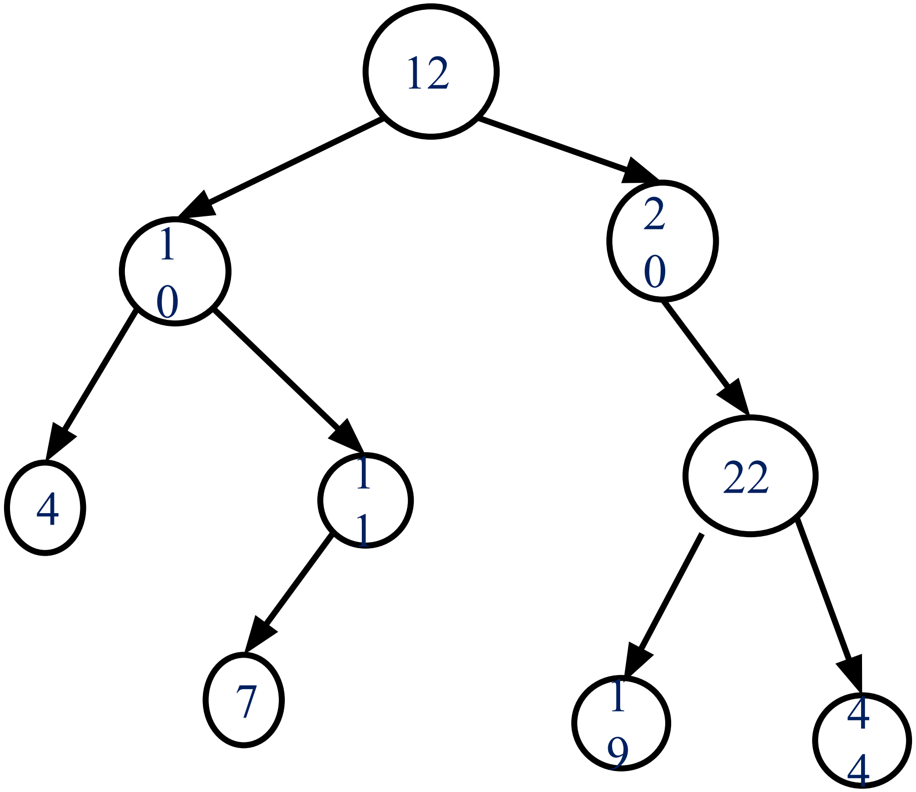
- Pre-order
- In-order
- Post-order

Apply “**In Order Traversal**” to the given tree

1- visit left subtree

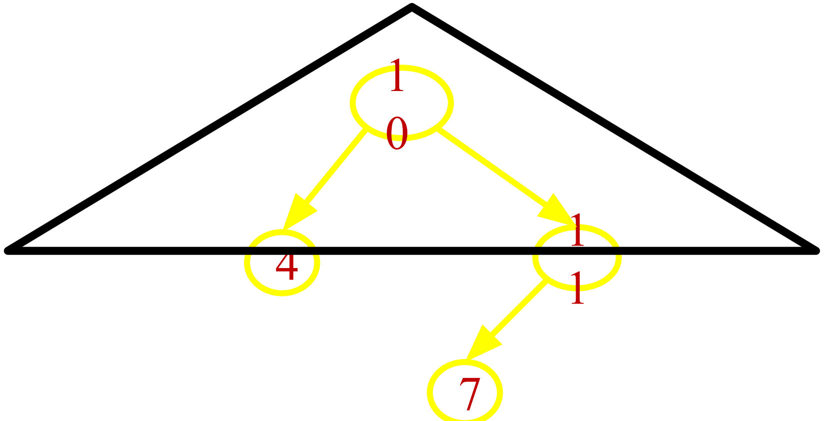
2- visit node

3- visit right subtree

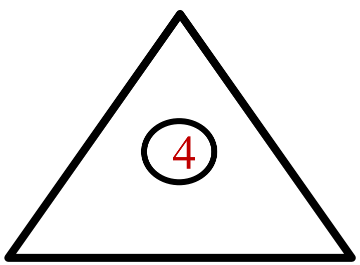


In Order Traversal

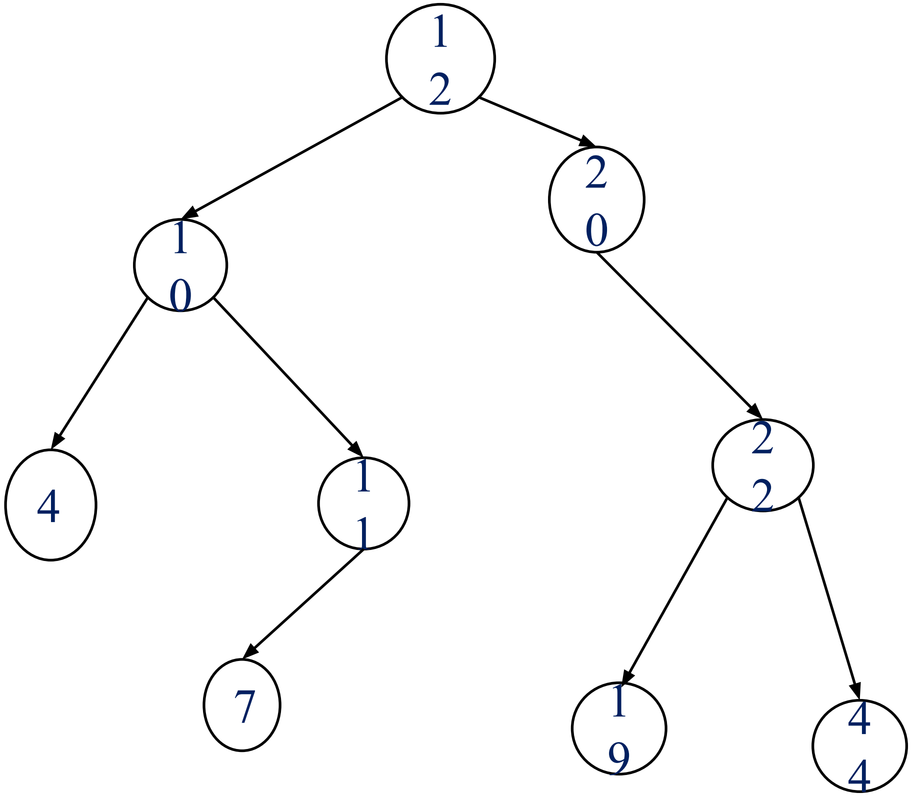
(1) goto subtree of node 12



(2) goto left subtree of node 10



(3) goto left subtree of node 4

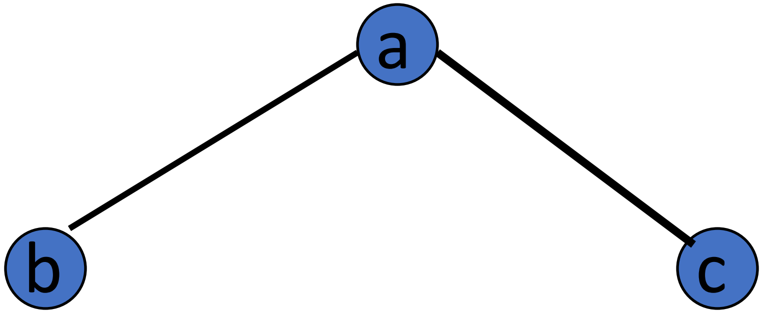


Result of applying “In order traversal”:
4→10→7 -> 11→12→20→19→22→44

Inorder Traversal

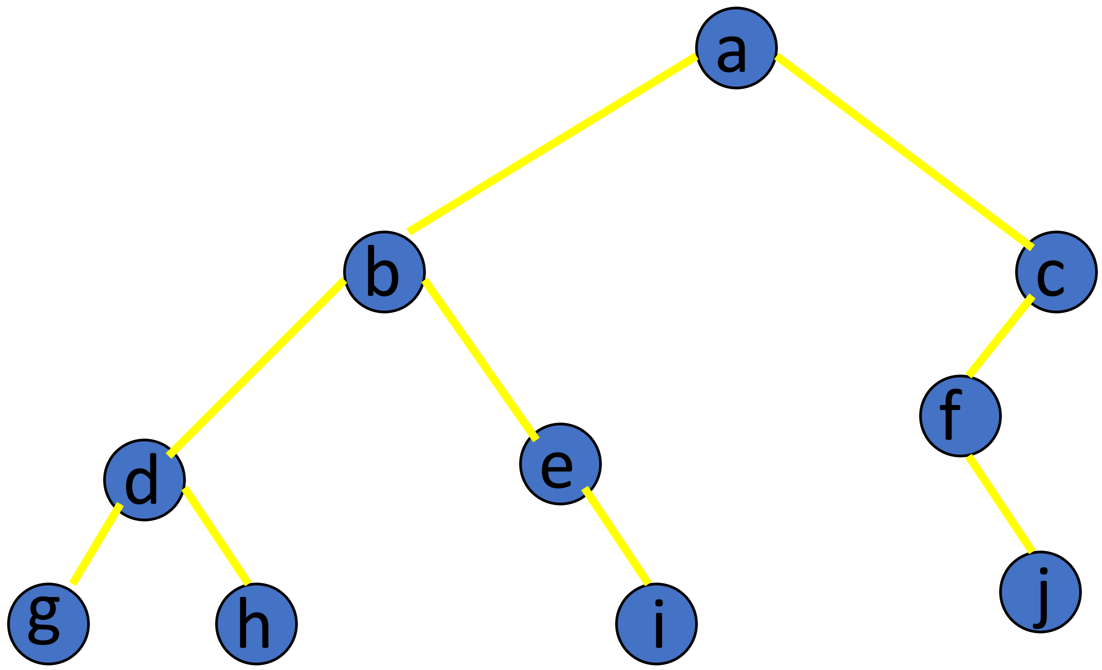
```
public static void inOrder(BinaryTreeNode t)
{
    if (t != null)
    {
      inOrder(t.leftChild);
      visit(t);
      inOrder(t.rightChild);
    }
}
```

Inorder Example (visit = print)



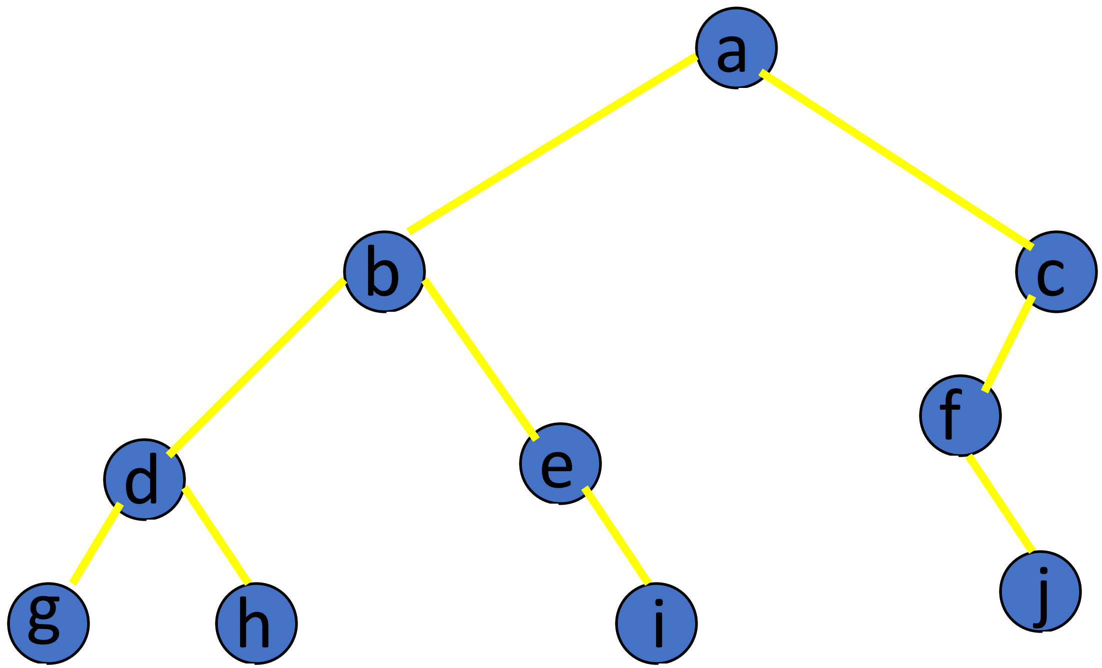
b a c

Inorder Example (visit = print)



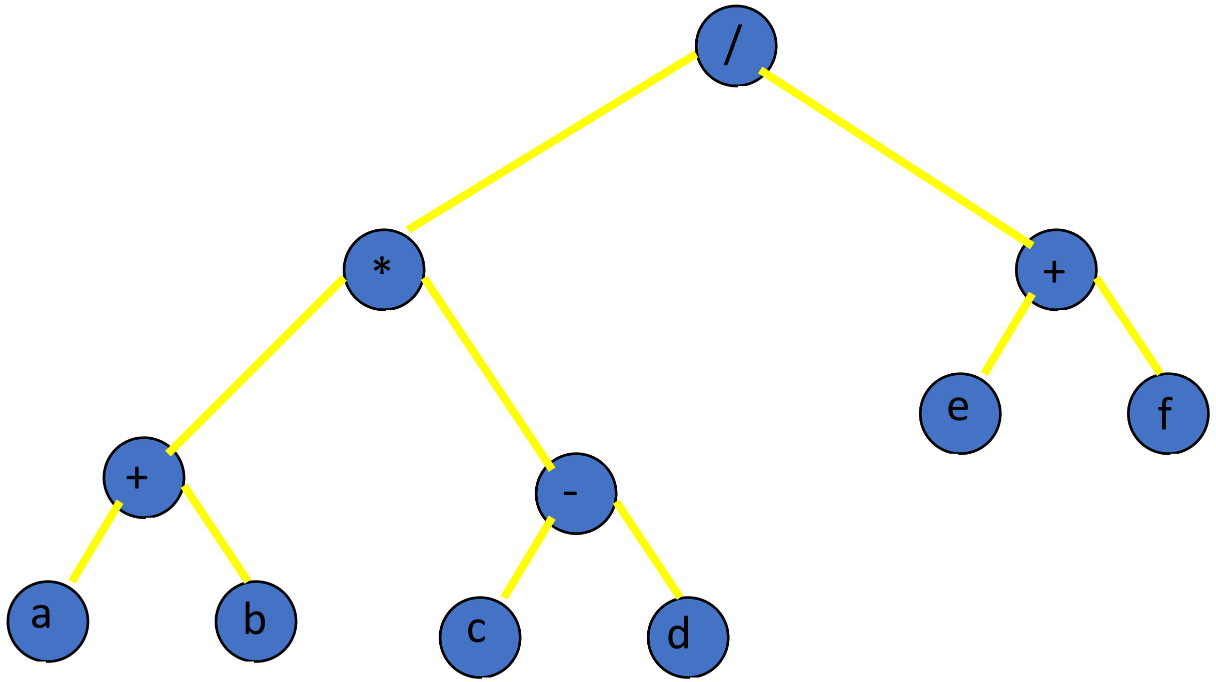
g d h b e i a f j c

Inorder By Projection (Squishing)



g d h b e i a f j c

Inorder Of Expression Tree



a + b * c - d / e + f

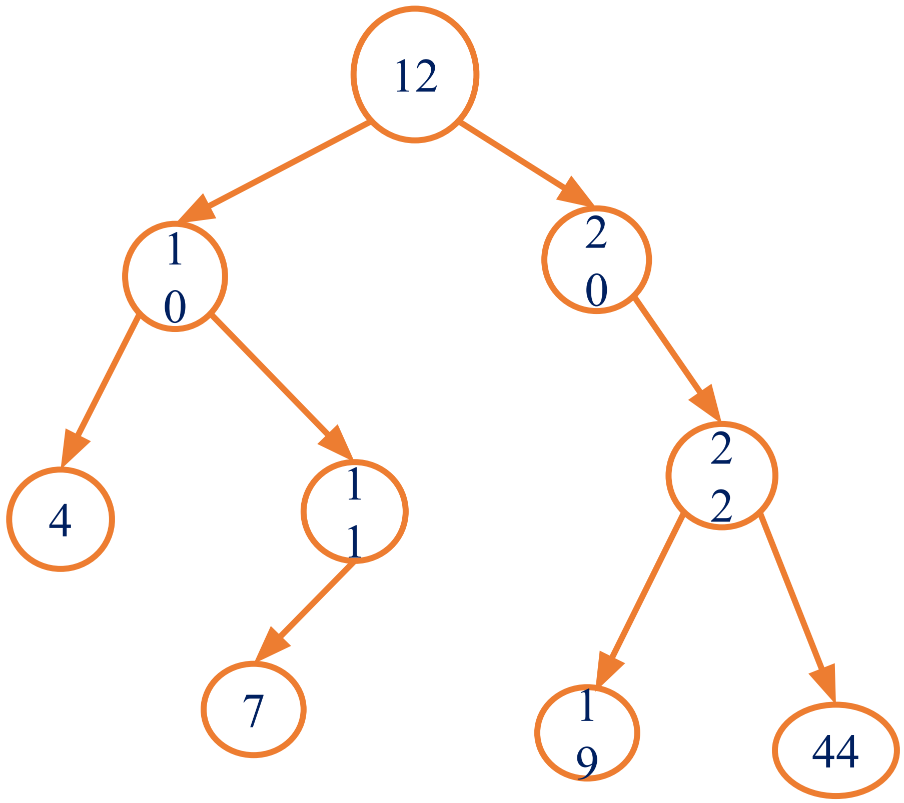
Gives infix form of expression (sans parentheses)!

Apply “**Post Order Traversal**” to the given tree

1- visit left subtree

2- visit right subtree

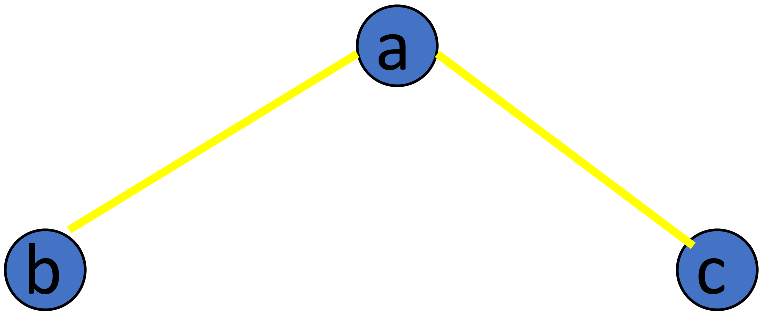
3- visit node



Postorder Traversal

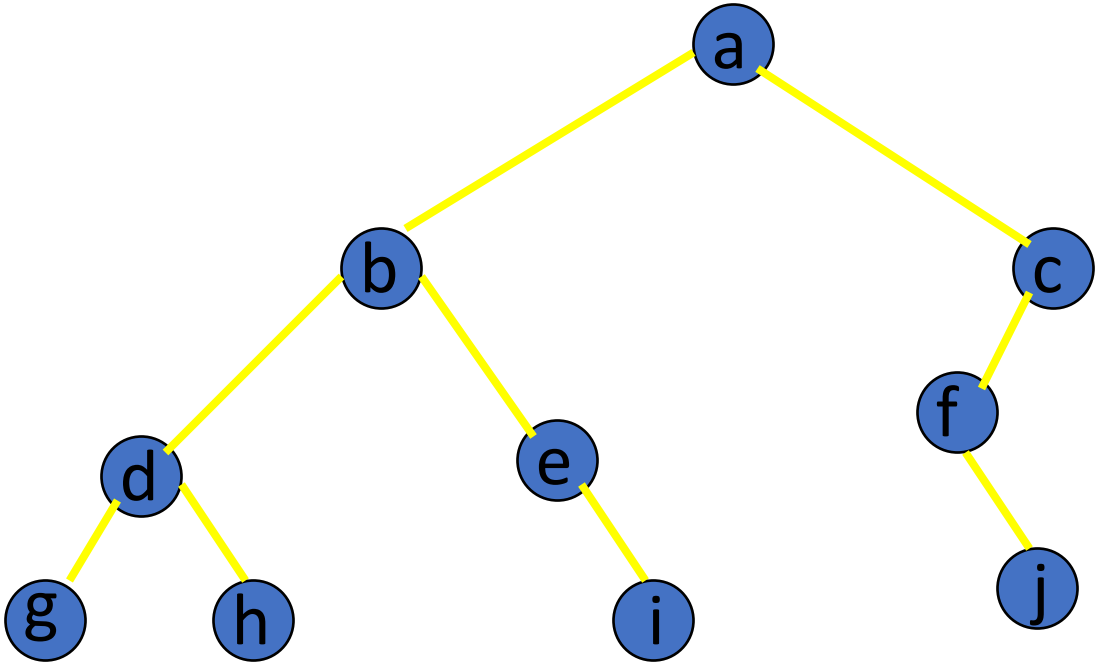
```
public static void postOrder(BinaryTreeNode t)
{
    if (t != null)
    {
      postOrder(t.leftChild);
      postOrder(t.rightChild);
      visit(t);
    }
}
```

Postorder Example (visit = print)



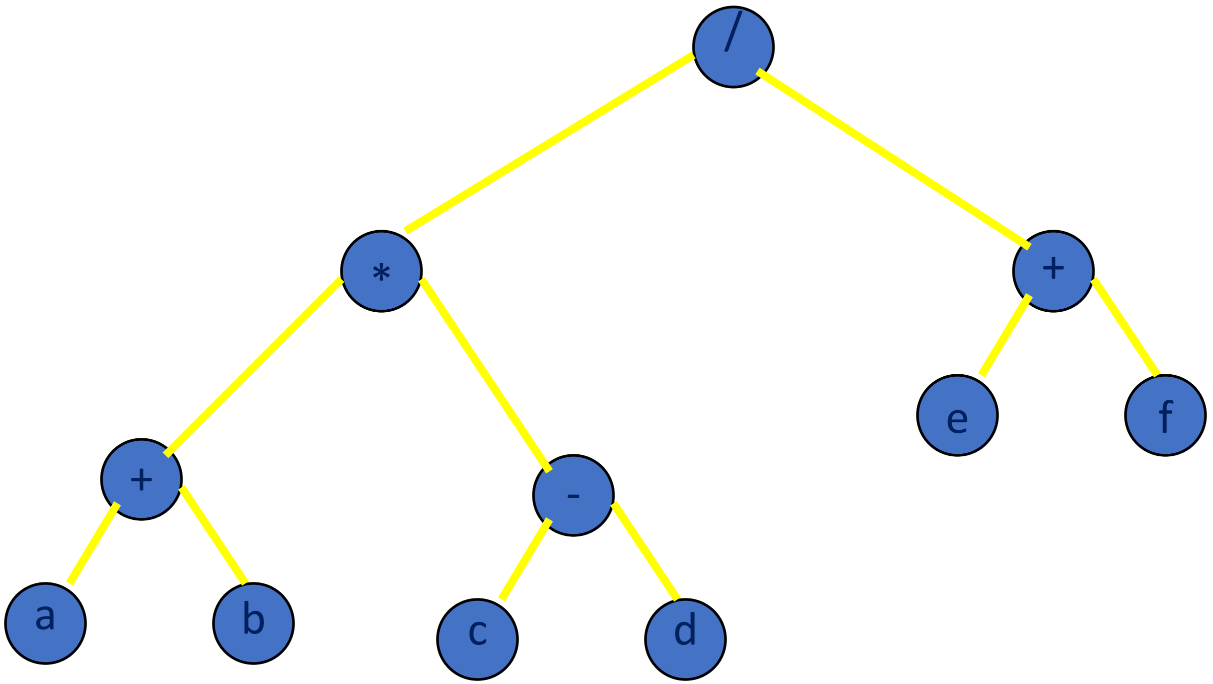
b c a

Postorder Example (visit = print)



g h d i e b j f c a

Postorder Of Expression Tree



a b + c d - * e f + /

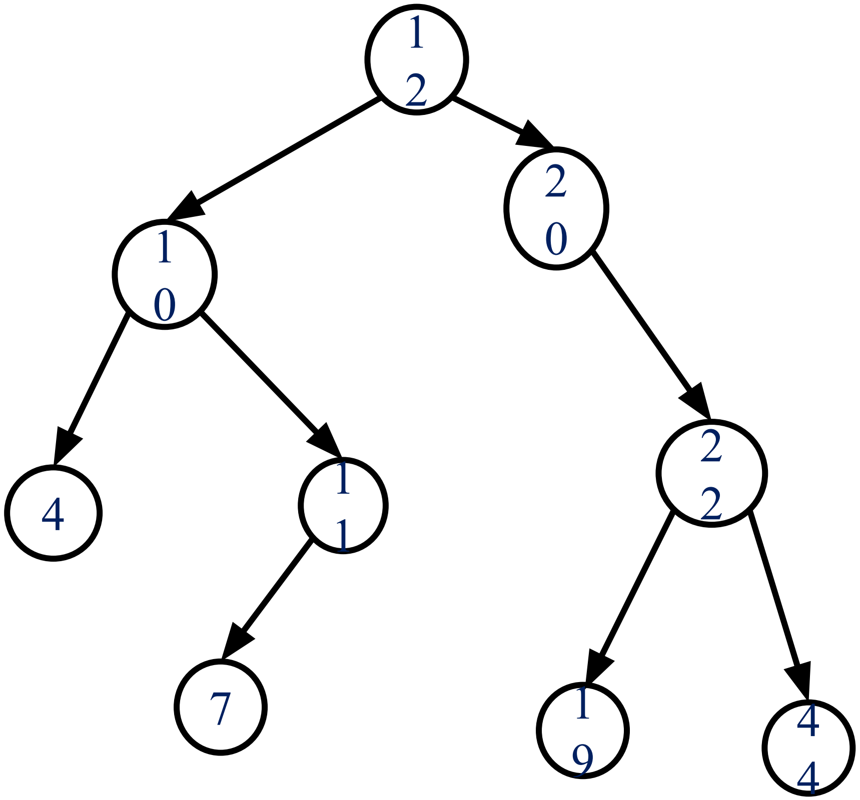
Gives postfix form of expression!

Apply “**Pre Order Traversal**” to the given tree

1- visit node

2- visit left subtree

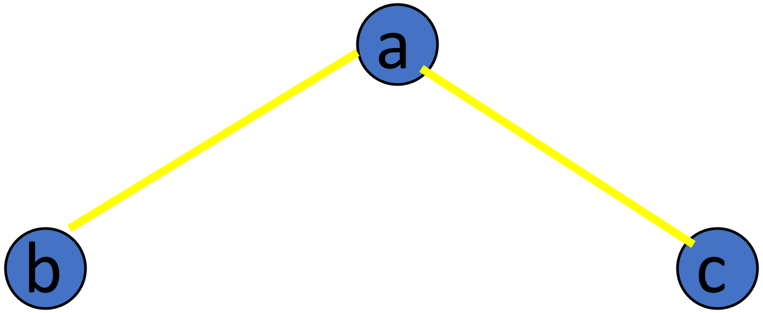
3- visit right subtree



Preorder Traversal

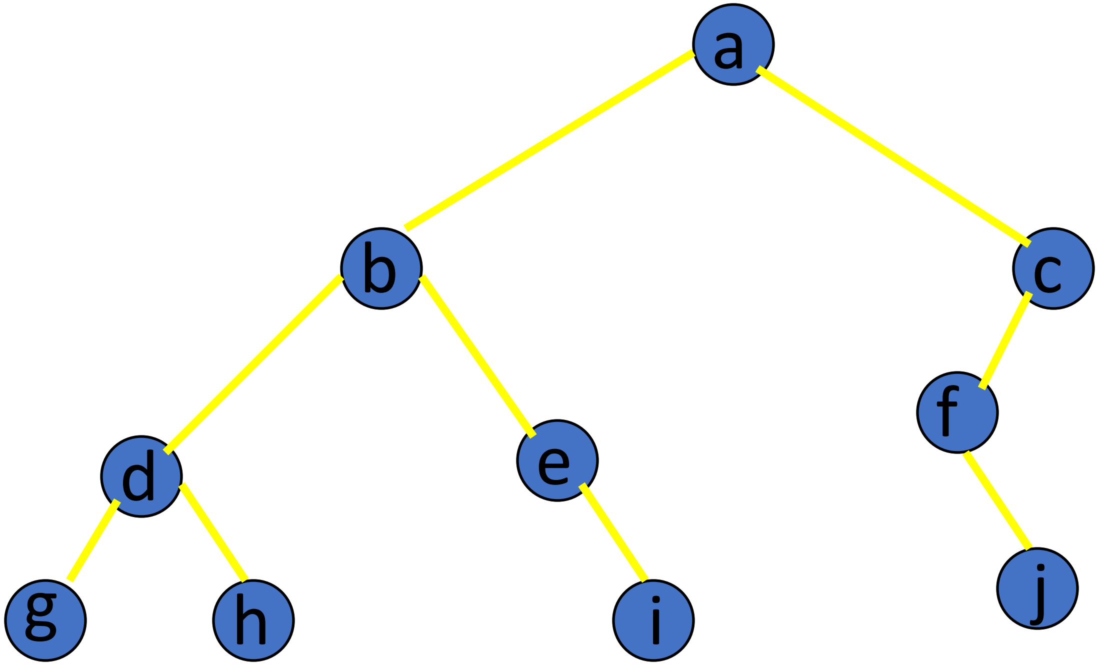
```
public static void preOrder(BinaryTreeNode t)
{
    if (t != null)
    {
      visit(t);
      preOrder(t.leftChild);
      preOrder(t.rightChild);
    }
}
```

Preorder Example (visit = print)



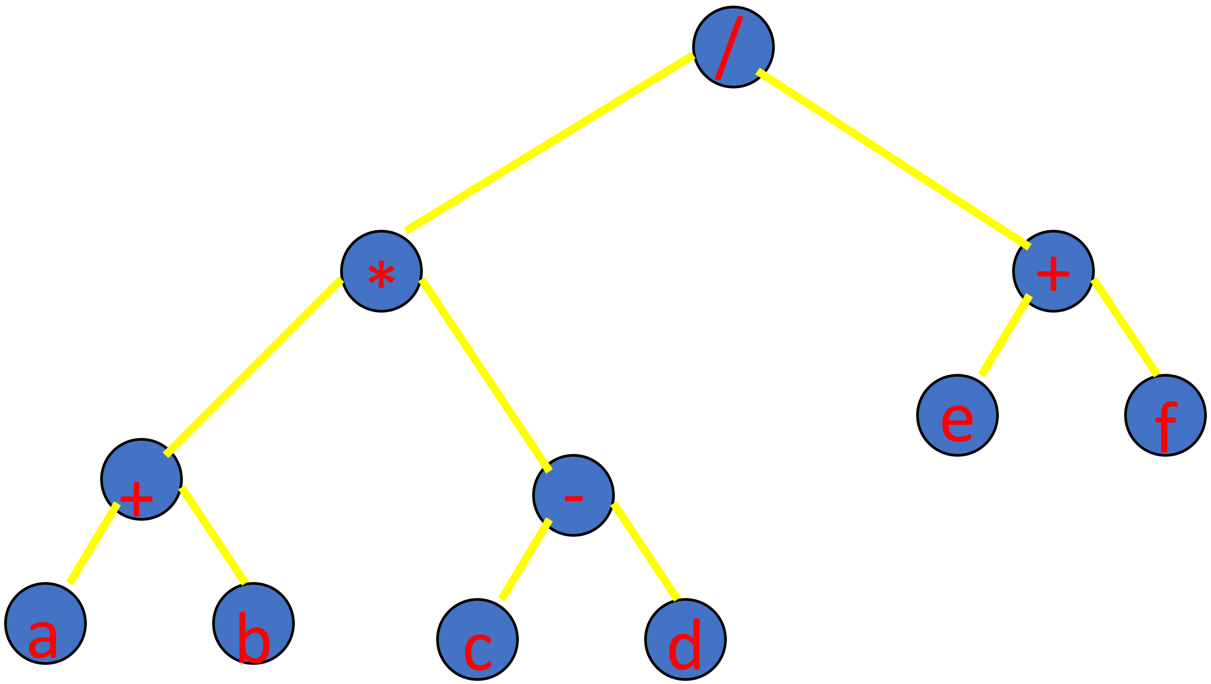
a b c

Preorder Example (visit = print)



a b d g h e i c f j

Preorder Of Expression Tree



/ * + a b - c d + e f

Binary Tree Construction

Binary Tree Construction

- Suppose that the elements in a binary tree are distinct
- Can you construct the binary tree from which a given traversal sequence came?
- When a traversal sequence has more than one element, the binary tree is not uniquely defined
- Therefore, the tree from which the sequence was obtained cannot be reconstructed uniquely

Some Examples

Preorder

= ab

Inorder

= ab

Postorder

= ab

Level order

= ab

Binary Tree Construction

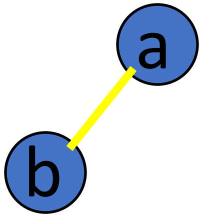
- Can you construct the binary tree, given two traversal sequences?

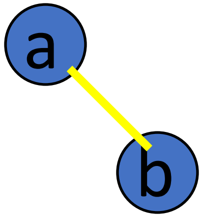
- Depends on which two sequences are given

Preorder And Postorder

preorder = ab

postorder = ba

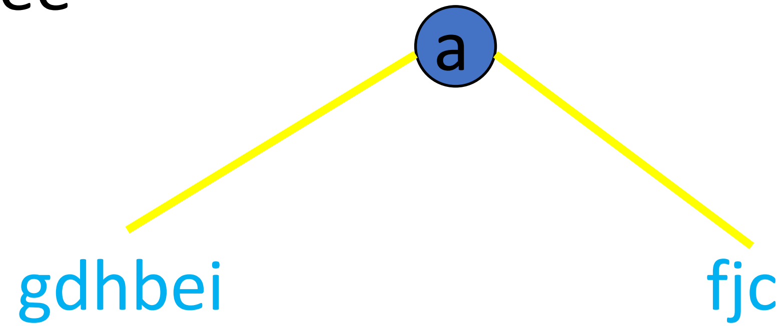




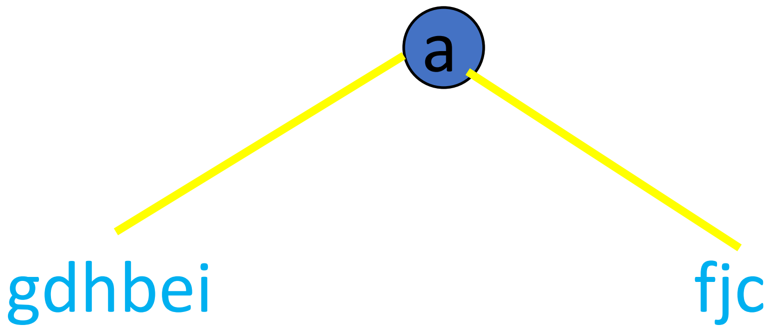
- Preorder and postorder do not uniquely define a binary tree.
- Nor do preorder and level order (same example)
- Nor do postorder and level order (same example)

Inorder And Preorder

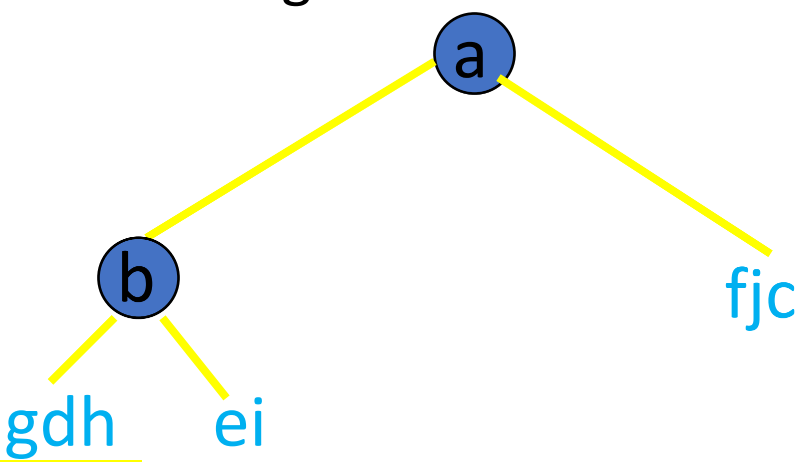
- inorder = g d h b e i a f j c
- preorder = a b d g h e i c f j
- Scan the preorder left to right using the inorder to separate left and right subtrees.
- a is the root of the tree
  - gdhbei are in the left subtree
  - fjc are in the right subtree



Inorder And Preorder

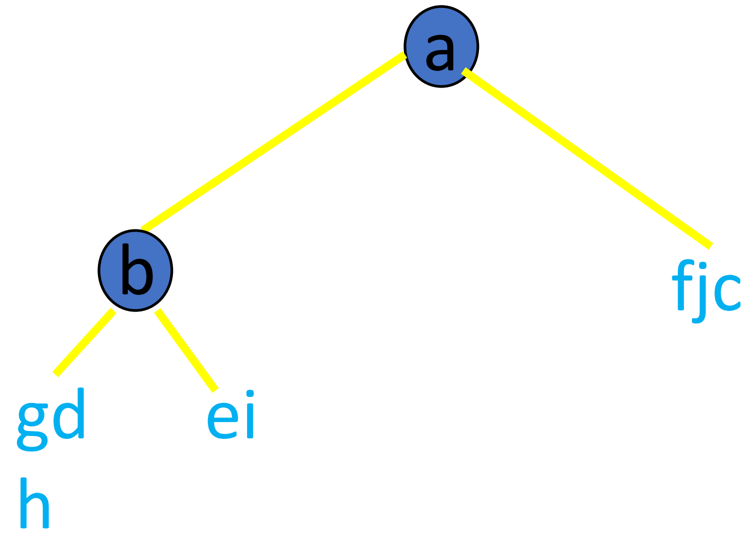


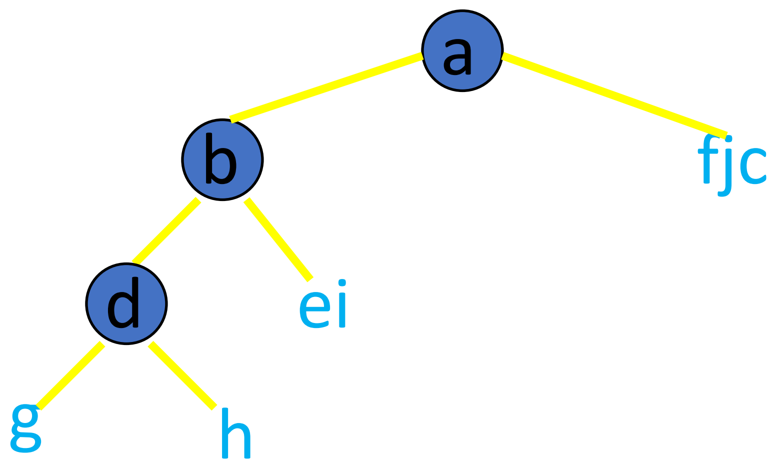
- preorder = a b d g h e i c f j
- b is the next root
  - gdh are in the left subtree
  - ei are in the right subtree



Inorder And Preorder

- preorder = a b d g h e i c f j
- d is the next root
  - g is in the left subtree
  - h is in the right subtree





Inorder And Postorder

- Scan postorder from right to left using inorder to separate left and right subtrees.
- inorder = g d h b e i a f j c
- postorder = g h d i e b j f c a
- Tree root is a
  - gdhbei are in left subtree
  - fjc are in right subtree

Inorder And Level Order

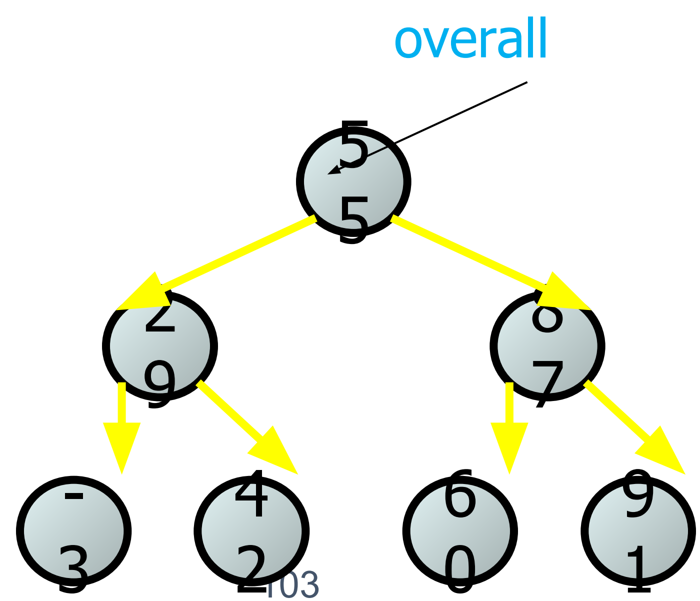
- Scan level order from left to right using inorder to separate left and right subtrees.
- inorder = g d h b e i a f j c
- level order = a b c d e f g h i j
- Tree root is a
  - gdhbei are in left subtree
  - fjc are in right subtree

Types of Trees:

Binary Search Tree (BST)

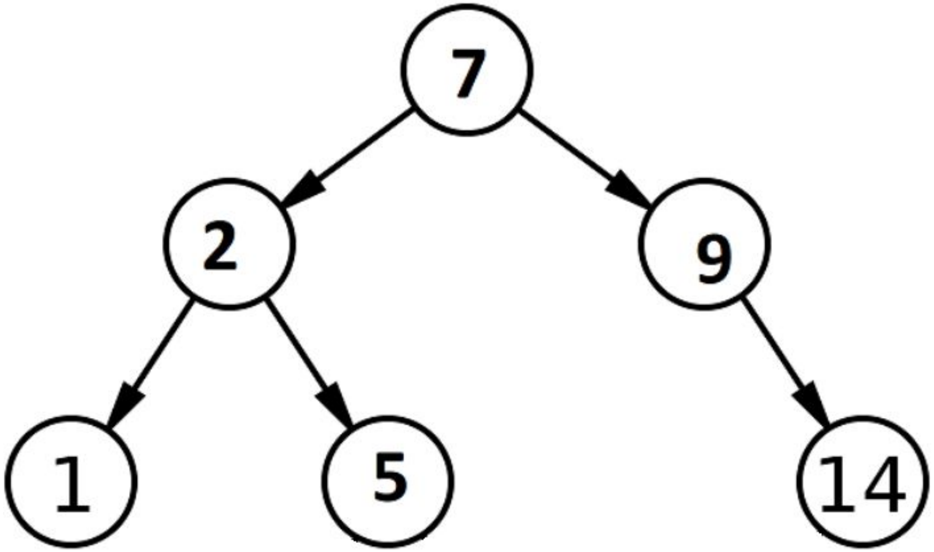
Binary Search Trees

- A binary tree that is either:
  - empty (`null`), or
  - a root node R such that:
    - every element of R's left subtree contains data "less than" R's data
    - every element of R's right subtree contains data "greater than" R's
    - R's left and right subtrees are also
- BSTs store their elements in sorted order, which is helpful for searching/sorting tasks



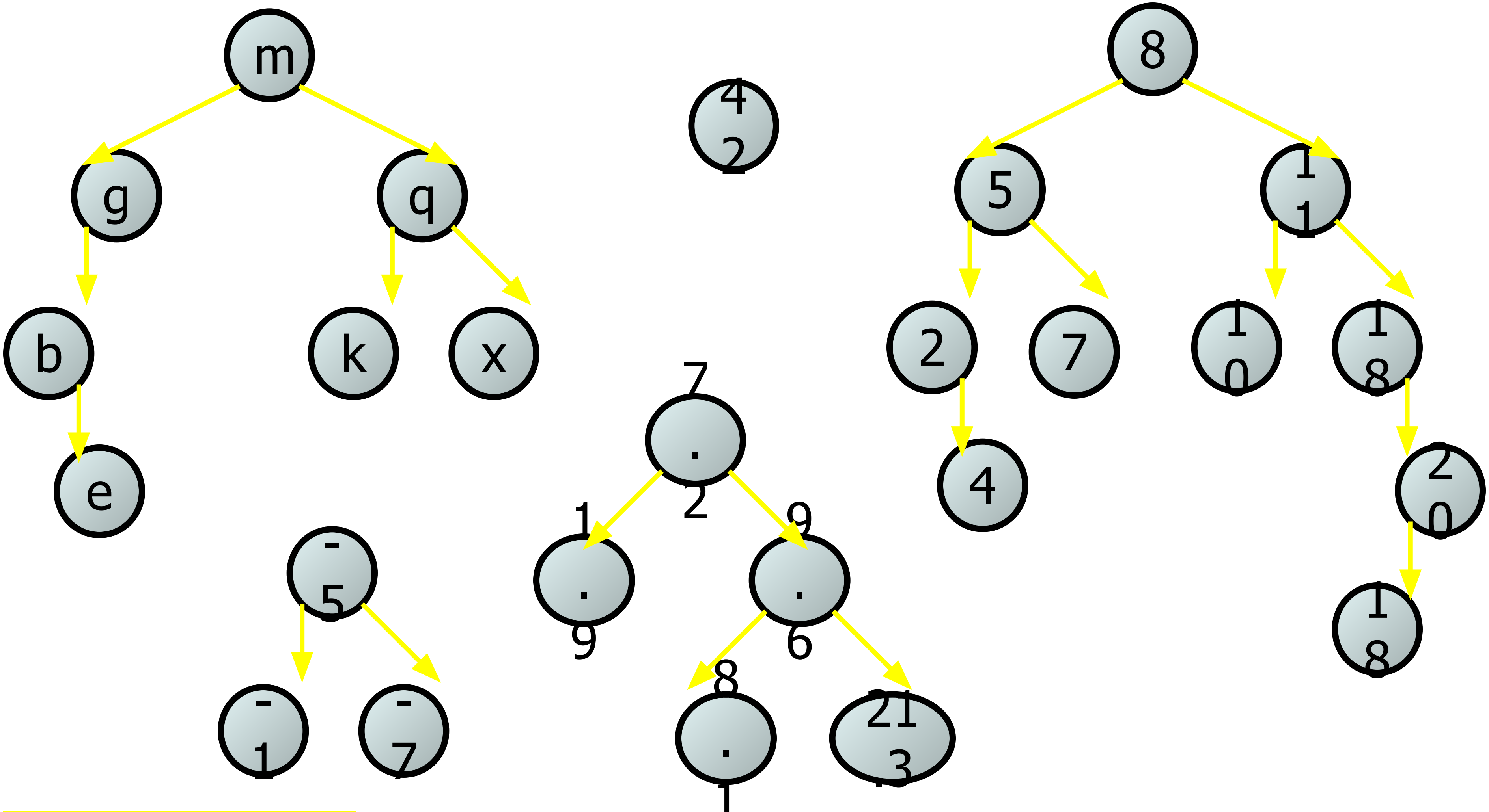
Definition Of Binary Search Tree

- A binary tree
- Each node has a (key, value) pair
- For every node x, all keys in the left subtree of x are smaller than that in x
- For every node x, all keys in the right subtree of x are greater than that in x

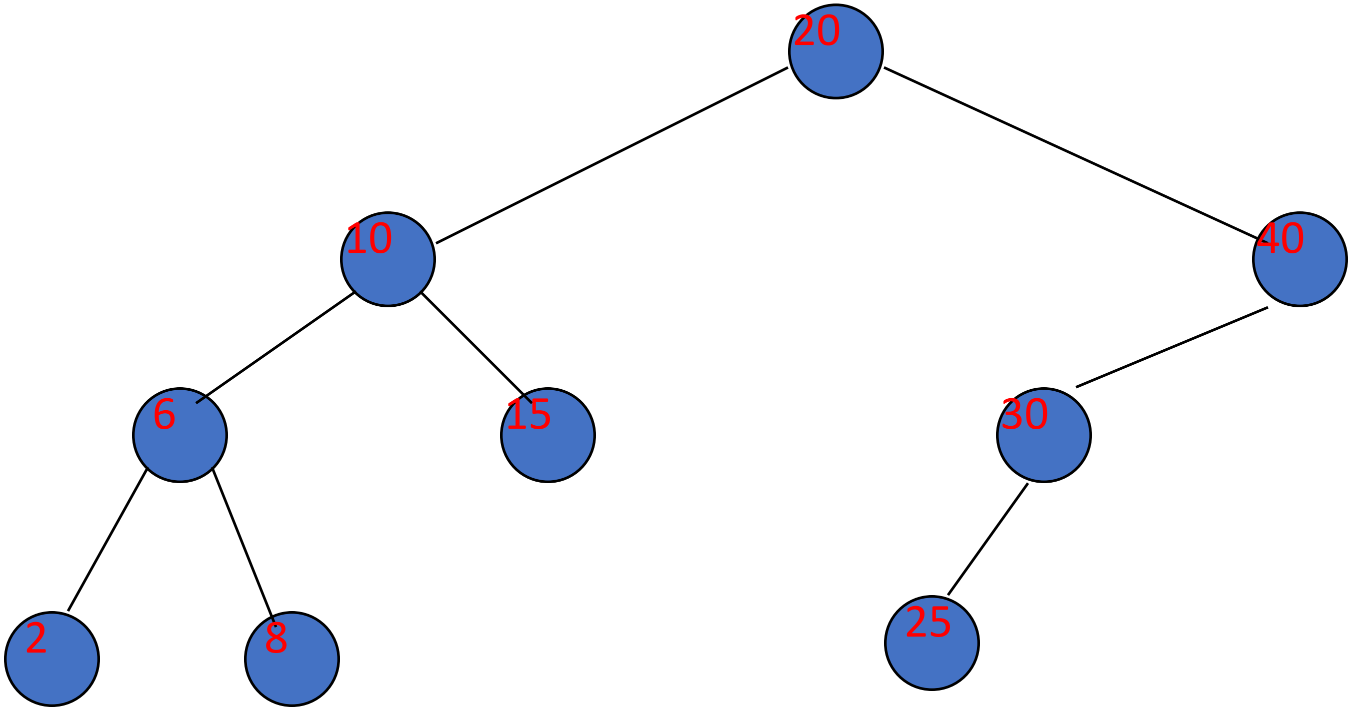


Exercise

- Which of the trees shown are legal binary search trees?



Example Binary Search Tree

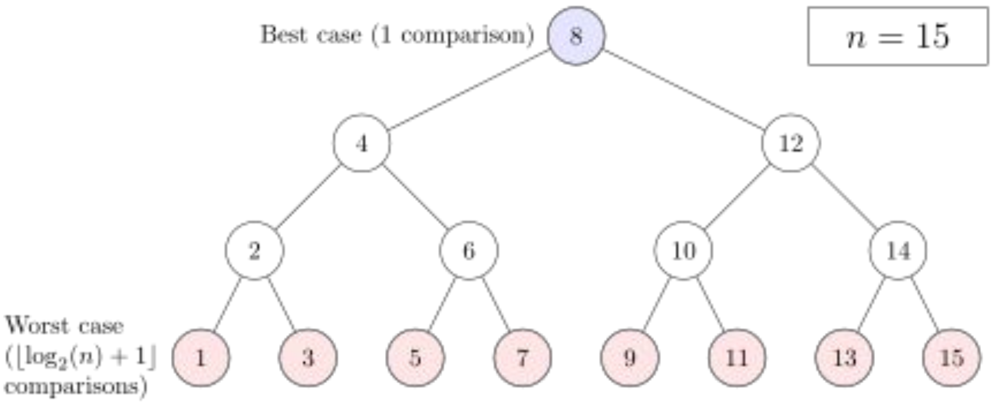


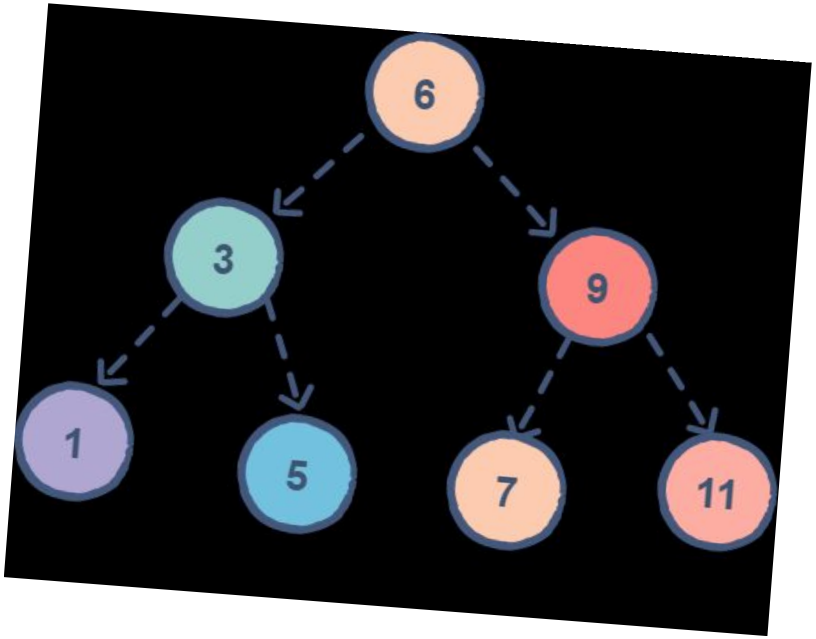
Only keys are shown

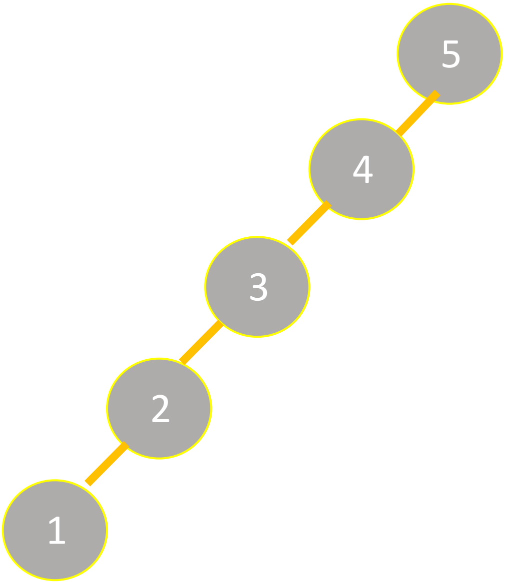
Useful pages

https://www.geeksforgeeks.org/tree-traversals-inorder-preorder-and-postorder/

Examples of a BST.



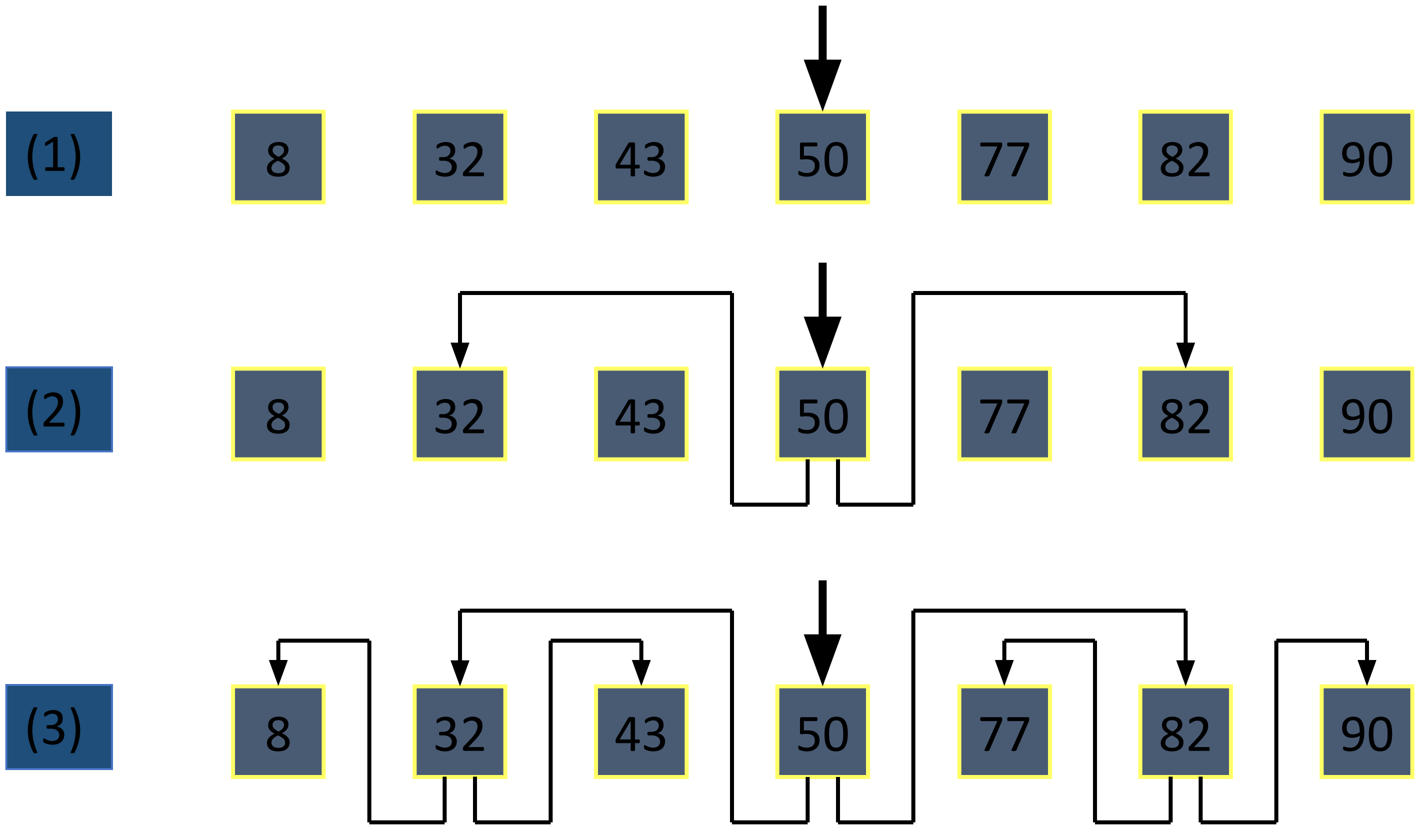




Time Complexity		Space Complexity
Average Case	Worst Case	
O(log n)	O(**n**)	O(n)

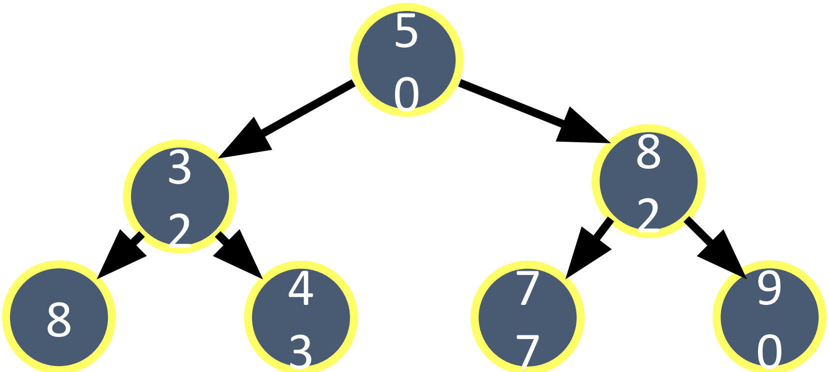
Step-by-Step of Binary Search [1]

13	28	35	49	62	66	80

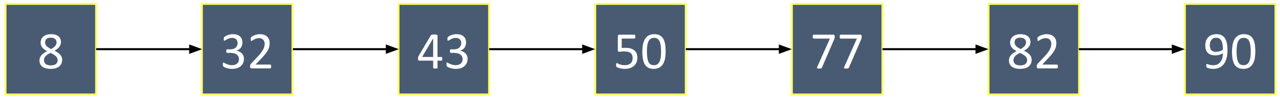


Step-by-Step of Binary Search [1]

Equivalent binary tree structure



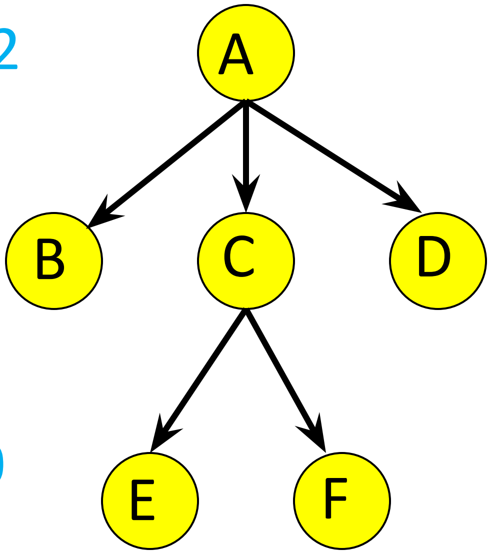
Question: Does binary-search work on sorted Linked-List?



Binary Search Tree Definitions

- Length of a path = number of edges

depth=0, height = 2



- Depth of a node = length of path from root to n

depth = 2, height=0

- Height of node n = length of longest path from n to a leaf

- Depth and height of tree = height of root

Tree animation

https://www.cs.usfca.edu/~galles/visualization/BST.html

Binary Trees: Some Numbers

- Recall: height of a tree = length of longest path from the root to a leaf.
- For binary tree of height *h*:
  - max # of leaves: 2^h
  - max # of nodes: $2^{(h+1)} - 1$
  - min # of leaves: 1
  - min # of nodes: $h + 1$

Implementing Set ADT (Revisited)

	Insert	Remove	Search
Unsorted array	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$
Sorted array	$\Theta(\log(n)+n)$	$\Theta(\log(n) + n)$	$\Theta(\log(n))$
Linked list	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$
BST (if balanced)	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\log n)$

Binary Search Tree (BST)

BST

- Provide an excellent data structure for
  - searching a list
  - Inserting data into the list
  - deleting data into the list

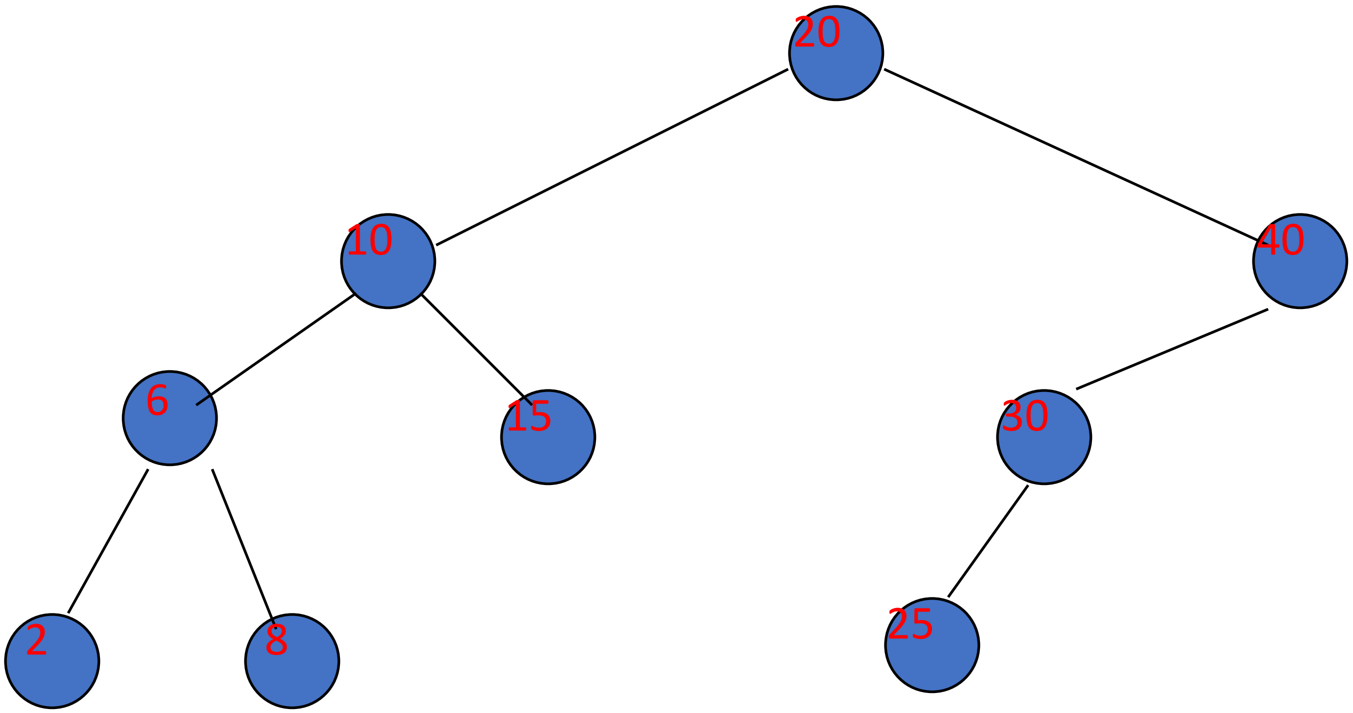
Complexity of different operations in Binary Search Tree (BST).

- Searching: You have to have to traverse all elements. Therefore, searching in binary search tree has worst case complexity of O(n). In general, time complexity is O(h) where h is height of BST.
- Insertion: To inert an element in a BST, one needs to traverse all elements which has worst case complexity of O(n). In general, time complexity is O(h).
- Deletion: To delete an element form a BST, one has to traverse all elements. Therefore, deletion in binary tree has worst case complexity of O(n). In general, time complexity is O(h .

Binary Search Trees

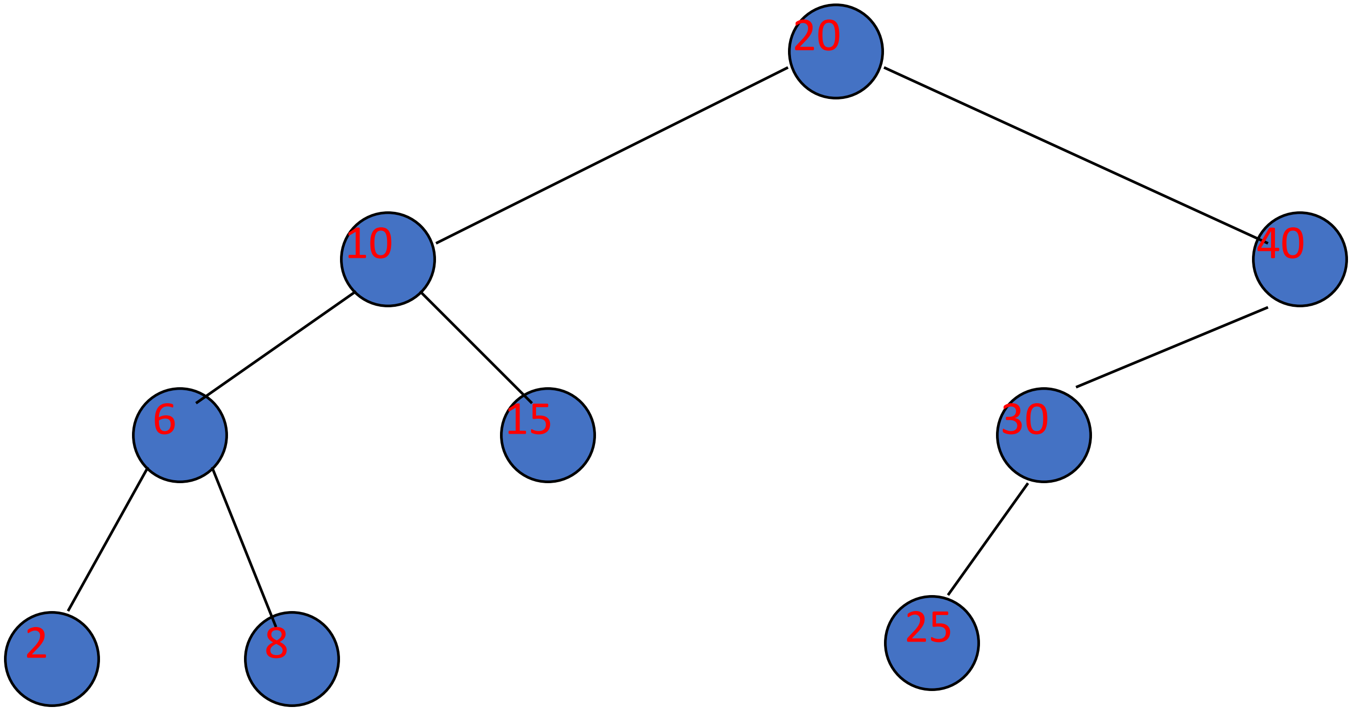
- Dictionary Operations:
  - get(key)
  - put(key, value)
  - remove(key)
- Additional operations:
  - ascend()
  - get(index) (indexed binary search tree)
  - remove(index) (indexed binary search tree)

The Operation ascend()



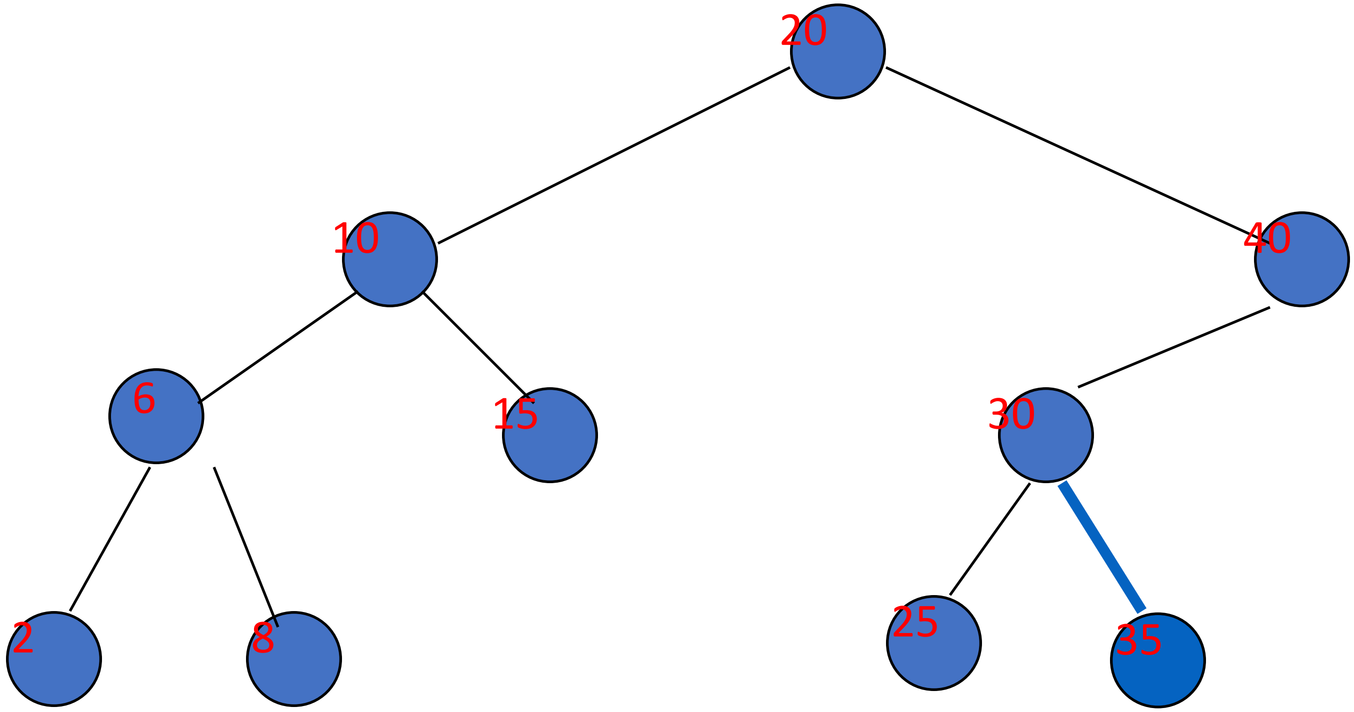
Do an inorder traversal. O(n) time

The Operation get()



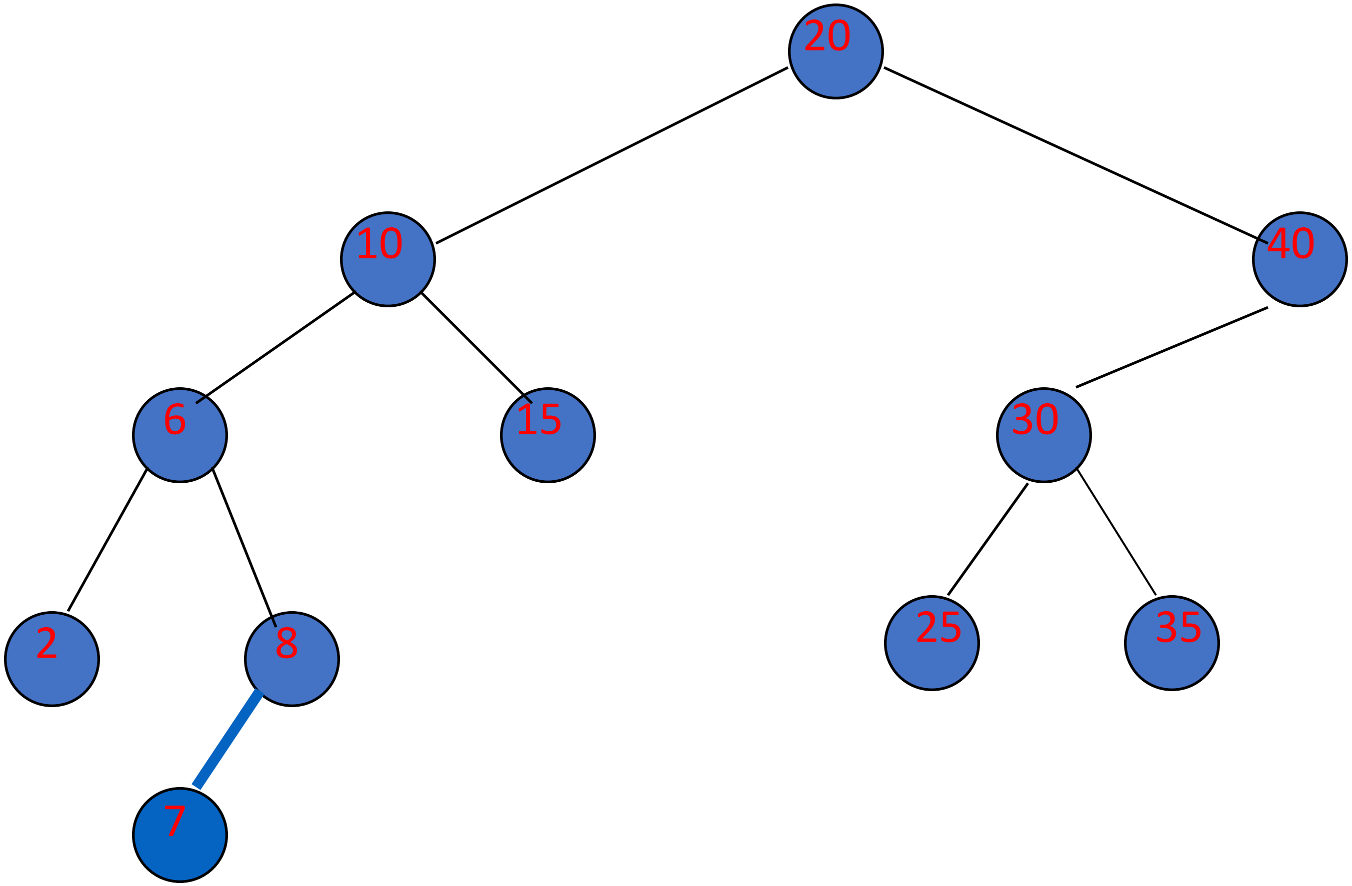
Complexity is O(height) = O(n), where n is number of nodes/elements

The Operation put()



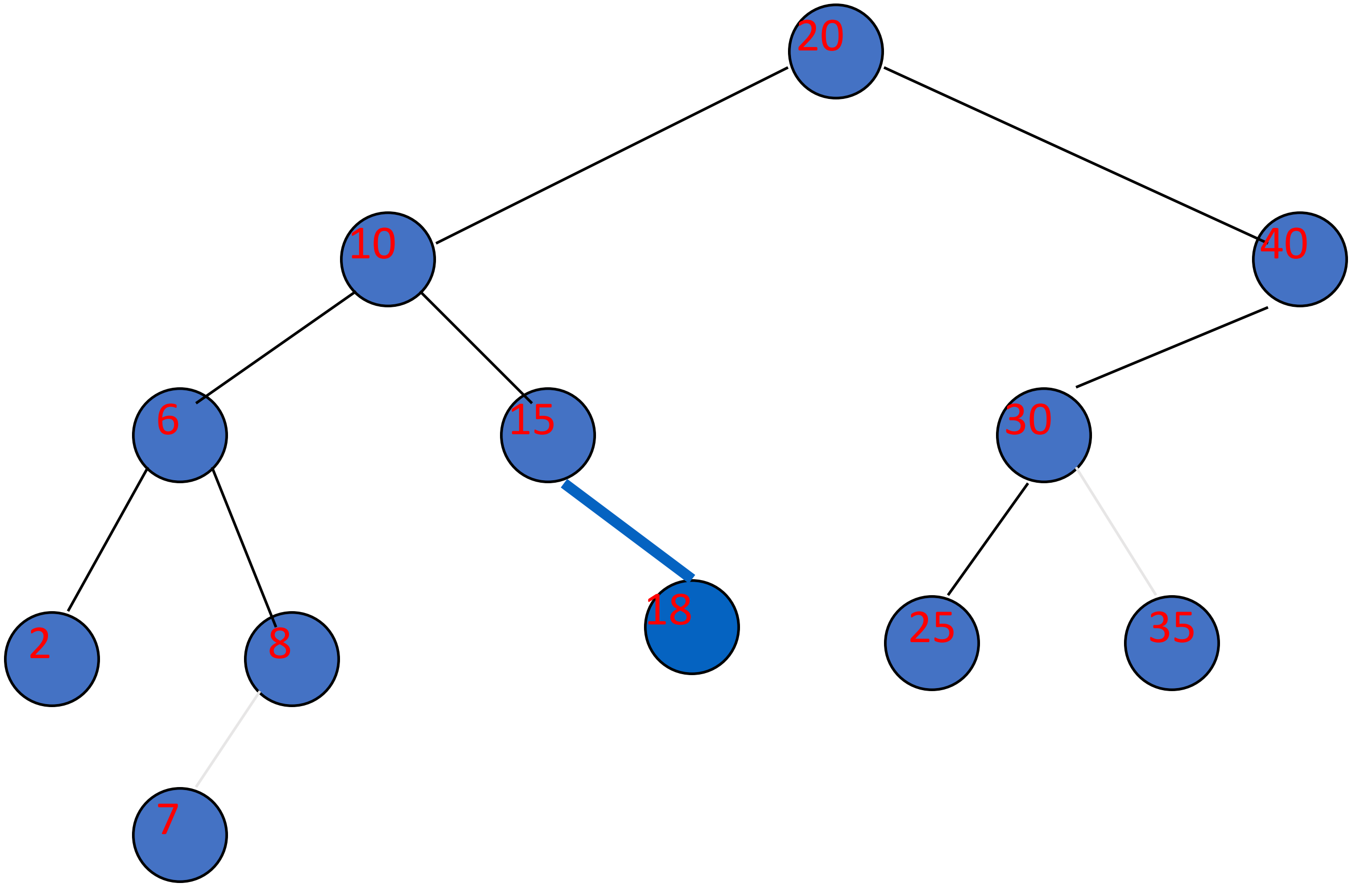
35

The Operation put()



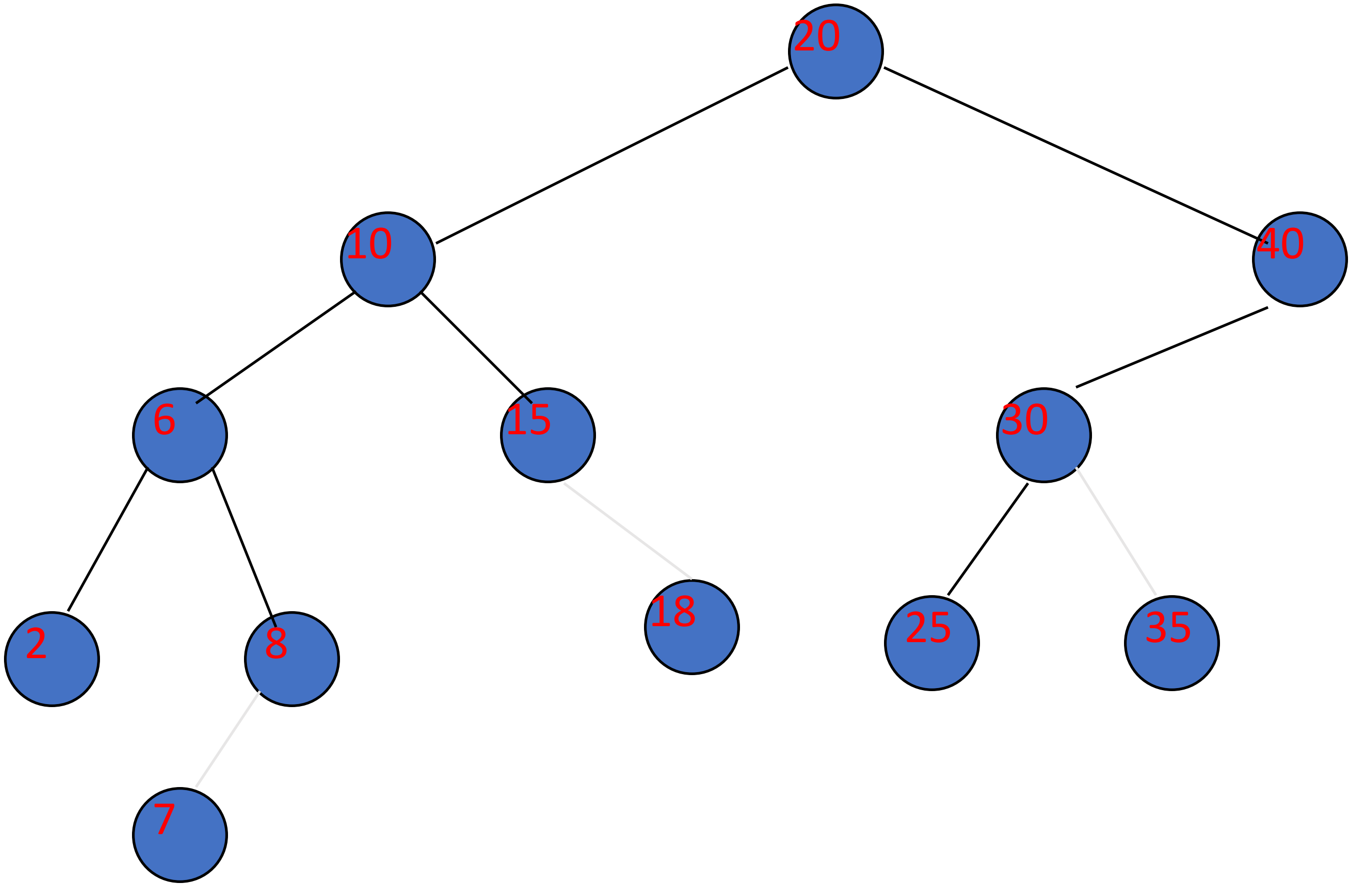
Put a pair whose key is 7.

The Operation put()



Put a pair whose key is 18.

The Operation put()



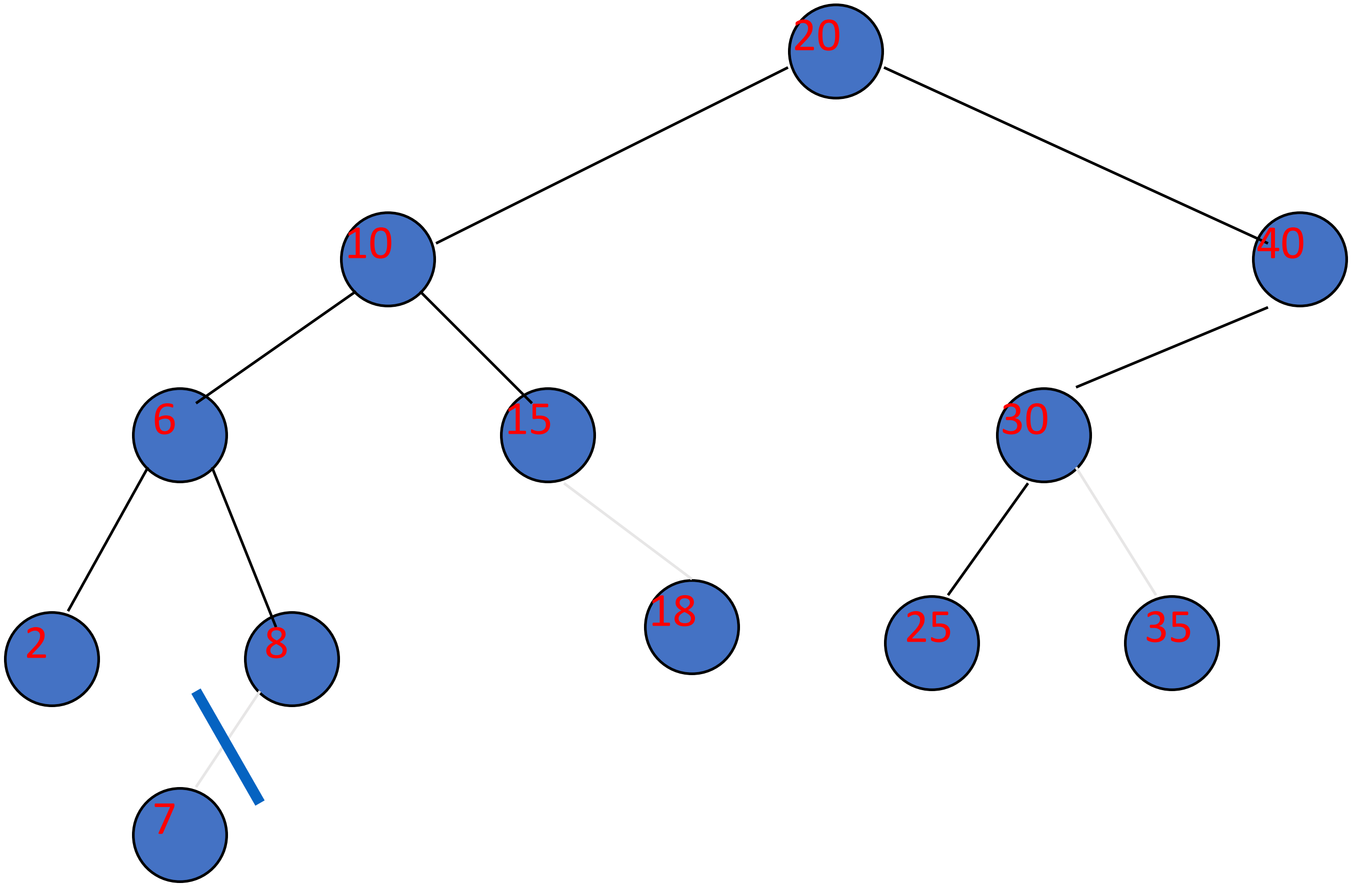
Complexity of put() is O(height).

The Operation remove()

Three cases:

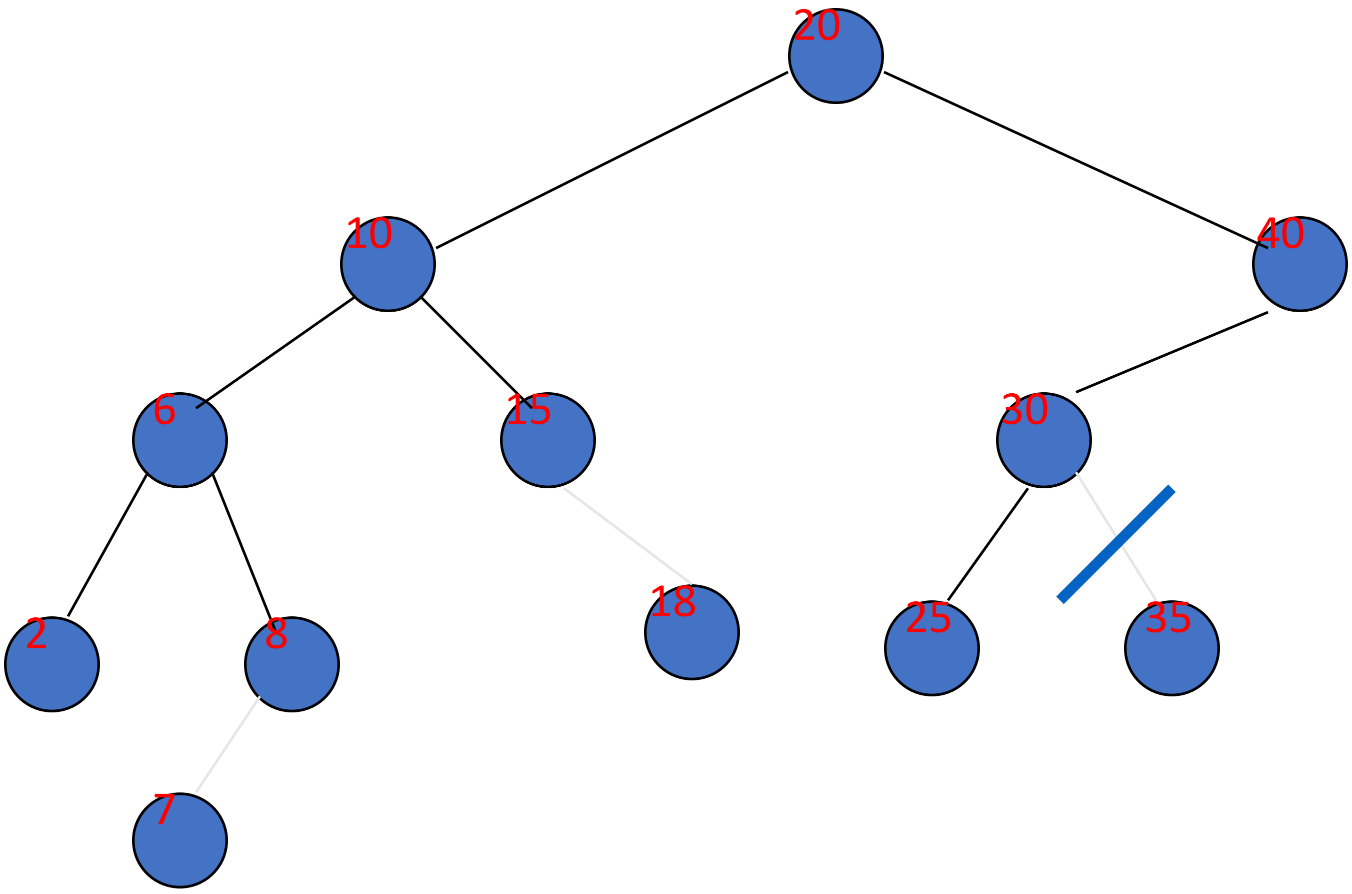
- Element is in a leaf
- Element is in a degree 1 node
- Element is in a degree 2 node

Remove from a Leaf



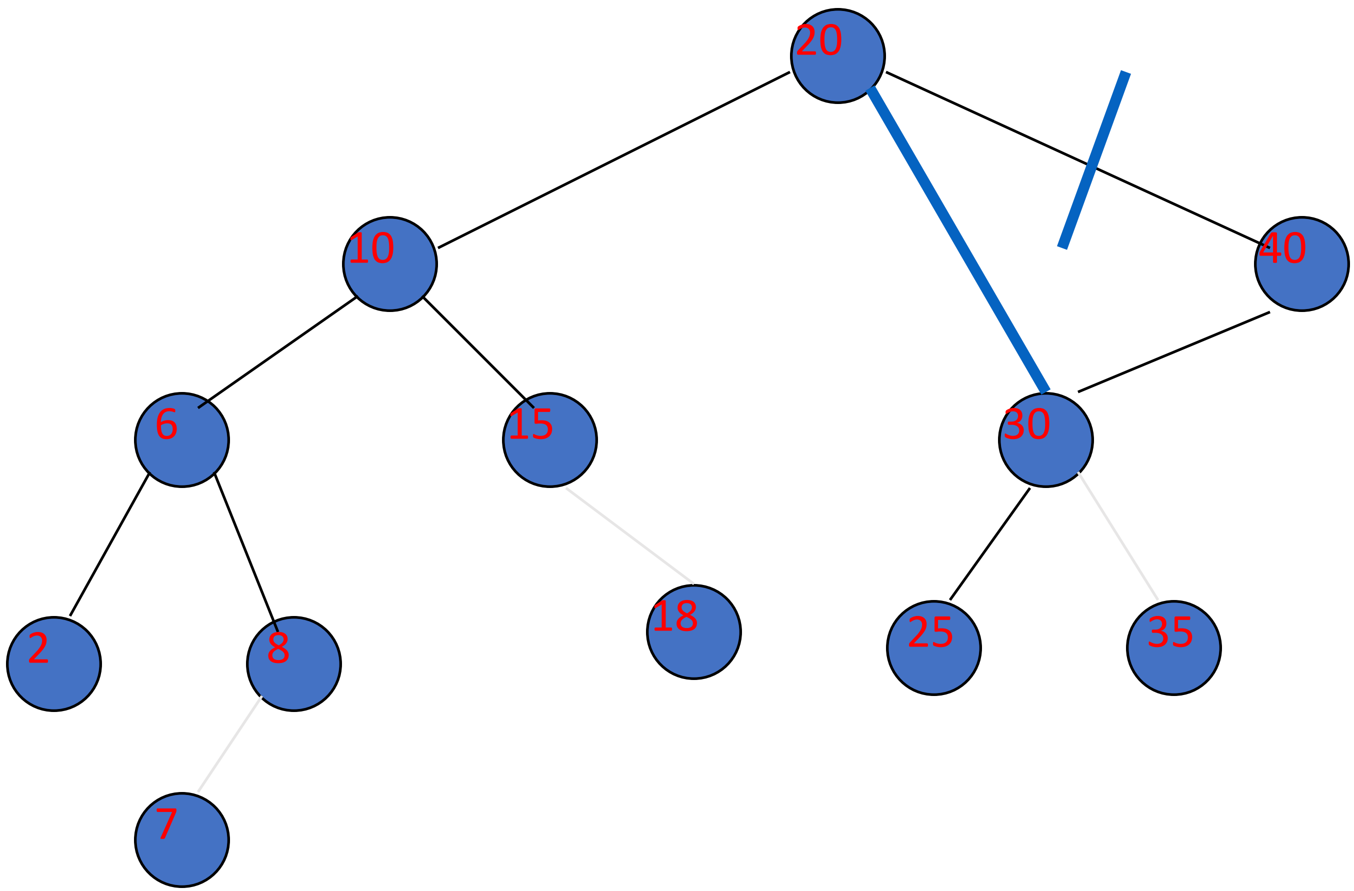
Remove a leaf element. key = 7

Remove from a Leaf (contd.)



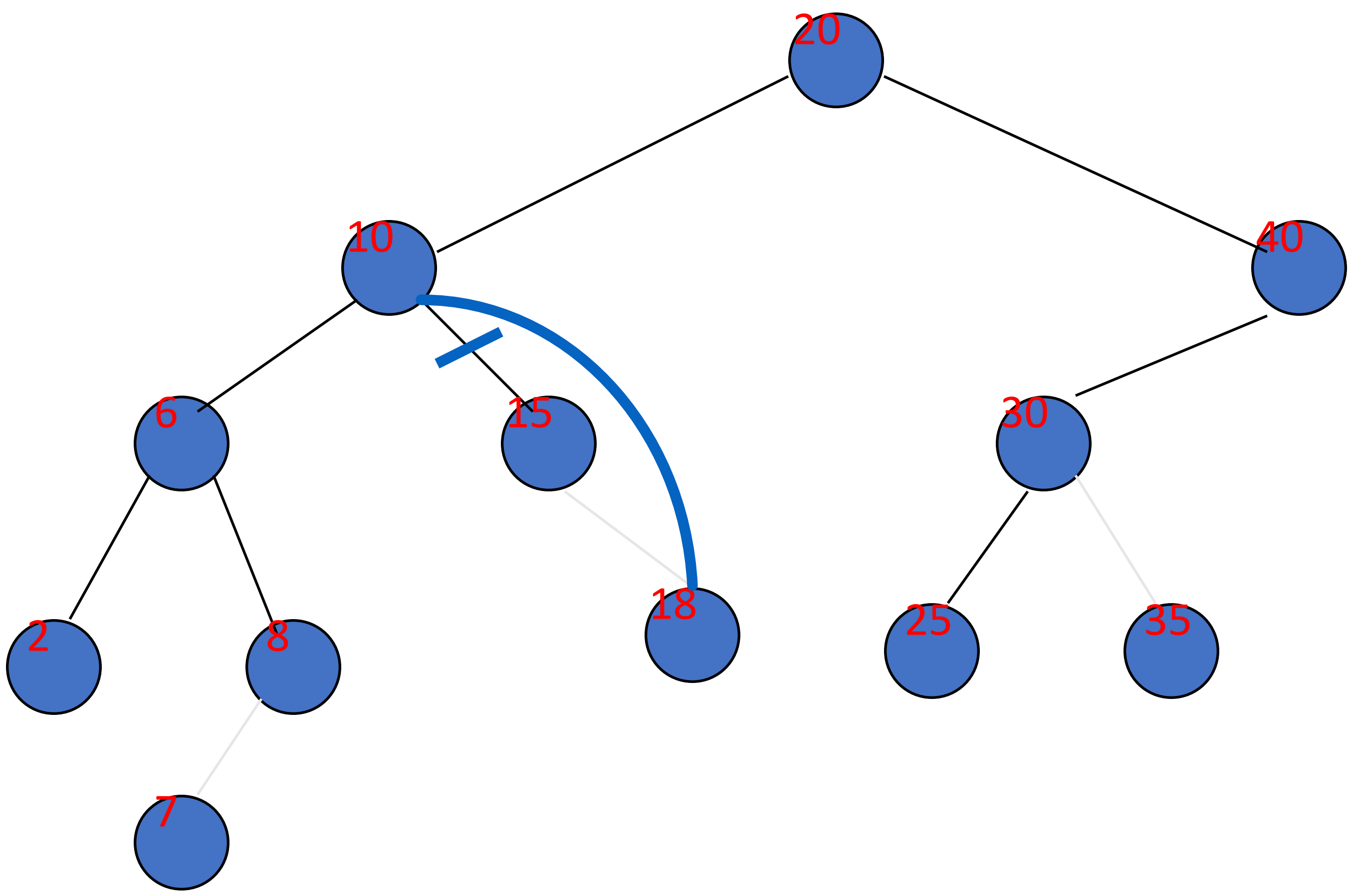
Remove a leaf element. key = 35

Remove from a Degree 1 Node



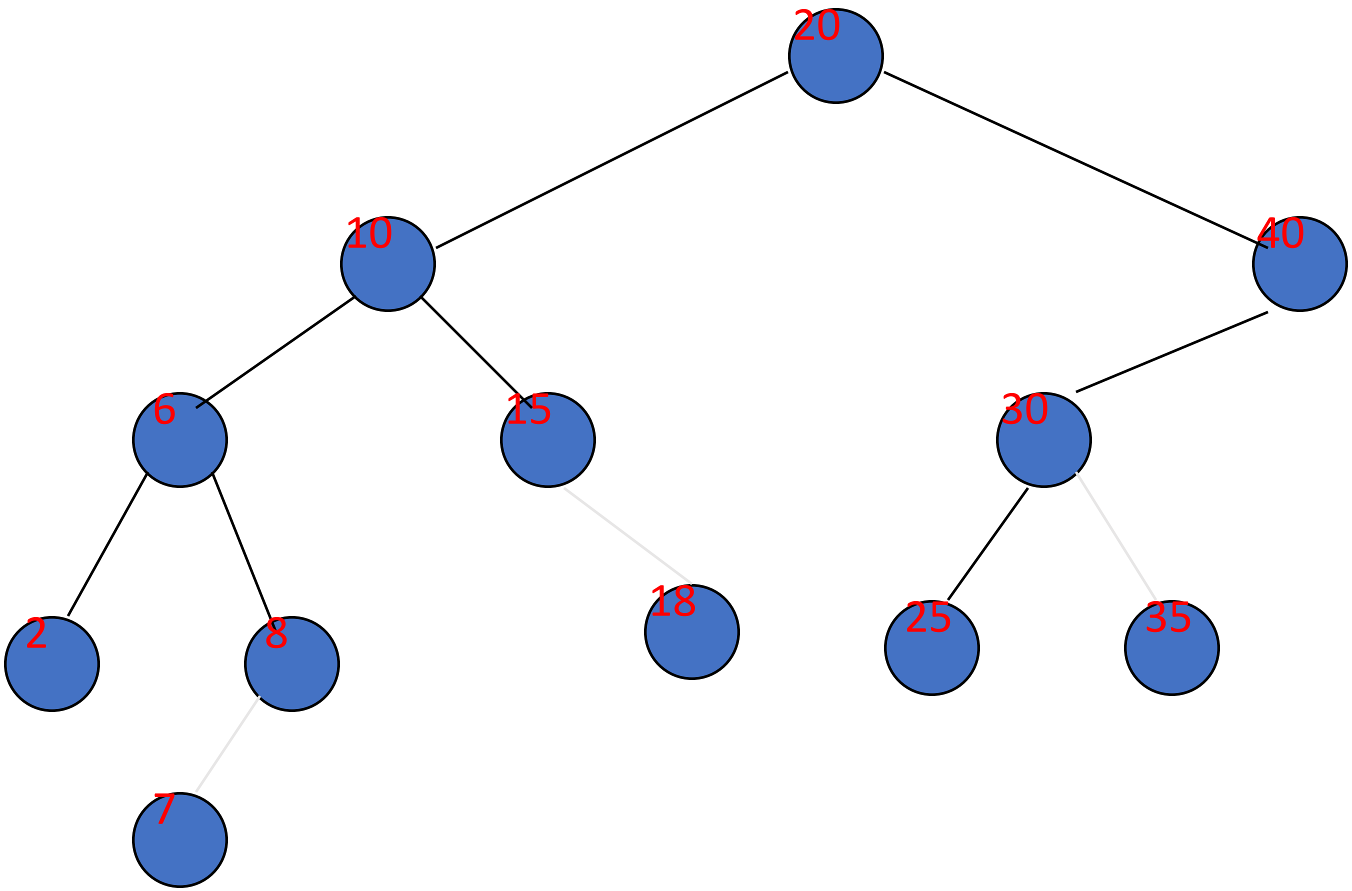
Remove from a degree 1 node. key = 40

Remove from a Degree 1 Node (contd.)



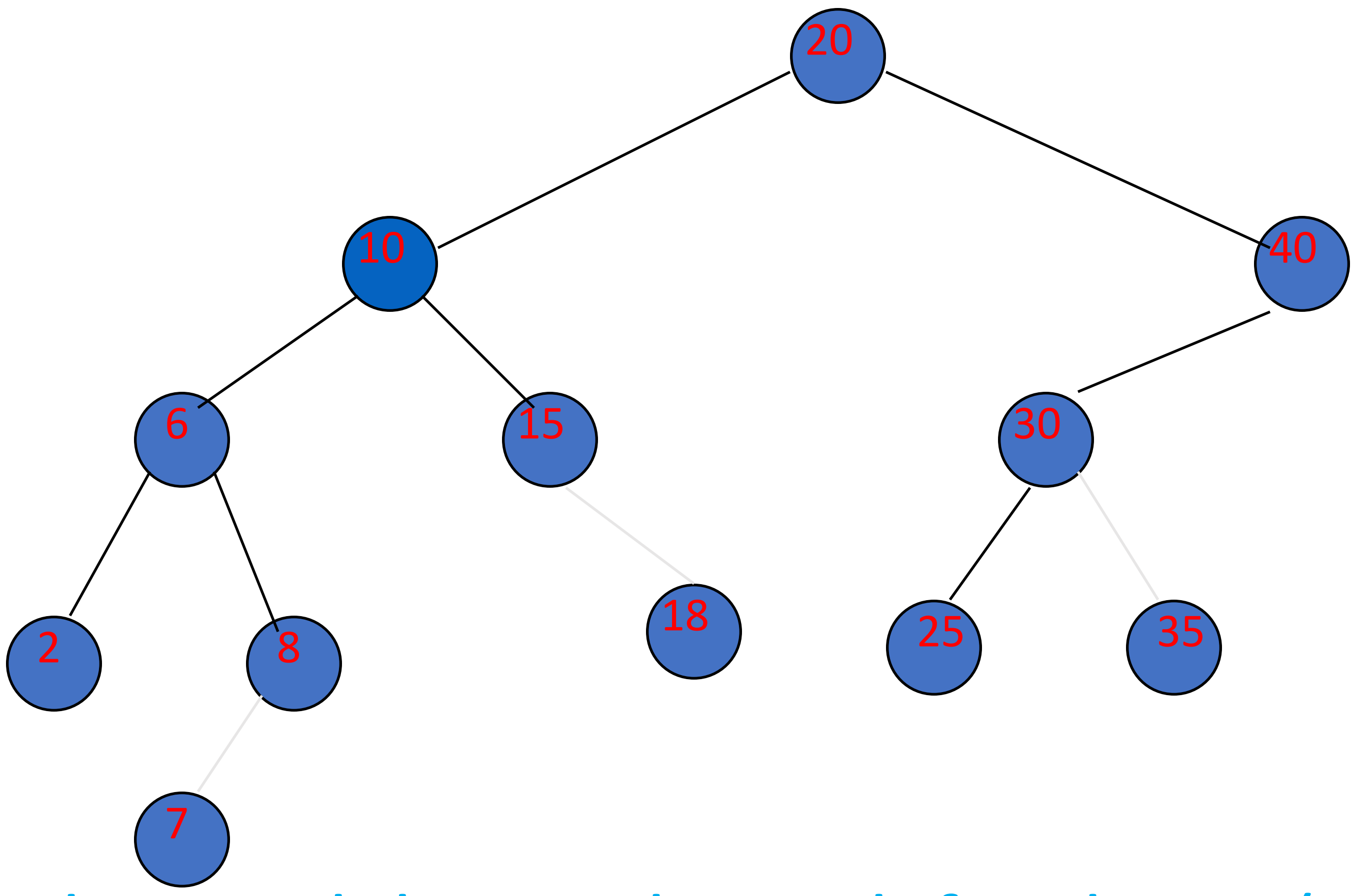
Remove from a degree 1 node. key = 15

Remove from a Degree 2 Node



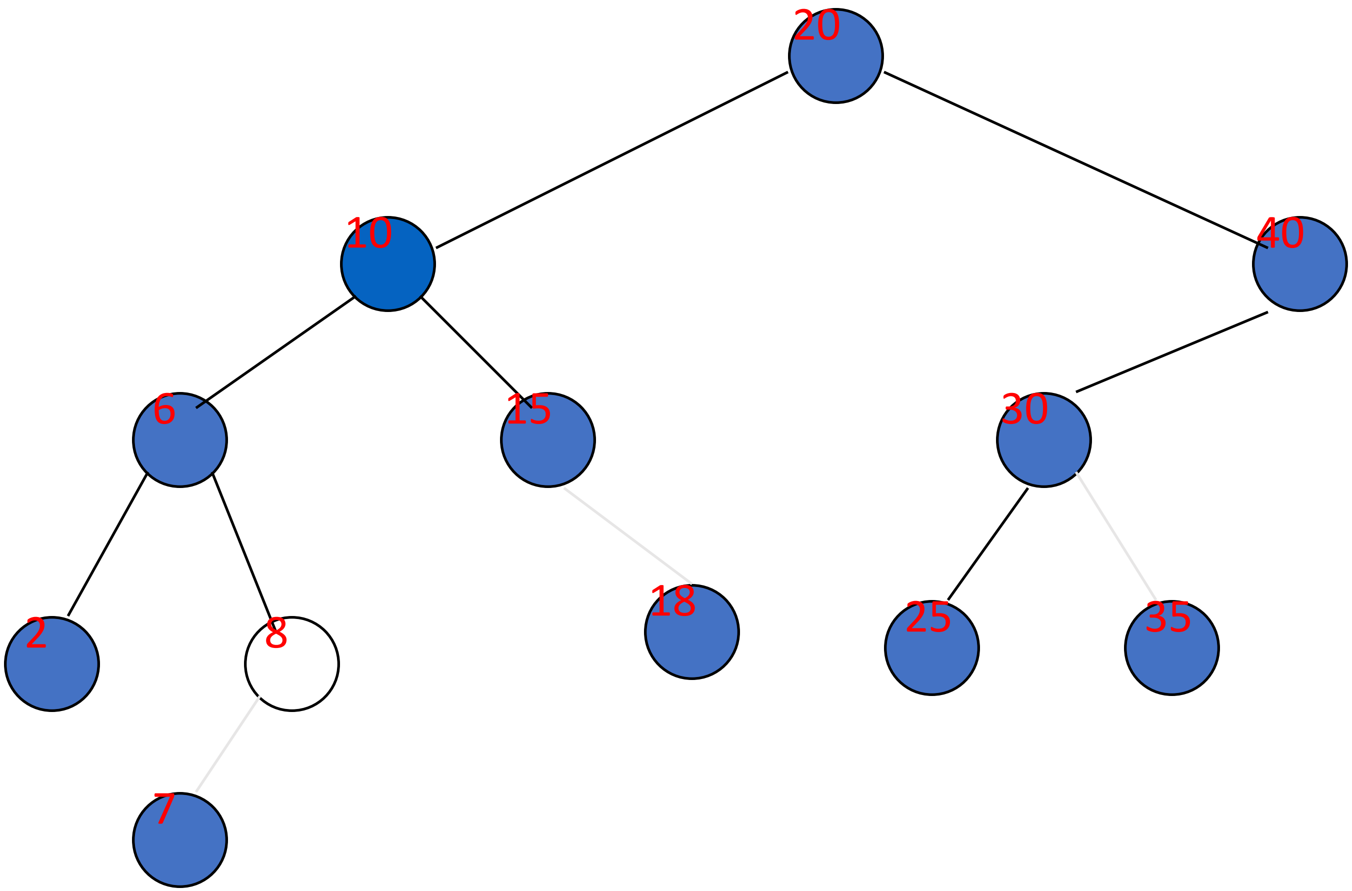
Remove from a degree 2 node. key = 10

Remove from a Degree 2 Node



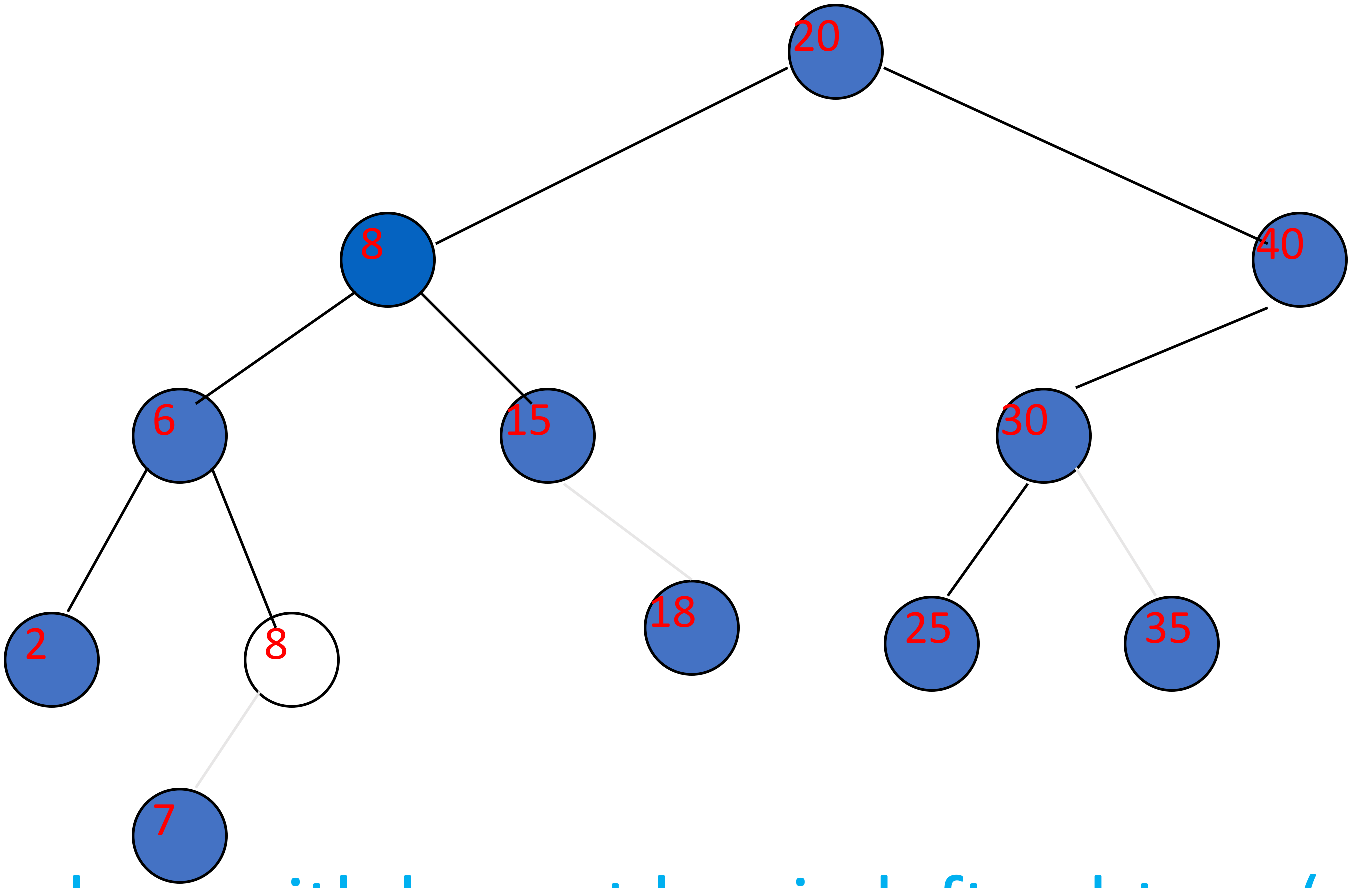
Replace with largest key in left subtree (or smallest in right subtree)

Remove from a Degree 2 Node



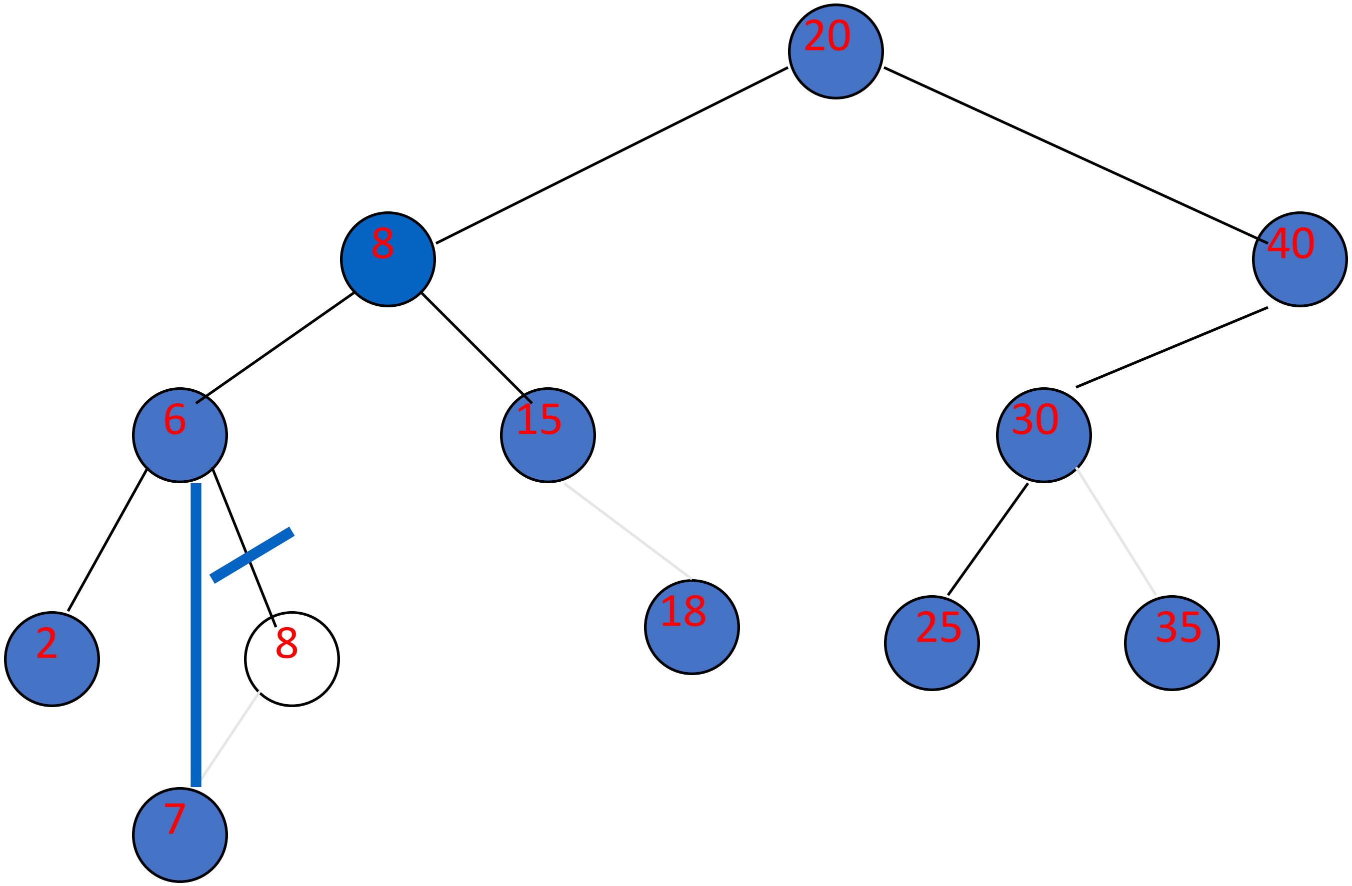
Replace with largest key in left subtree (or smallest in right subtree)

Remove from a Degree 2 Node



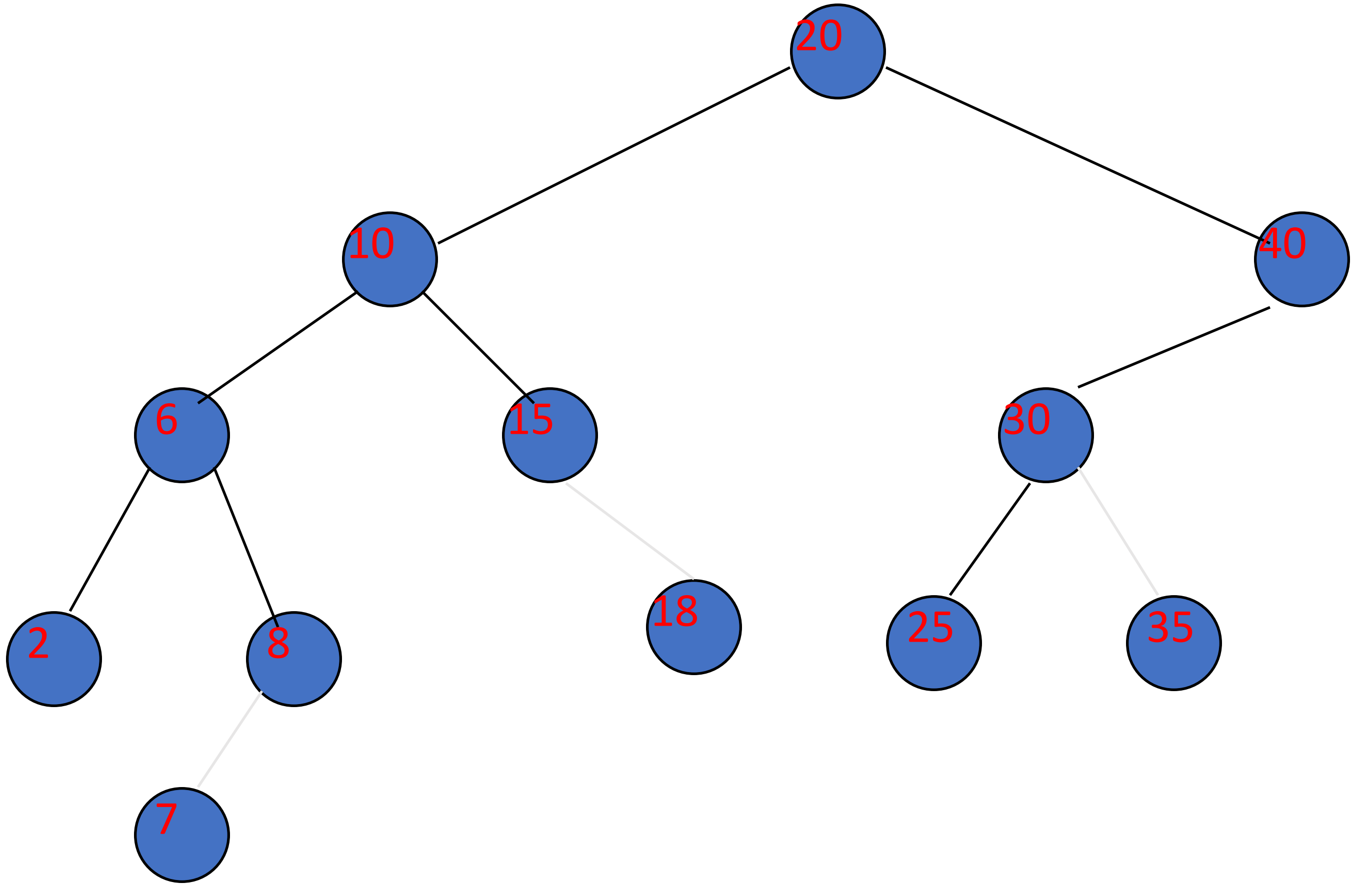
Replace with largest key in left subtree (or smallest in right subtree).

Remove from a Degree 2 Node



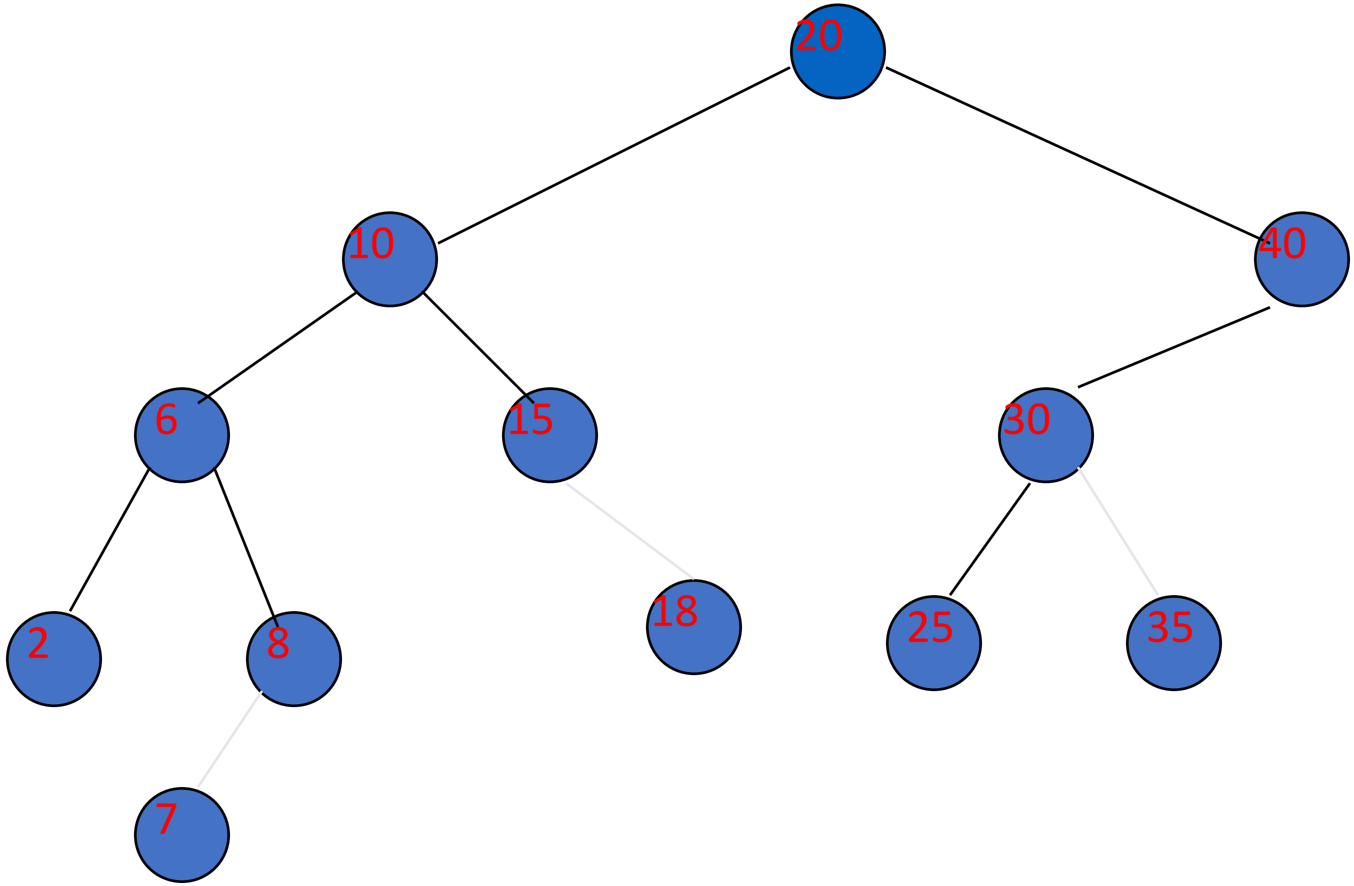
Largest key must be in a leaf or degree 1 node.

Another Remove from a Degree 2 Node



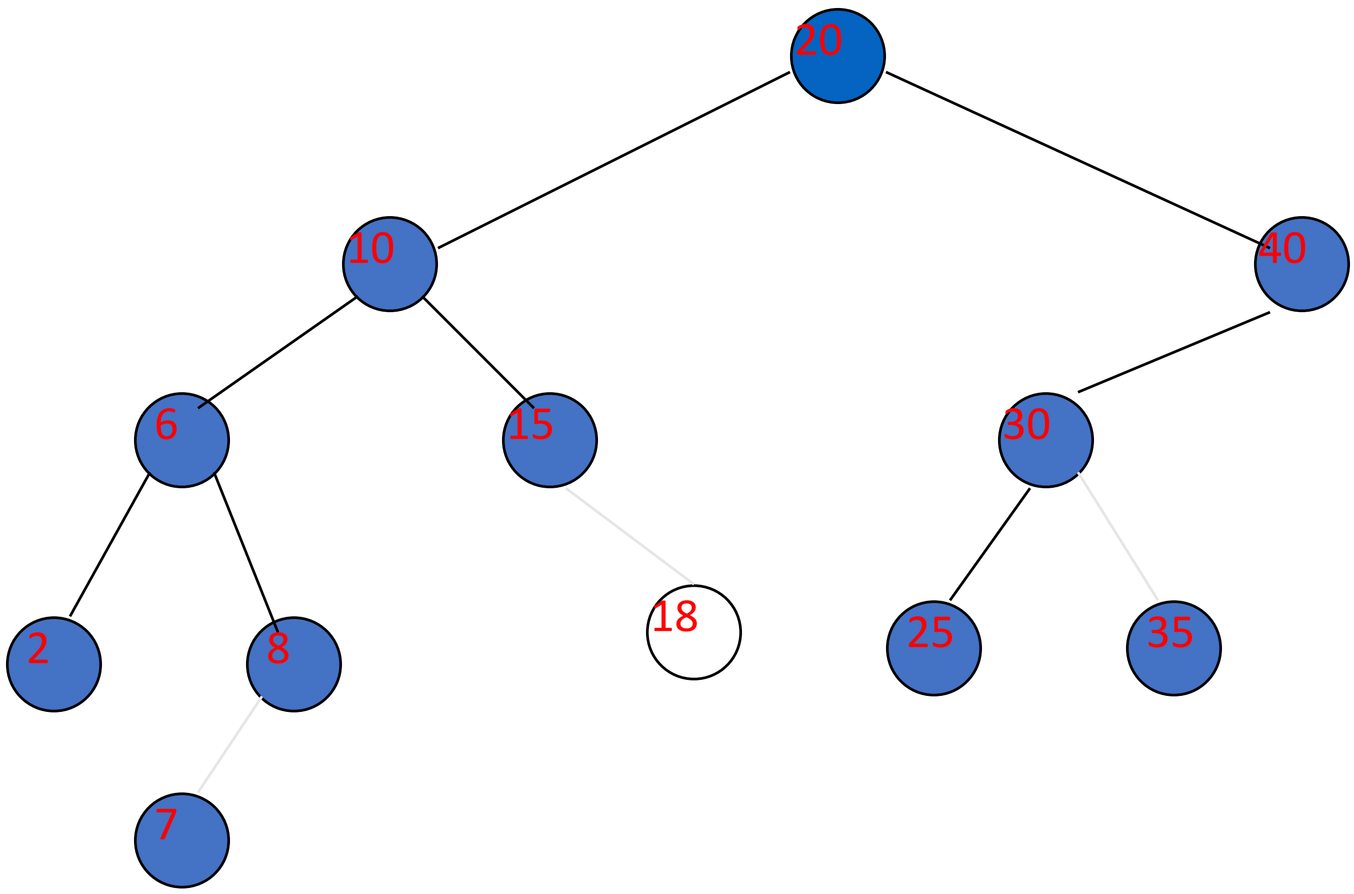
Remove from a degree 2 node. key = 20

Remove from a Degree 2 Node



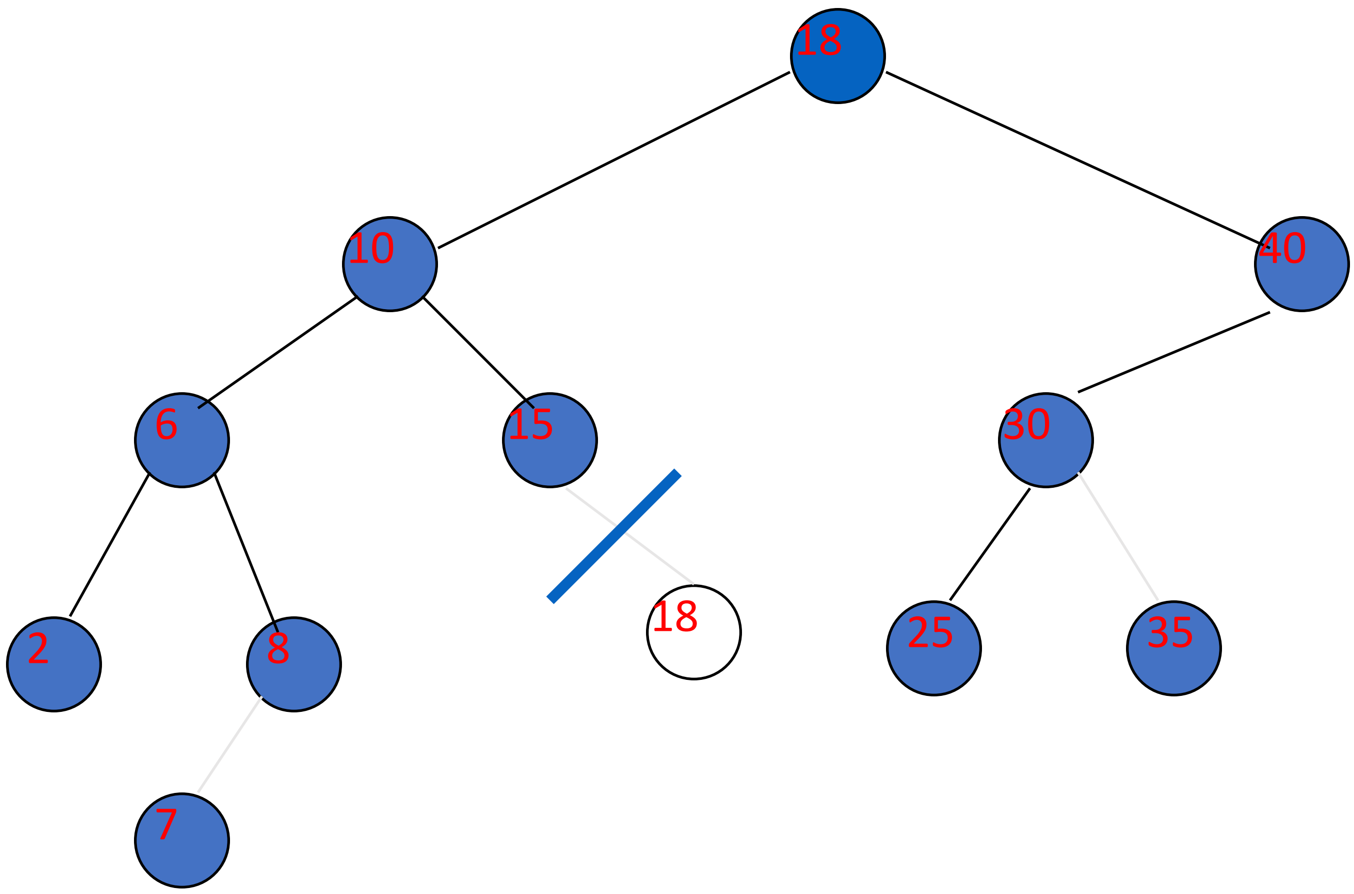
Replace with largest in left subtree.

Remove from a Degree 2 Node



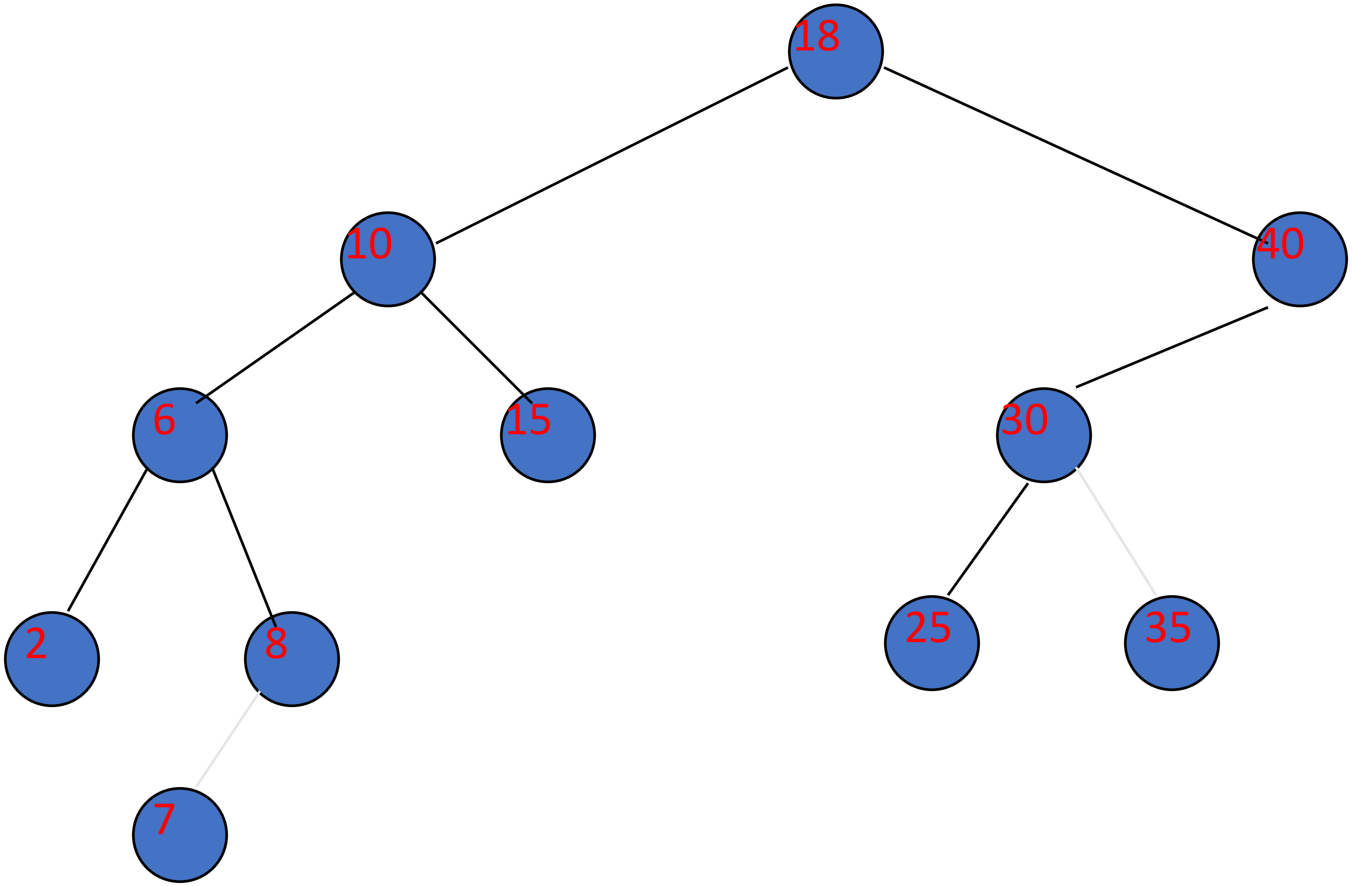
Replace with largest in left subtree.

Remove from a Degree 2 Node



Replace with largest in left subtree.

Remove from a Degree 2 Node



Complexity is O(height).

Complexity of Dictionary Operations

Data Structure	Worst Case	Expected
Hash Table	O(n)	O(1)
Binary Search Tree	O(n)	O(log n)
Balanced Binary Search Tree	O(log n)	O(log n)

n is number of elements in dictionary

Complexity of Other Operations

Data Structure	ascend	get and remove
Hash Table	$O(D + n \log n)$	$O(D + n \log n)$
Indexed BST	$O(n)$	$O(n)$
Indexed Balanced BST	$O(n)$	$O(\log n)$

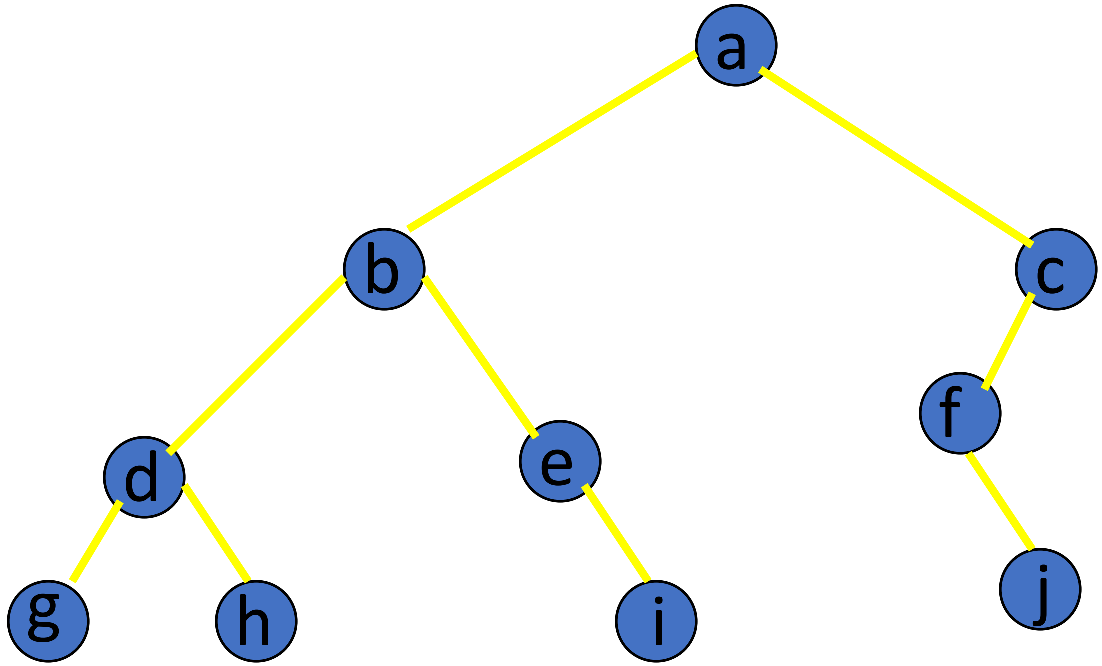
D is number of buckets

Binary Search Tree (BST)

- Binary Search Tree and its implementation
  - insertion
  - traversal
  - deletion

- Application:
  - evaluate expression tree

Traversal Applications



- Make a clone
- Determine height
- Determine number of nodes

Expression Trees

Arithmetic Expressions

$Y = (a + b) * (c + d) + e - f/g*h + 3.25$

- Expressions comprise three kinds of entities
  - Operators (+, -, /, *, %)
  - Operands (a, b, c, d, e, f, g, h, 3.25, (a + b), (c + d), etc...)
  - Delimiters ((,))

Operator Degree

$$Y = (a + b) * (c + d) + e - f/g*h + 3.25$$

- Number of operands that the operator requires
- Binary operator requires two operands

  a + b

  c / d

  e - f

- Unary operator requires one operand

  + g

  - h

Infix Form

- Normal way to write an expression
- Binary operators come in between their left and right operands

    a * b

    a + b * c

    a * b / c

    (a + b) * (c + d) + e – f/g*h + 3.25

Operator Priorities

- How do you figure out the operands of an operator?

  a + b * c

  a * b + c / d

- This is done by assigning operator priorities

  priority(*) = priority(/) > priority(+) = priority(-)

- When an operand lies between two operators, the operand associates with the operator that has higher priority

Tie Breaker

- When an operand lies between two operators that have the same priority, the operand associates with the operator on the left

    a + b - c

    a * b / c / d

Delimiters

- Subexpression within delimiters is treated as a single operand, independent from the remainder of the expression

  (a + b) * (c – d) / (e – f)

- Need operator priorities, tie breaker, and delimiters
- This makes computer evaluation more difficult than is necessary
- Postfix and prefix expression forms do not rely on operator priorities, a tie breaker, or delimiters
- So it is easier for a computer to evaluate expressions that are in these forms

Postfix Form

- The postfix form of a variable or constant is the same as its infix form

    a, b, 3.25

- The relative order of operands is the same in infix and postfix forms
- Operators come immediately **after** the postfix form of their operands

    Infix = a + b

    Postfix = ab+

Postfix Examples

- Infix = a + b * c

  Postfix = a b c * +

- Infix = a * b + c

  Postfix = a b * c +

- Infix = (a + b) * (c – d) / (e + f)

  Postfix = a b + c d - * e f + /

Unary Operators

- Replace with new symbols

  + a => a @

  + a + b => a @ b +

  - a => a ?

  - a-b => a ? b -

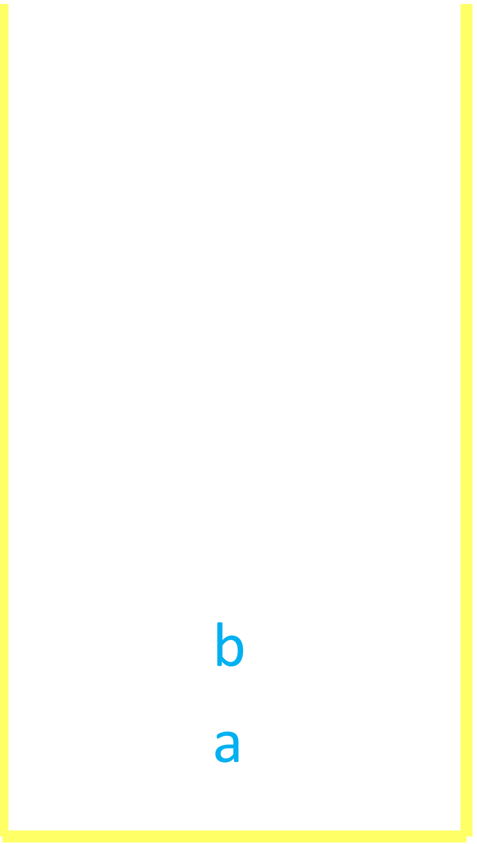
Postfix Evaluation

- Scan postfix expression from left to right pushing operands on to a stack
- When an operator is encountered
  - pop as many operands as this operator needs
  - evaluate the operator
  - push the result on to the stack
- This works because, in postfix, operators come immediately after their operands

Postfix Evaluation

- (a + b) * (c – d) / (e + f)
- a b + c d - * e f + /
- a b + c d - * e f + /

- a b + c d - * e f + /
- a b + c d - * e f + /



stack

Postfix Evaluation

- (a + b) * (c – d) / (e + f)
- a b + c d - * e f + /
- a b + c d - * e f + /

- a b + c d - * e f + /
- a b + c d - * e f + /
- a b + c d - * e f + /
- a b + c d - * e f + /
- a b + c d - * e f + /

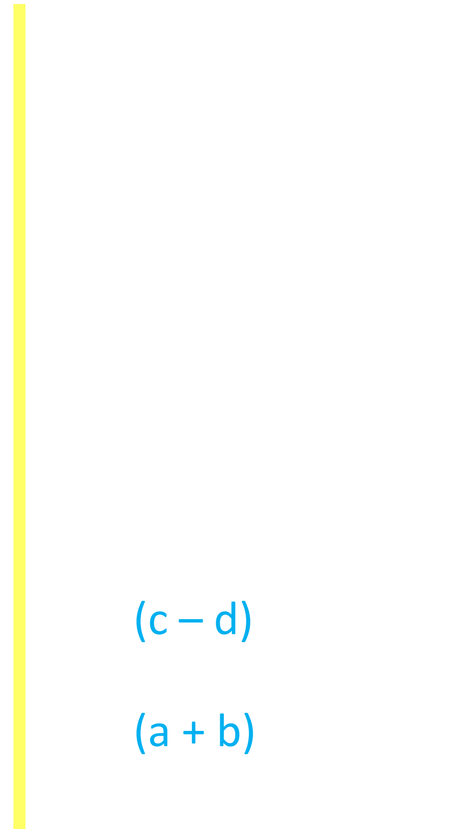
stack
d
c
(a + b)

stack

Postfix Evaluation

- (a + b) * (c – d) / (e + f)
- a b + c d - * e f + /

- a b + c d - * e f + /



stack

Postfix Evaluation

- (a + b) * (c – d) / (e + f)
- a b + c d - * e f + /

- a b + c d - * e f + /
- a b + c d - * e f + /
- a b + c d - * e f + /
- a b + c d - * e f + /

f
e
(a + b)*(c – d)

stack

Postfix Evaluation

- (a + b) * (c – d) / (e + f)
- a b + c d - * e f + /

- a b + c d - * e f + /
- a b + c d - * e f + /
- a b + c d - * e f + /
- a b + c d - * e f + /
- a b + c d - * e f + /

(e + f)

(a + b)*(c – d)

stack

Postfix Evaluation

- The prefix form of a variable or constant is the same as its infix form

  a, b, 3.25

- The relative order of operands is the same in infix and prefix forms
- Operators come immediately **before** the prefix form of their operands

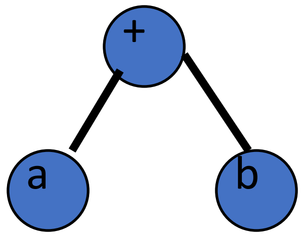
  Infix = a + b

  Postfix = ab+

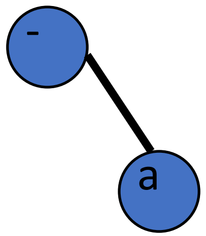
  Prefix = +ab

Binary SearchTree Form

- a + b

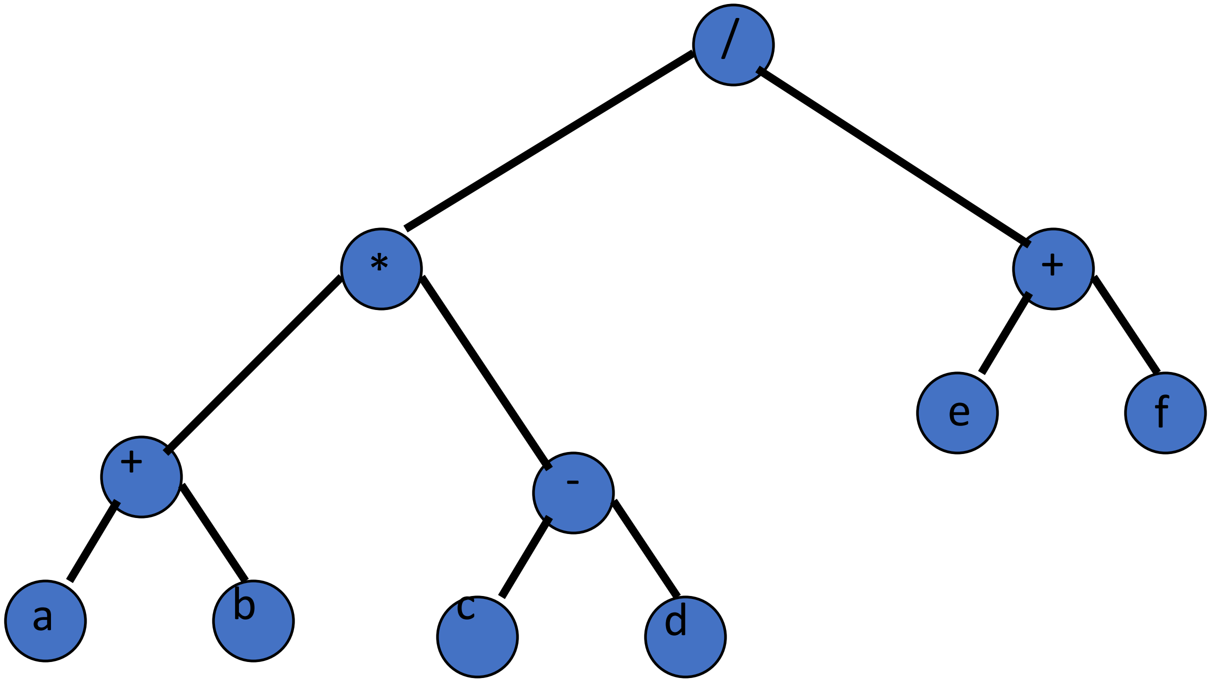


- - a



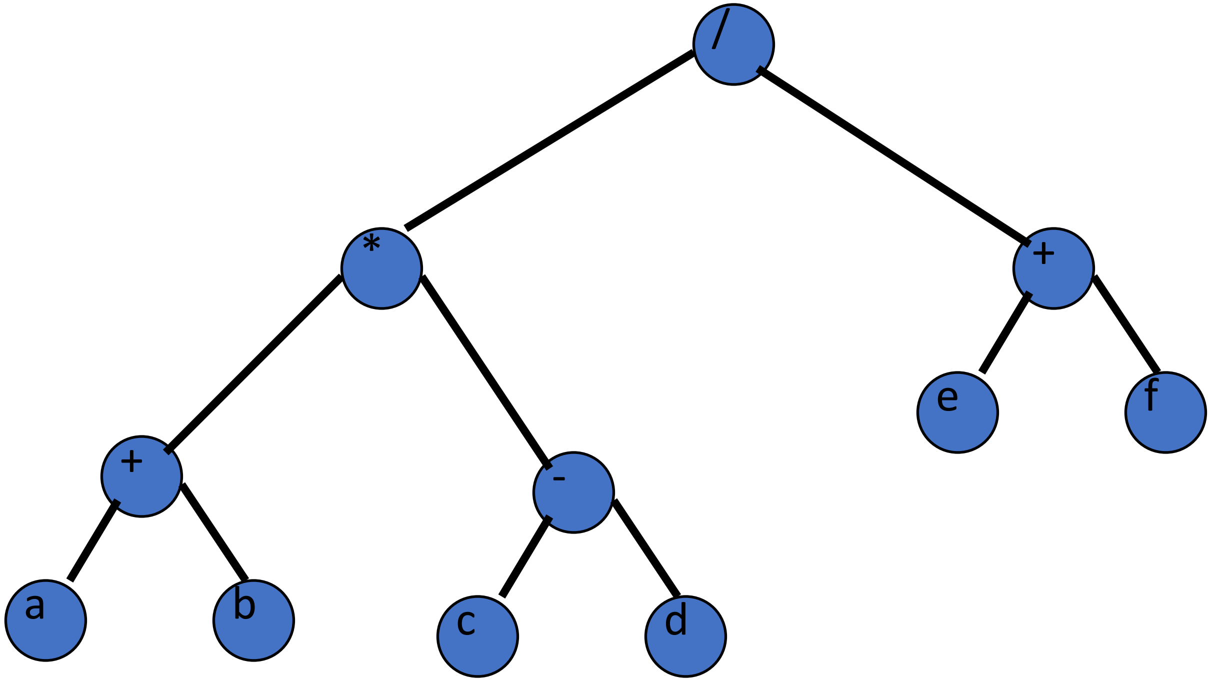
Binary SearchTree Form

(a + b) * (c – d) / (e + f)

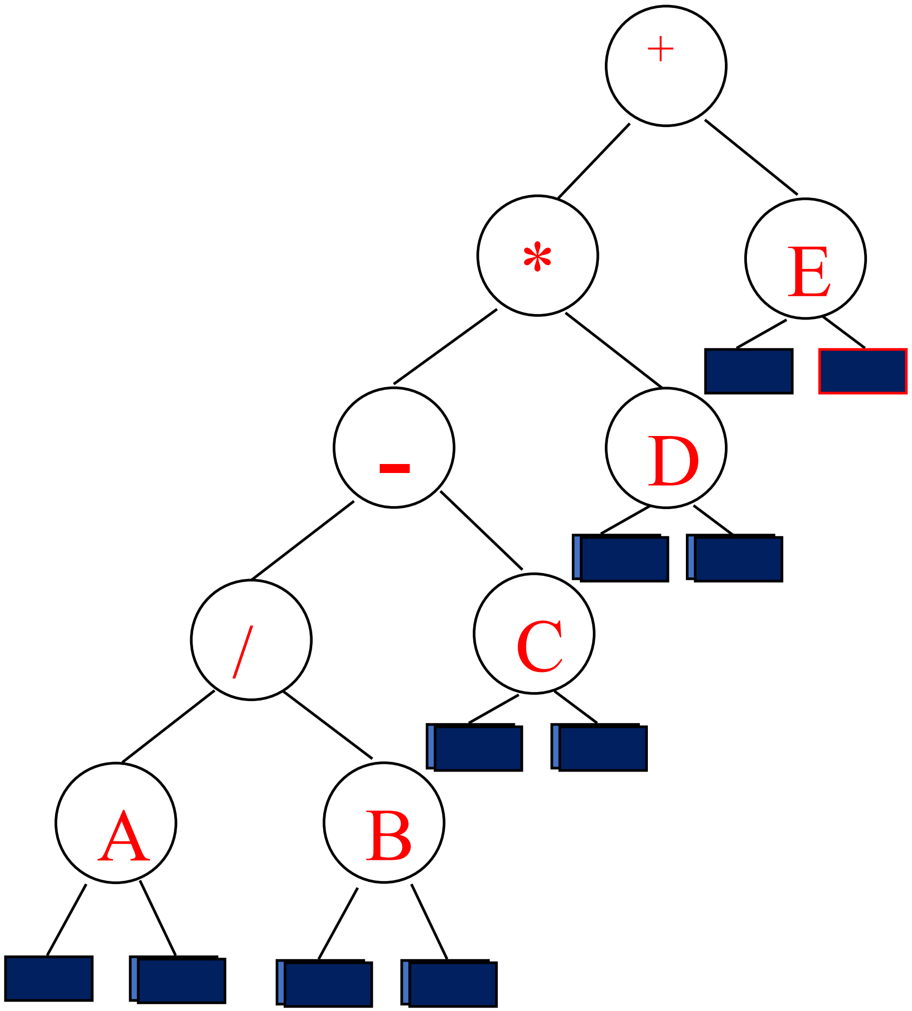


Merits Of Binary Search Tree Form

- Left and right operands are easy to visualize
- Code optimization algorithms work with the binary tree form of an expression
- Simple recursive evaluation of expression



Arithmetic Expression Using BST



inorder traversal – L V R

(A / B – C) * D + E

infix expression

preorder traversal – V L R

+ * - / A B C D E

prefix expression

postorder traversal - L R V

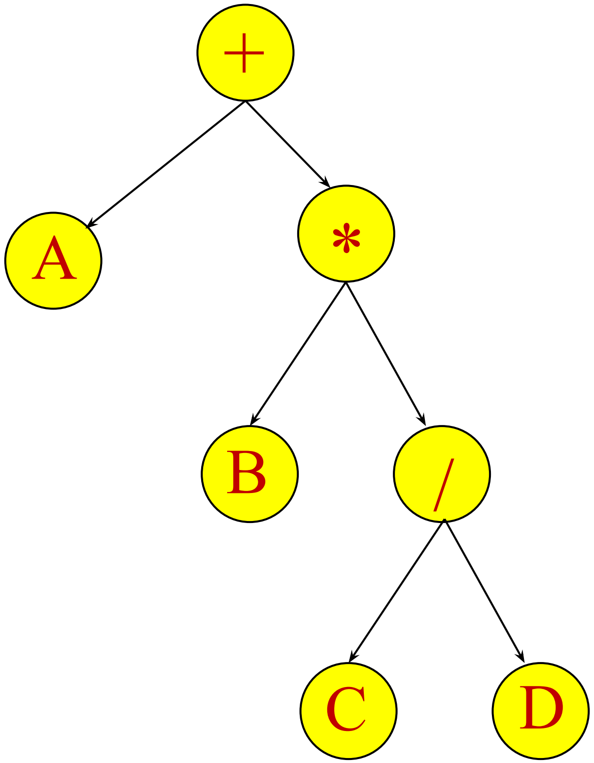
A B / C - D * E +

postfix expression

Arithmetic Expression Using BST

A + (B * (C / D))

- Preorder:
  - \+ A * B / C D
- Postorder:
  - A B C D / * +
- Inorder:
  - A + B * C / D



Preorder Implementation

```
Struct node
{ int key;
  leftchild *node;
  rightchild *node;
}
void preorder(node * t)
{   node * ptr;
    if (t!=NULL)
     { cout << t->key;
       preorder(t->leftchild);
       preorder(t->rightchild);
      }
     return;
}
```

Postorder Implementation

```
Struct node
{ int key;
  leftchild *node;
  rightchild *node;
}
void postorder(node * t)
{  node * ptr;
   if (t!=NULL)
    { postorder(t->leftchild);
      postorder(t->rightchild);
      cout<<t->key;
     }
    return;
}
```

Inorder Implementation

```
Struct node
{ int key;
  leftchild *node;
  rightchild *node;
}
void inorder(node * t)
{  node * ptr;
   if (t!=NULL)
    { inorder(t->leftchild);
      cout<<t->key;
      inorder(t->rightchild);
     }
    return;
}
```

Types of Trees:

Balanced Binary Search Tree (BST)

Balanced Binary Search Trees

- Height is O(log n), where n is the number of elements in the tree
- AVL (Adelson-Velsky and Landis) trees
- Red-Black trees
- get, put, and remove take O(log n) time

Balanced Binary Search Trees

- Indexed AVL trees
- Indexed red-black trees
- Indexed operations also take O(log n) time

Balanced Search Trees

- Weight balanced binary search trees
- 2-3 & 2-3-4 trees
- AA trees
- B-trees
- BBST
- etc…

<u>Types of Trees:</u>
AVLTree

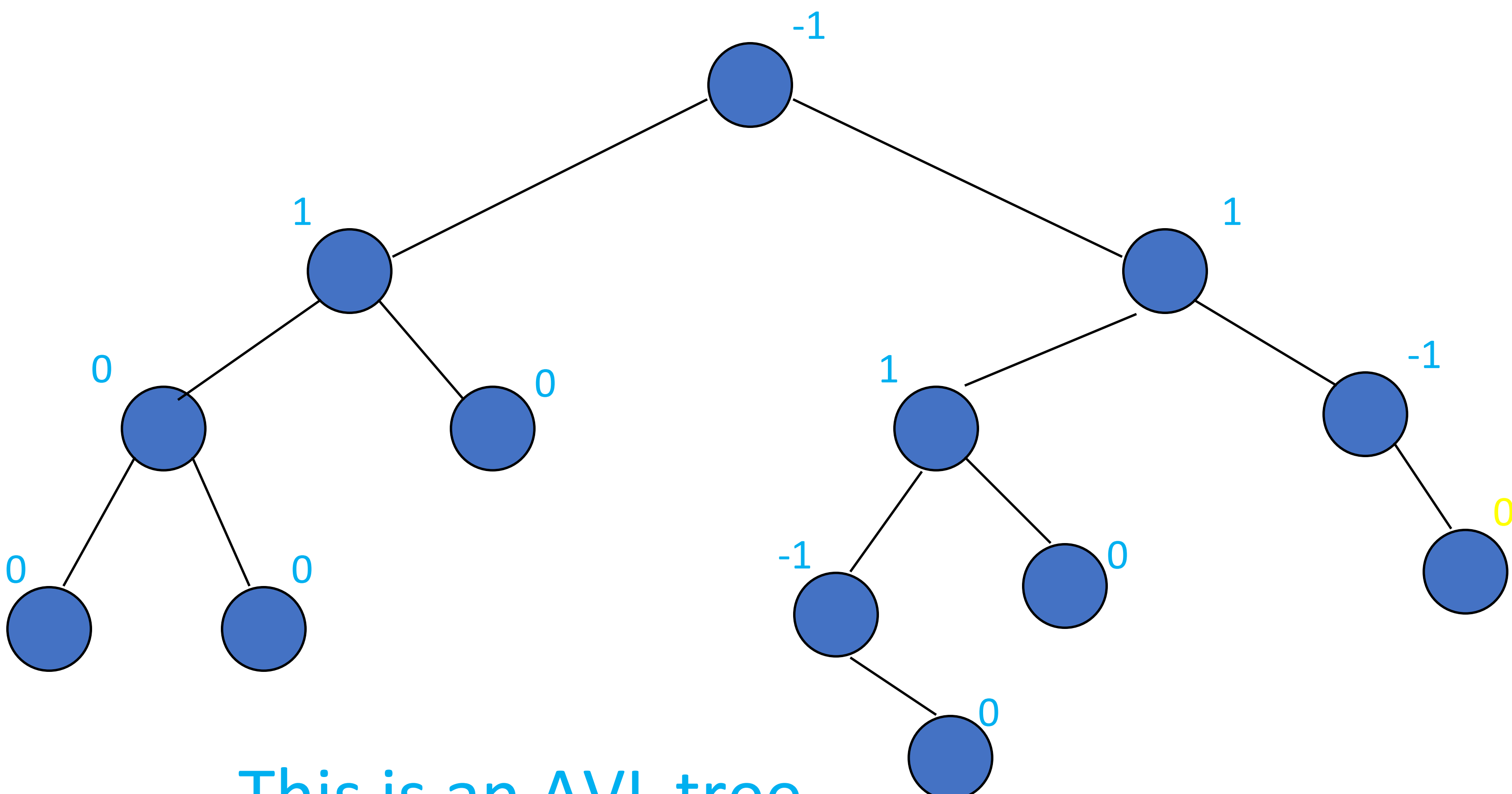
AVL Tree

- binary tree
- AVL (Adelson-Velsky and Landis) trees
- for every node x, define its **balance factor**

  balance factor of x = height of left subtree of x

  - height of right subtree of x
- balance factor of every node x is -1, 0, or 1

Balance Factors



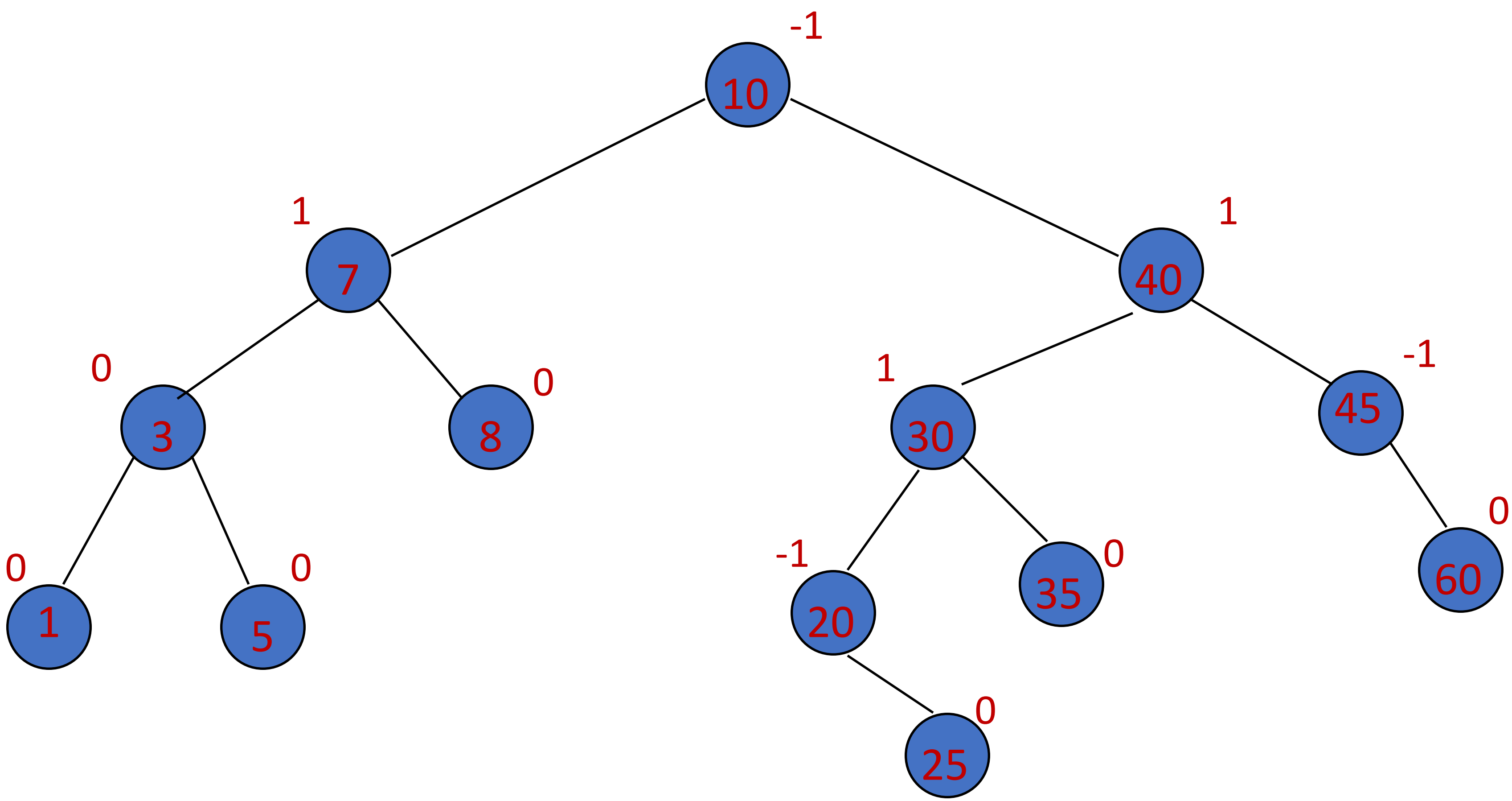
This is an AVL tree.

Height

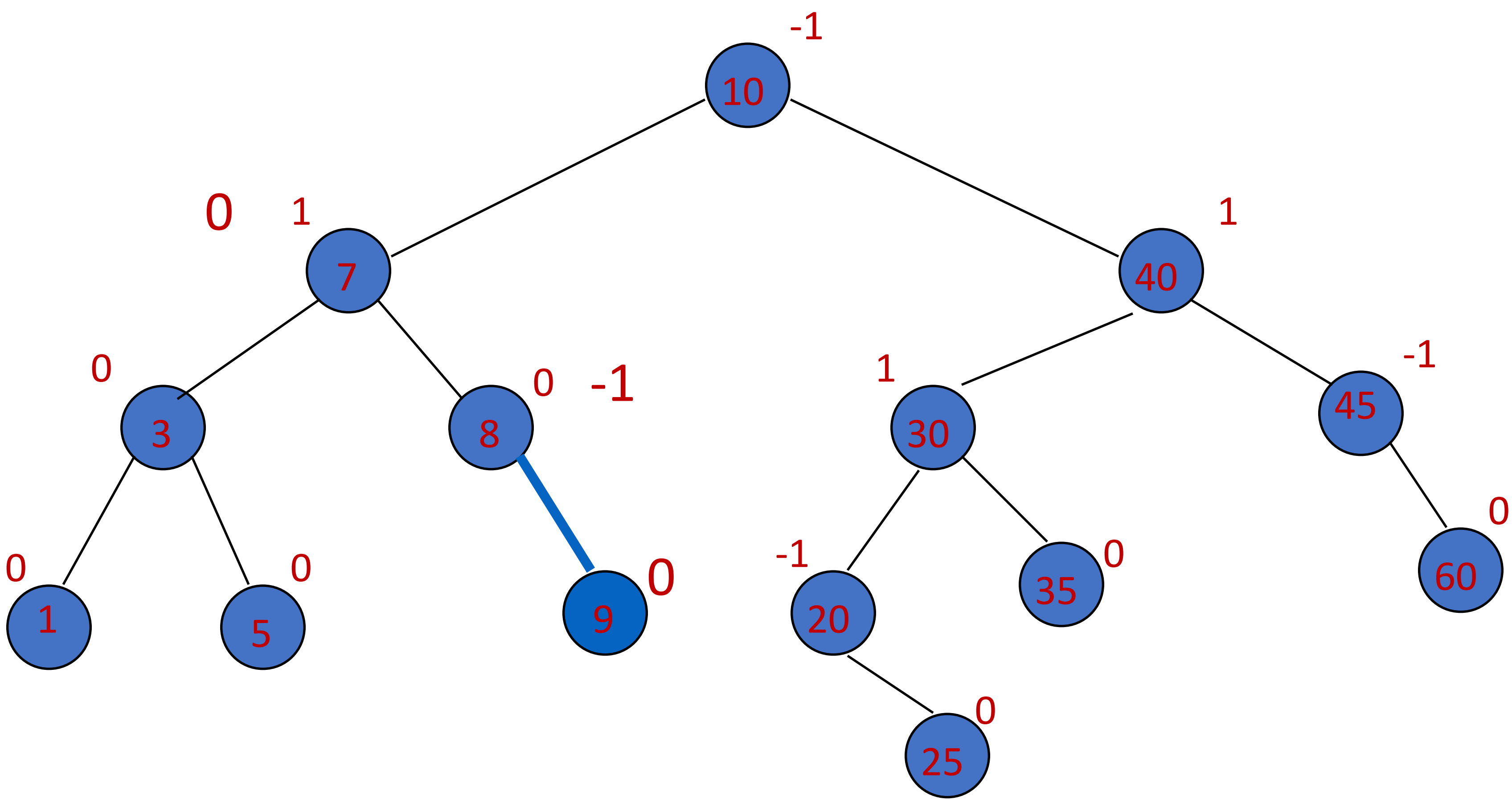
The height of an AVL tree that has n nodes is at most $1.44 \log_2 (n+2)$

The height of every n node binary tree is at least $\log_2 (n+1)$

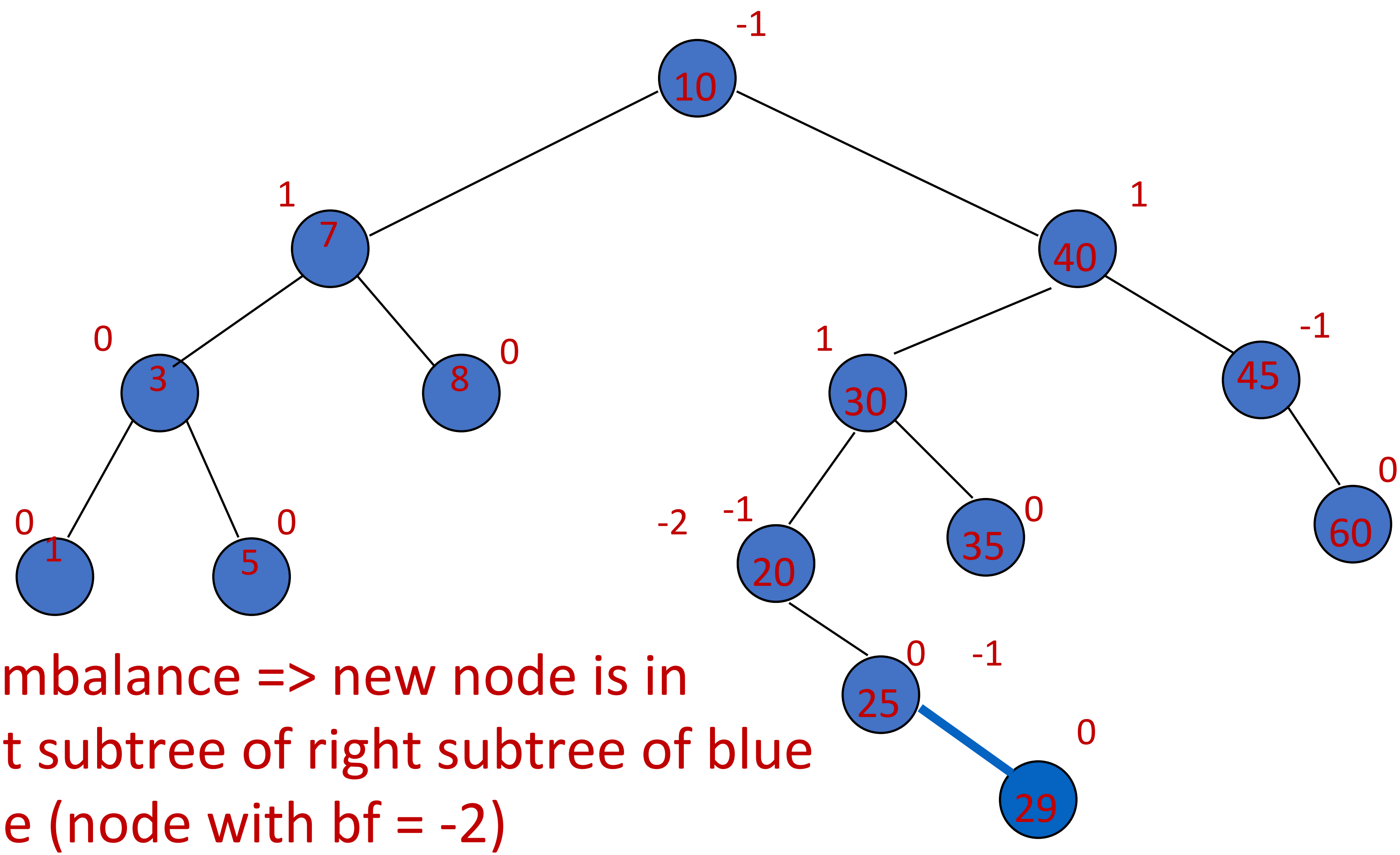
AVL Search Tree



put(9)

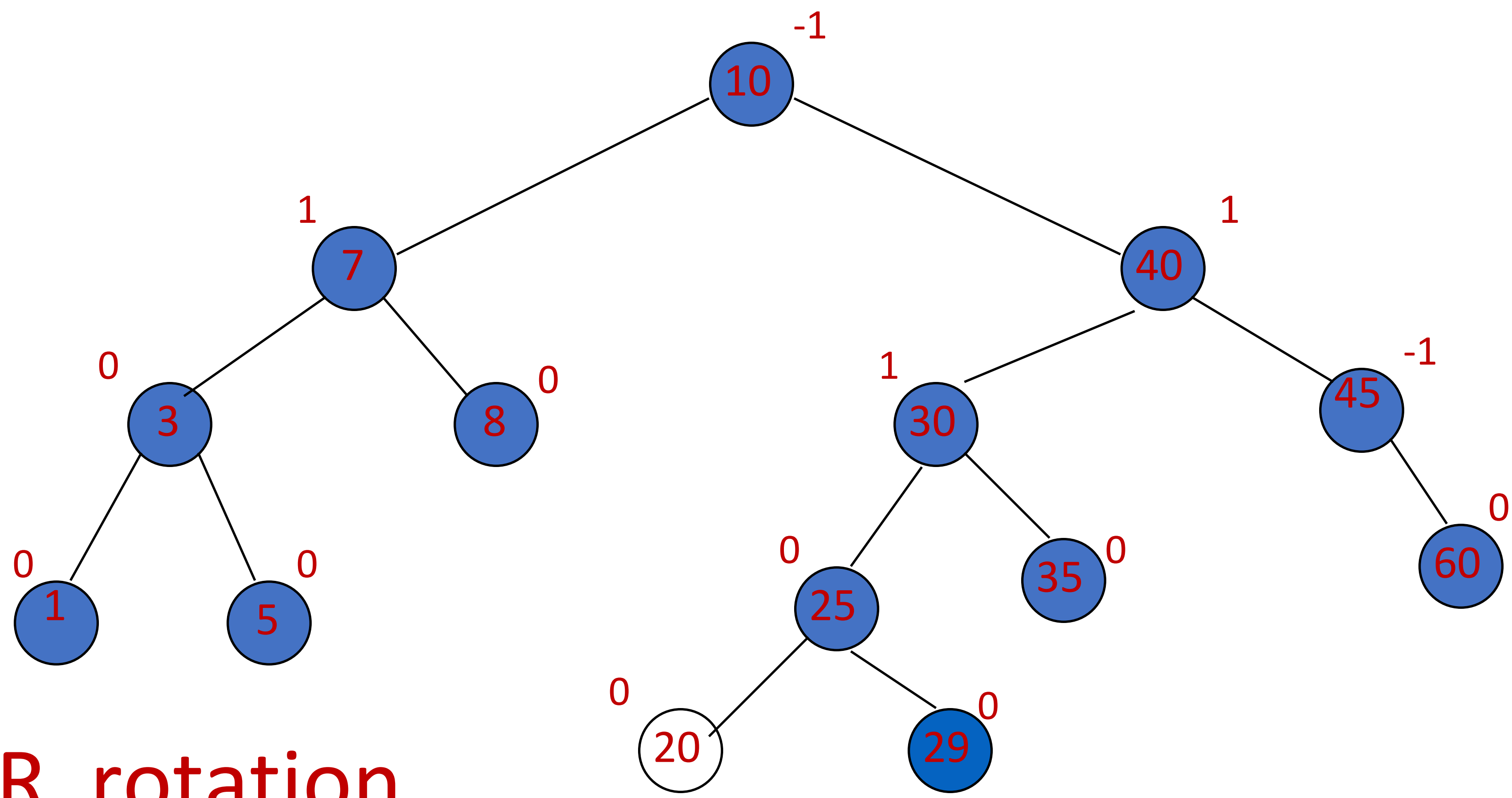


put(29)



RR imbalance => new node is in right subtree of right subtree of blue node (node with bf = -2)

put(29)



RR rotation

AVL Rotations

- RR
- LL
- RL
- LR

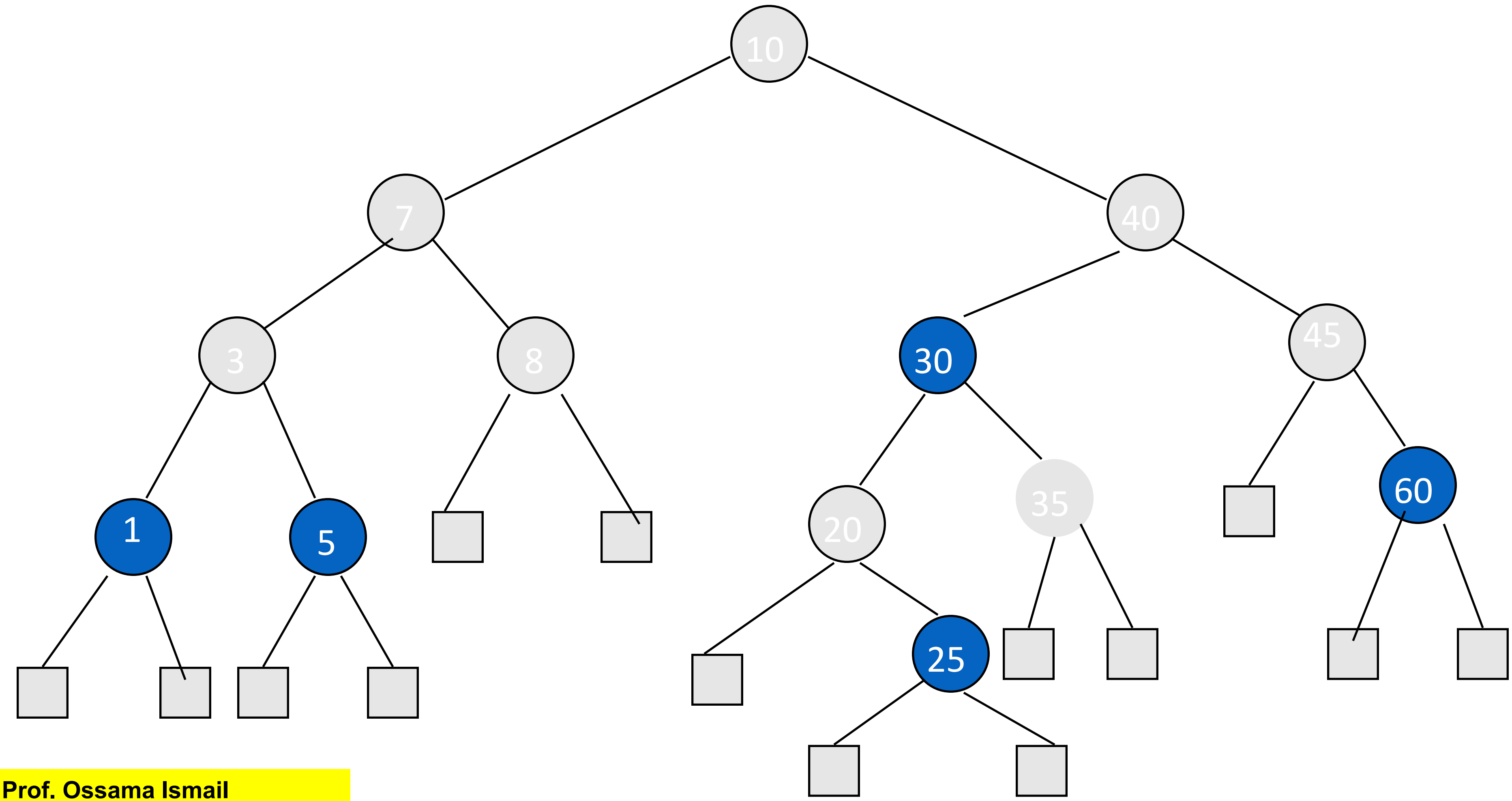
Types of Trees:
Red-Black Tree

Red-Black Trees

Colored Nodes Definition

- Binary search tree
- Each node is colored red or black
- Root and all external nodes are black
- No root-to-external-node path has two consecutive red nodes
- All root-to-external-node aths have the same number of black nodes

Example Red Black Tree

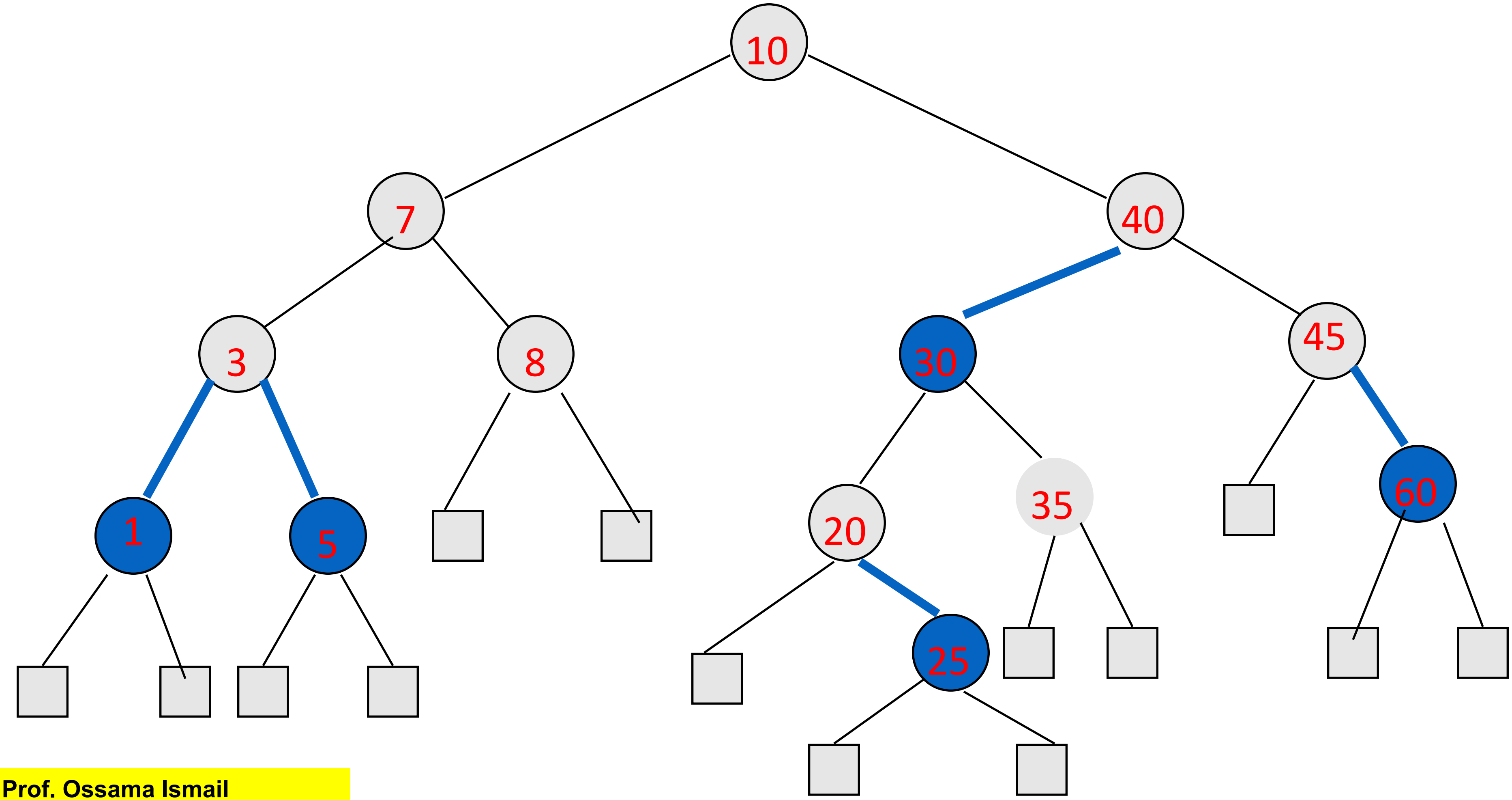


Red Black Trees

Colored Edges Definition

- Binary Search Tree
- Child pointers are colored red or **black**
- Pointer to an external node is black
- No root to external node path has two consecutive red pointers
- Every root to external node path has the same number of black pointers

Example Red Black Tree



Red Black Tree

- The height of a red black tree that has n (internal) nodes is between $\log_2(n+1)$ and $2\log_2(n+1)$
- java.util.TreeMap => red black tree

Questions ????