Data Structures & Algorithms

Adil M. Khan

Professor of Computer Science

Innopolis University
a.khan@innopolis.ru

Recap

- What is sorting?
- Why learn about sorting?
- Properties of sorting algorithms
- Various sorting algorithms and their time complexities

Objectives

- What is a Tree (as a data structure)?
- Learn about different types of trees and the associated properties, definitions and terminologies
- Special emphasis on Binary Search Trees
  - How does a BST work?
  - How to traverse in a BST?
  - Time complexity of BST operations (search, insert, delete)

Tree

- A tree combines the advantages of two other data structures:
  - An ordered array
  - A linked list

Ordered Array

- Quick to search for a particular element, using binary search

- On the other hand, insertions are slow

  1. First we need to find the position where the element will go

  2. Then move all the objects with greater keys up one space to make the room

Linked List

- Insertions and deletions are quick
    - Simply requires changing a few constant number of references
- Finding a particular element is slow
    1. Must start at the beginning of the linked list
    2. Visit each element until you find the one you’re looking for

What We Desire?

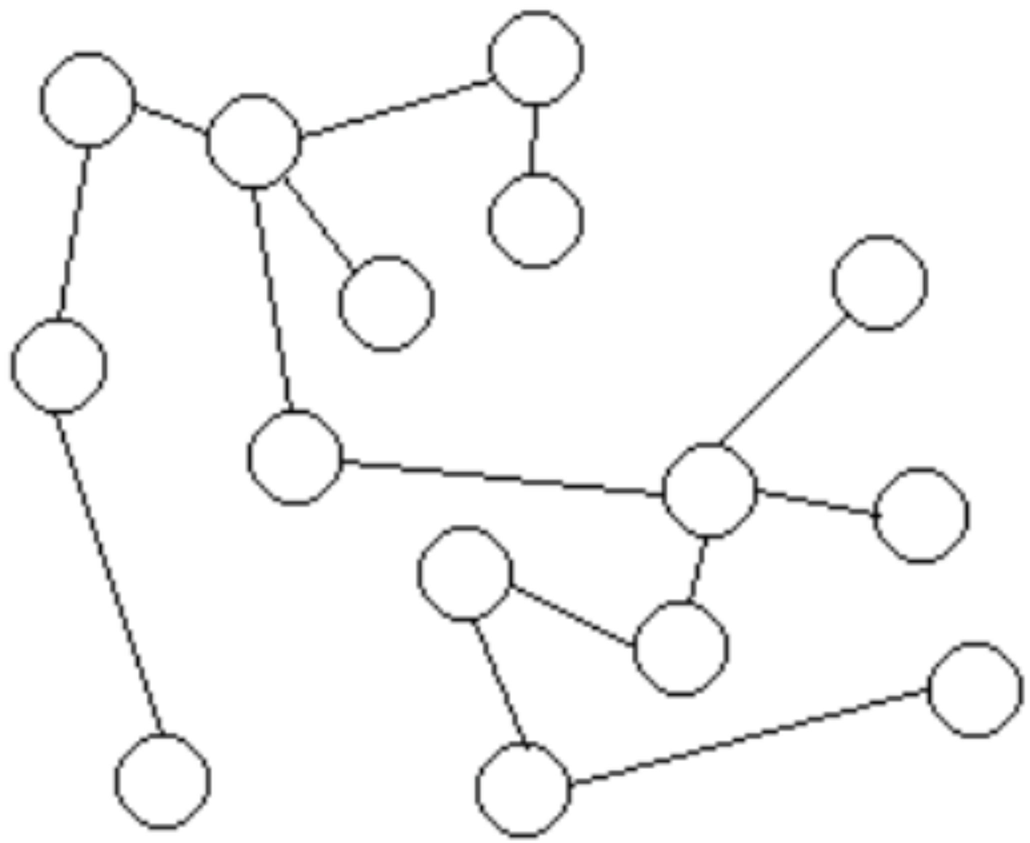
- It would be nice to have a data structure
  - With quick insertions and deletions of a linked list
  - And, the quick searching of an ordered array

Trees to Rescue

- Trees provide both these characteristics
- Our main focus will be a binary search tree

Tree

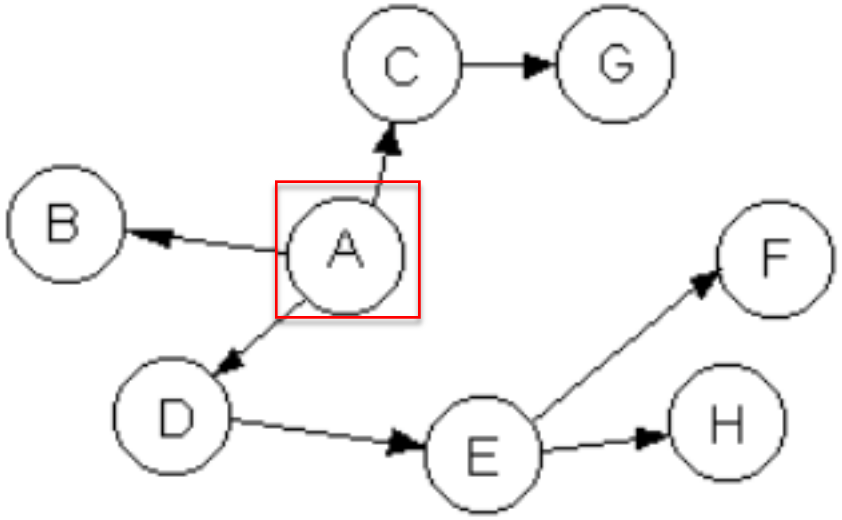
- Consists of nodes connected by edges



Oriented Trees

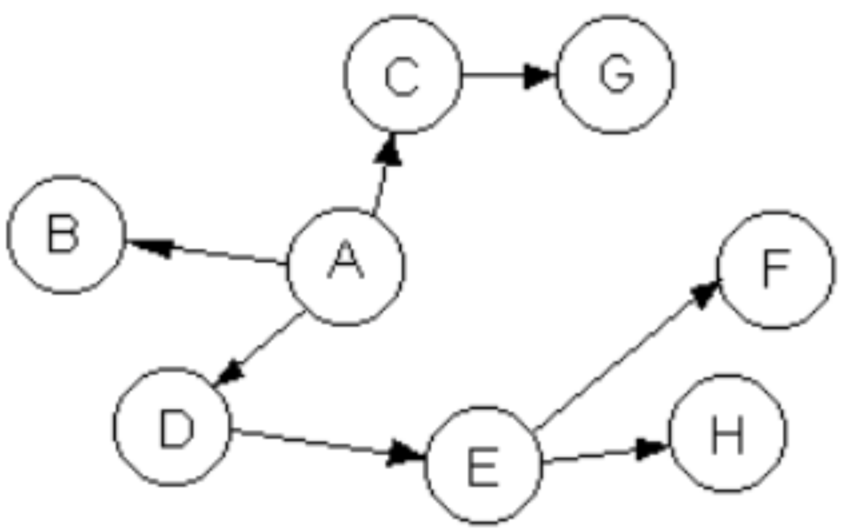
Oriented Trees

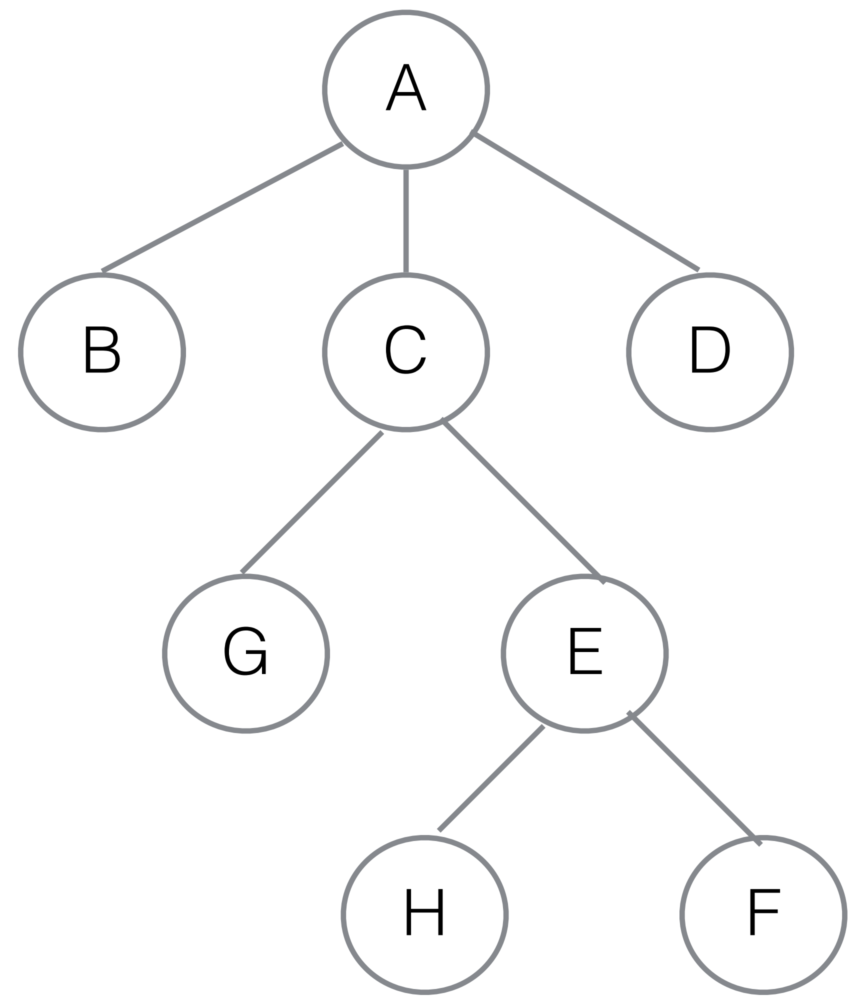
- A tree used to represent a hierarchical data.
- All edges are directed outward from a distinguished node called the root node



Oriented Trees

- Usually drawn with the root at the top,
- all edges pointing downward, the arrows are thus redundant and are often omitted.

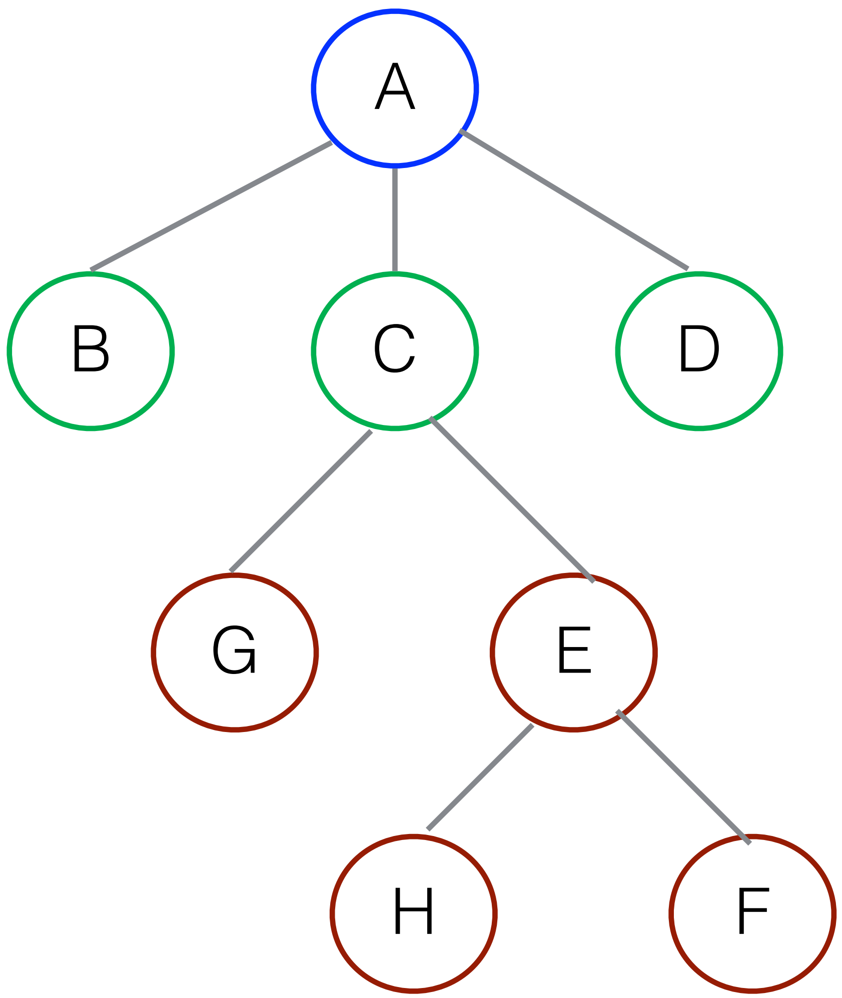




Tree Terminology

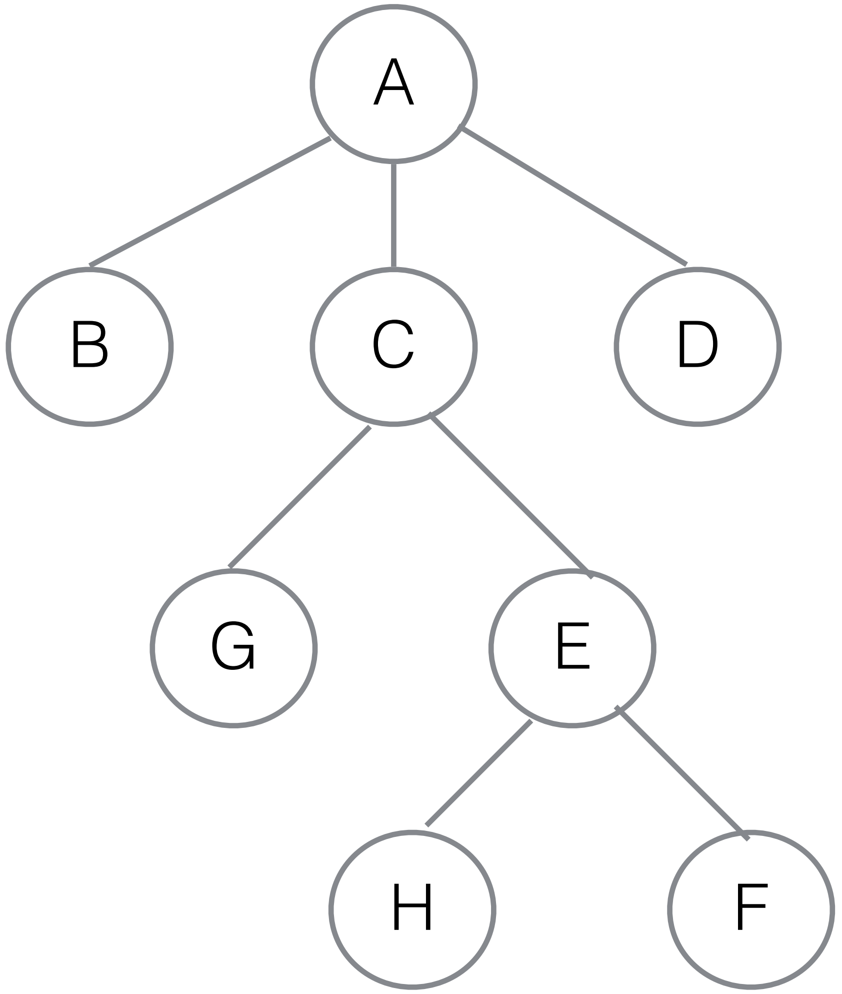
Node: Any object or value stored in the tree represents a node.

In the figure, the root and all of its children and descendants are nodes.



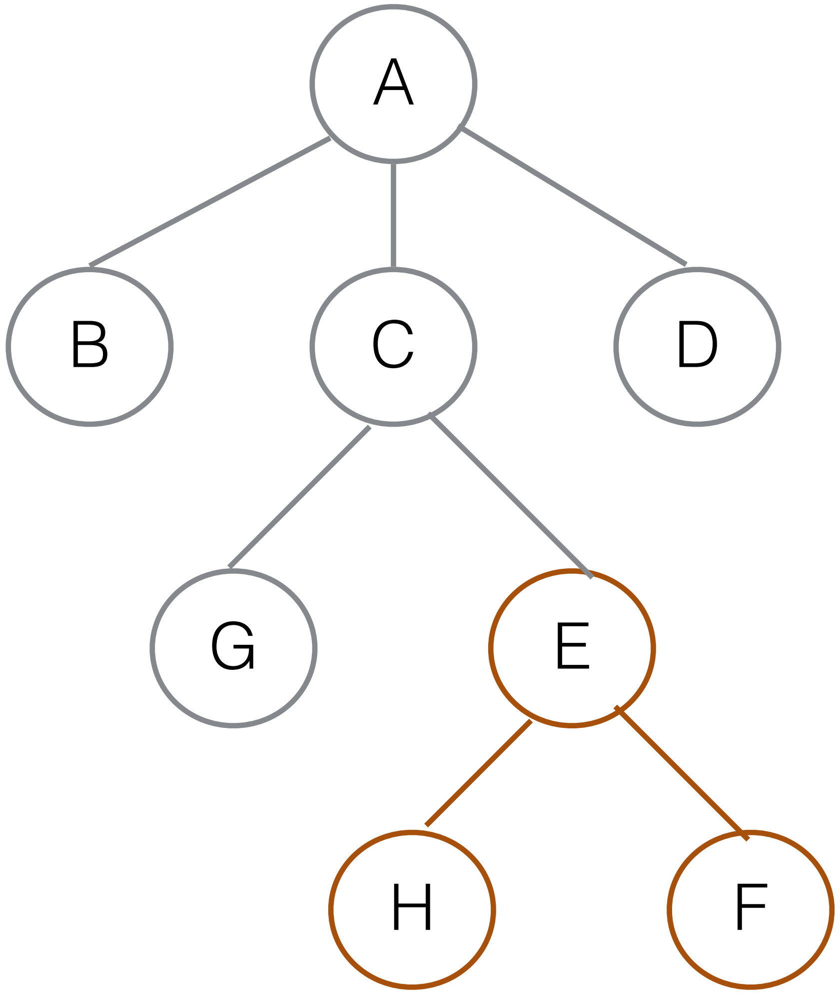
Tree Terminology

Parent: A parent node is any node which has 1...n child nodes.



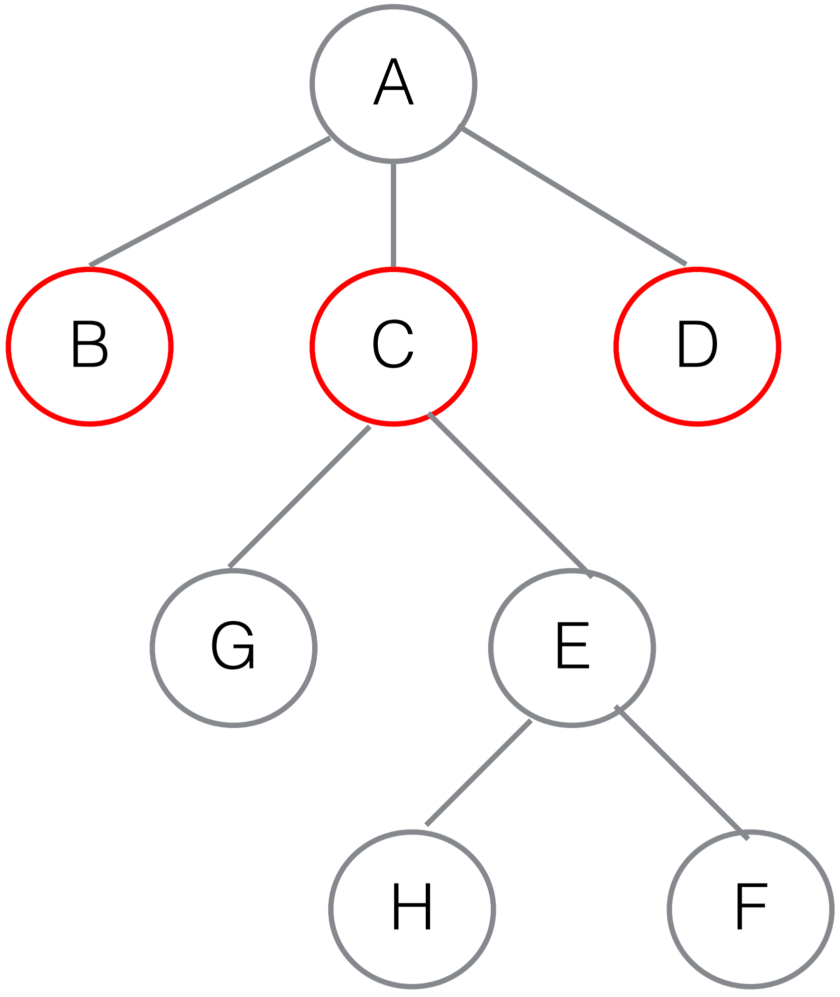
Tree Terminology

Child: Any node other than the root node is a child to one (and only one) other node.



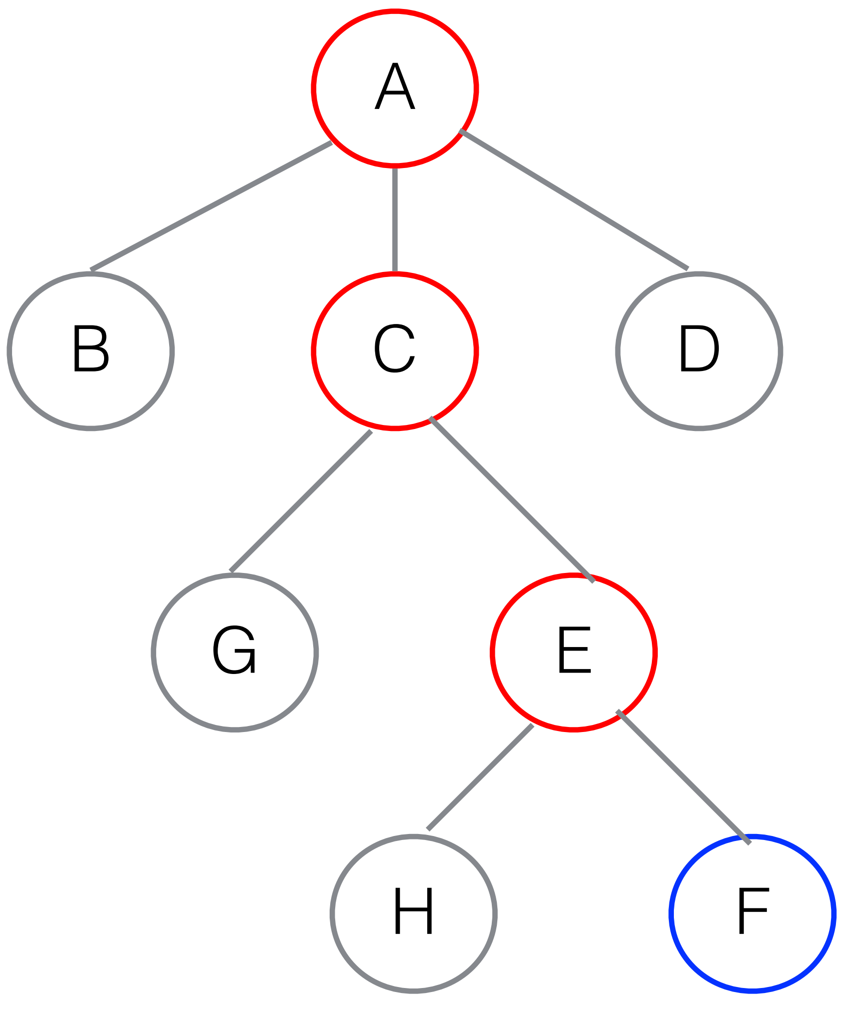
Tree Terminology

Siblings: Siblings represent the collection of all of the child nodes to one particular parent.



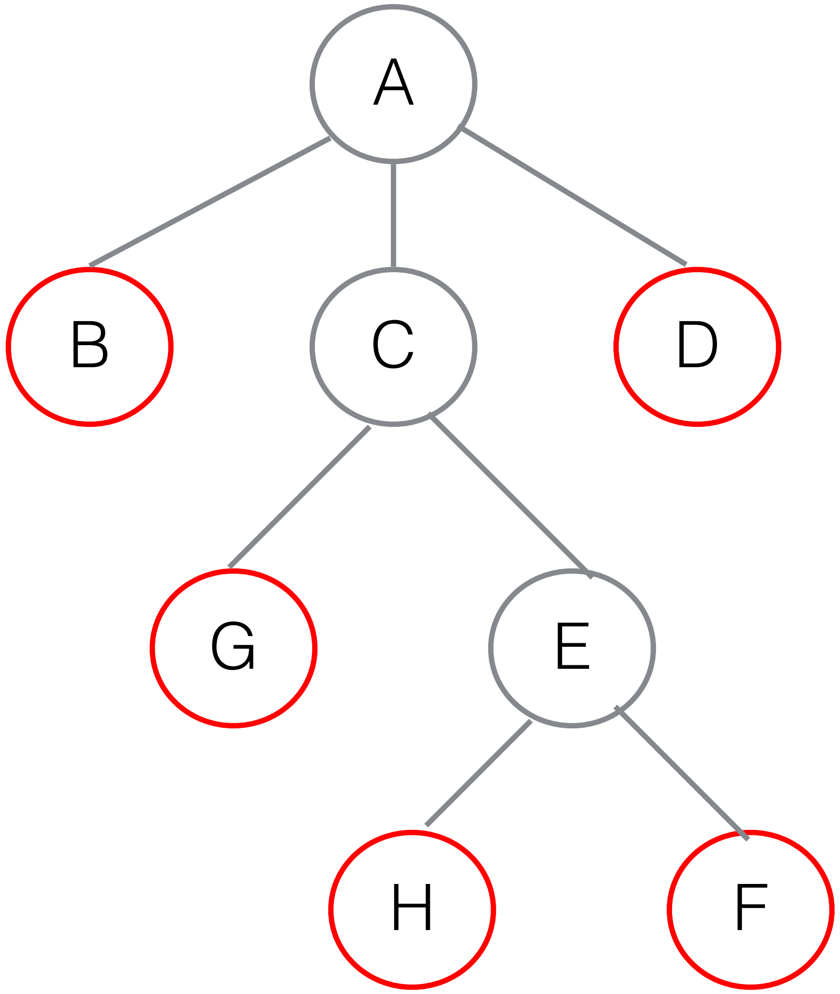
Tree Terminology

Ancestor: The ancestors of a node are any of the nodes that can be reached from that node following edges toward the root node.



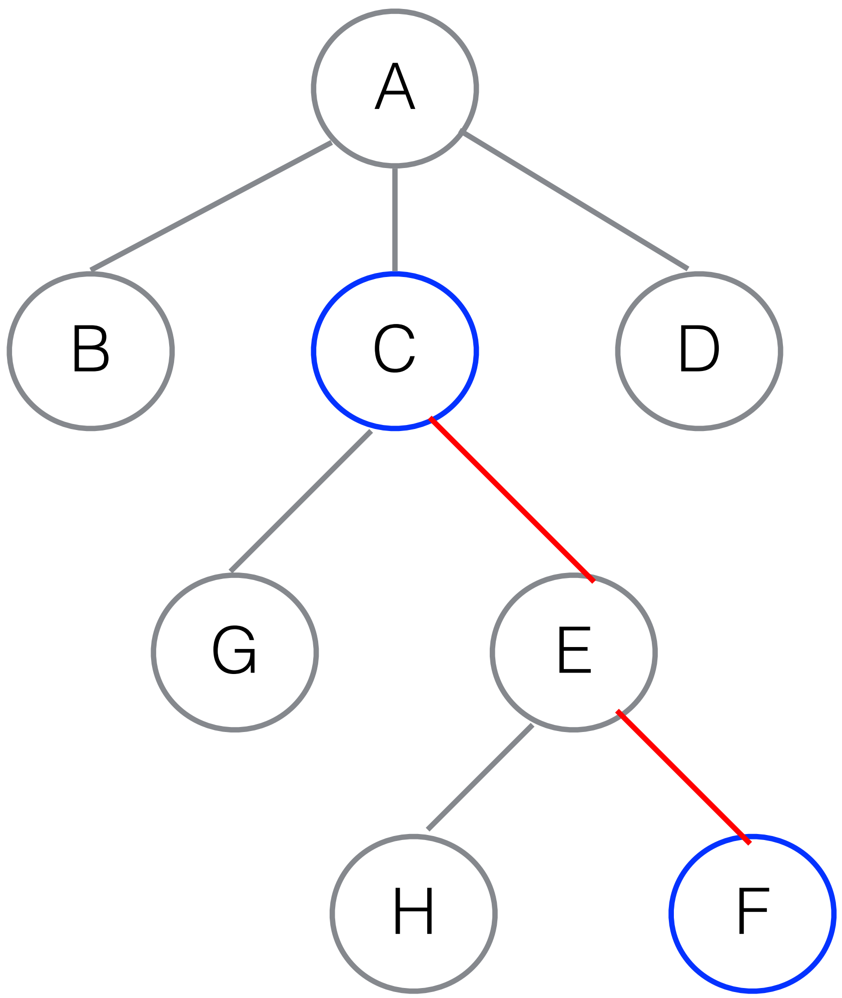
Tree Terminology

Leaf: Any node that has no child nodes is called a leaf.



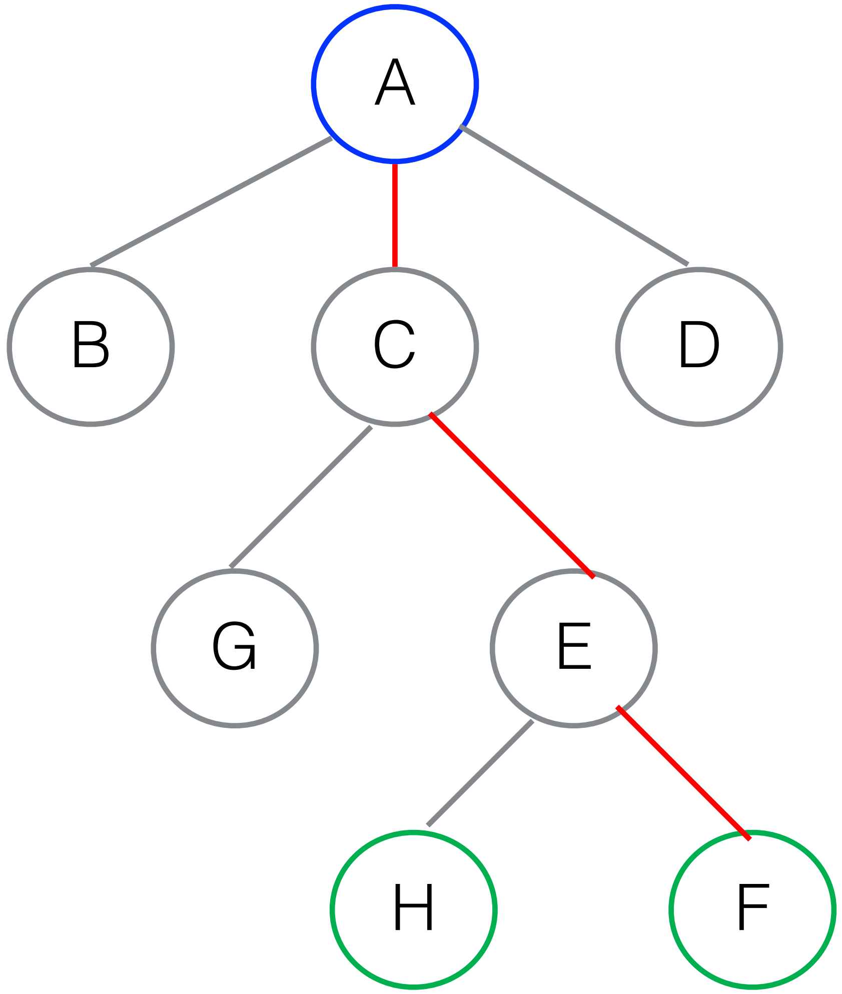
Tree Terminology

Path: A path is described as a list of edges between a node and one of its descendants.



Tree Terminology

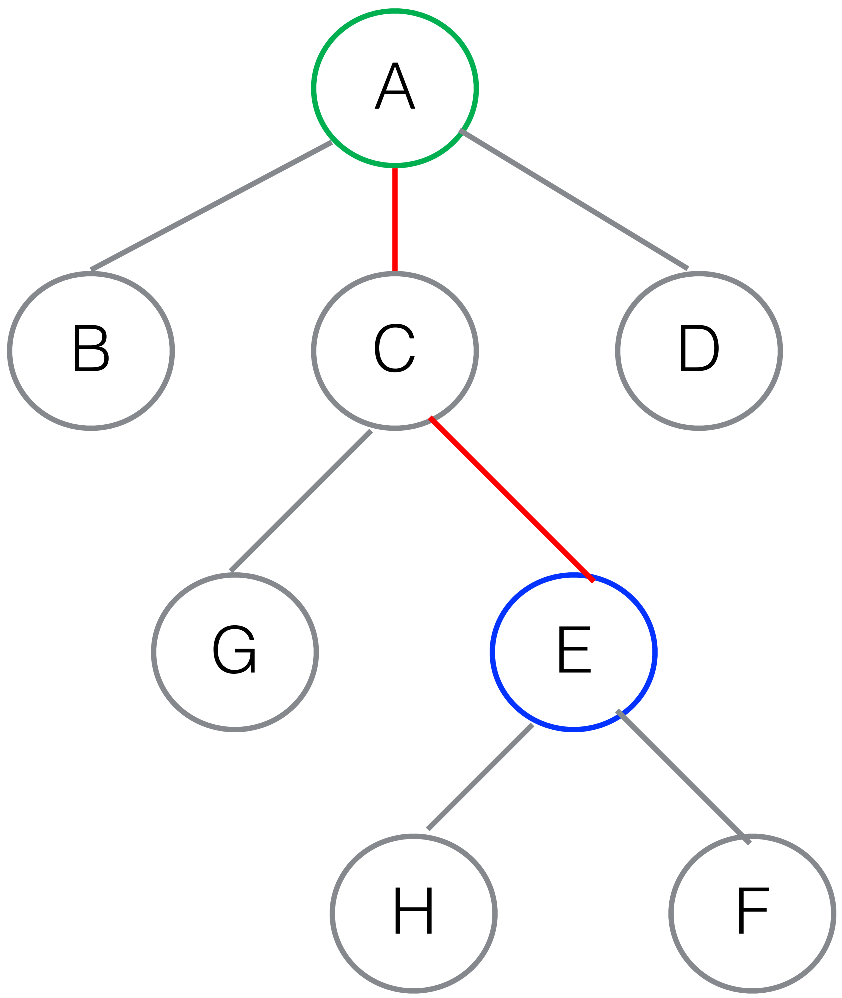
Height of tree: The height of a tree represents the number of edges between the root node and the leaf that is farthest from the root node.



Tree Terminology

Depth: The number of edges between that node and the root node represents the depth of a node.

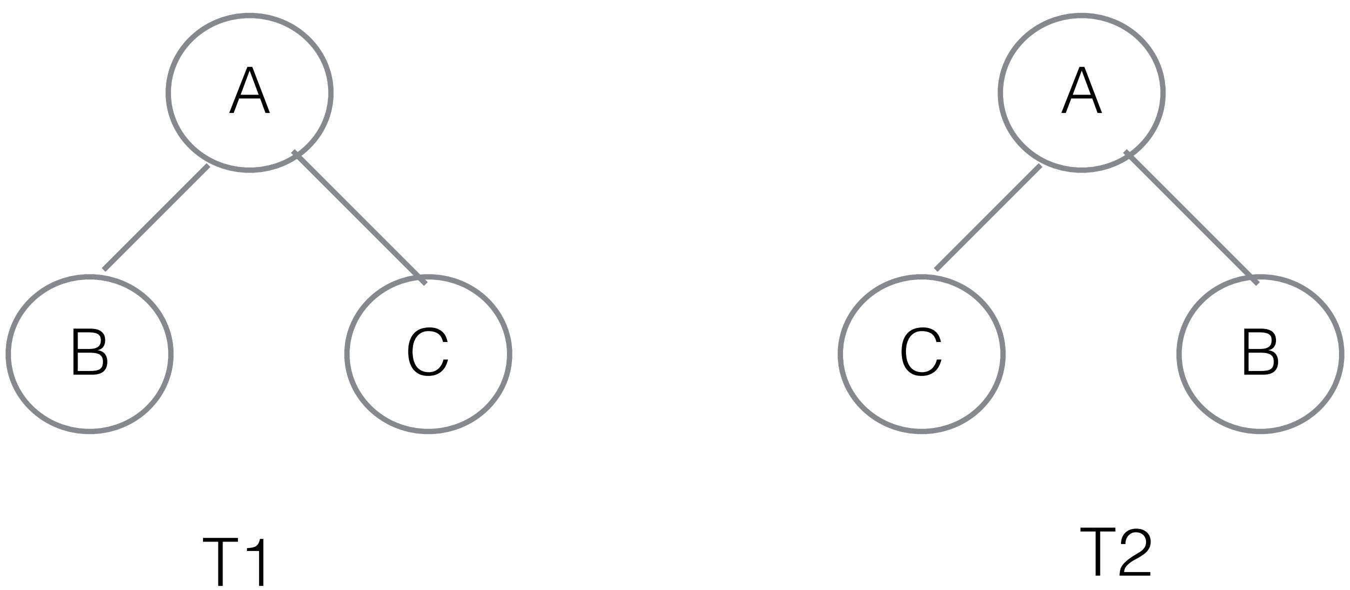
The root node, therefore, has a depth equal to zero.



Ordered Trees

Ordered Trees

- An oriented tree in which the children of a node are somehow **ordered**.



If T1 and T2 are ordered trees then T1 =! T2,
otherwise T1 = T2

k-ary Trees

- An ordered tree in which the children of a node appear at distinct index positions in $0..k-1$

- Maximum number of children for a node is k

Types of k-ary Trees

- 2-ary Trees, known as **Binary Trees**
- 3-ary Trees, known as **Ternary Trees**
- 1-ary Trees, known as **Lists**

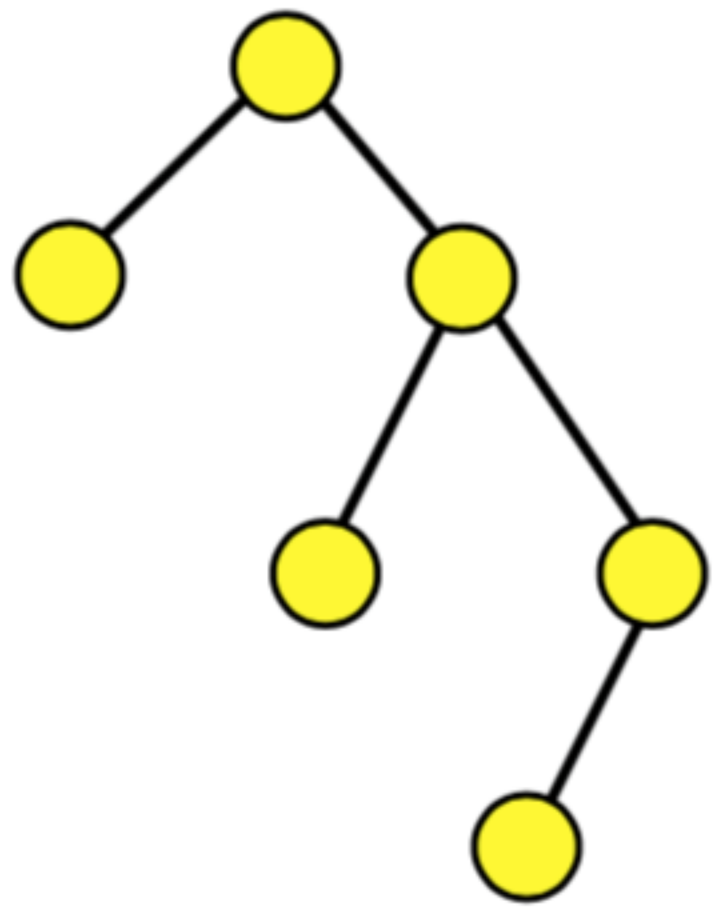
Binary Tree

Binary Tree

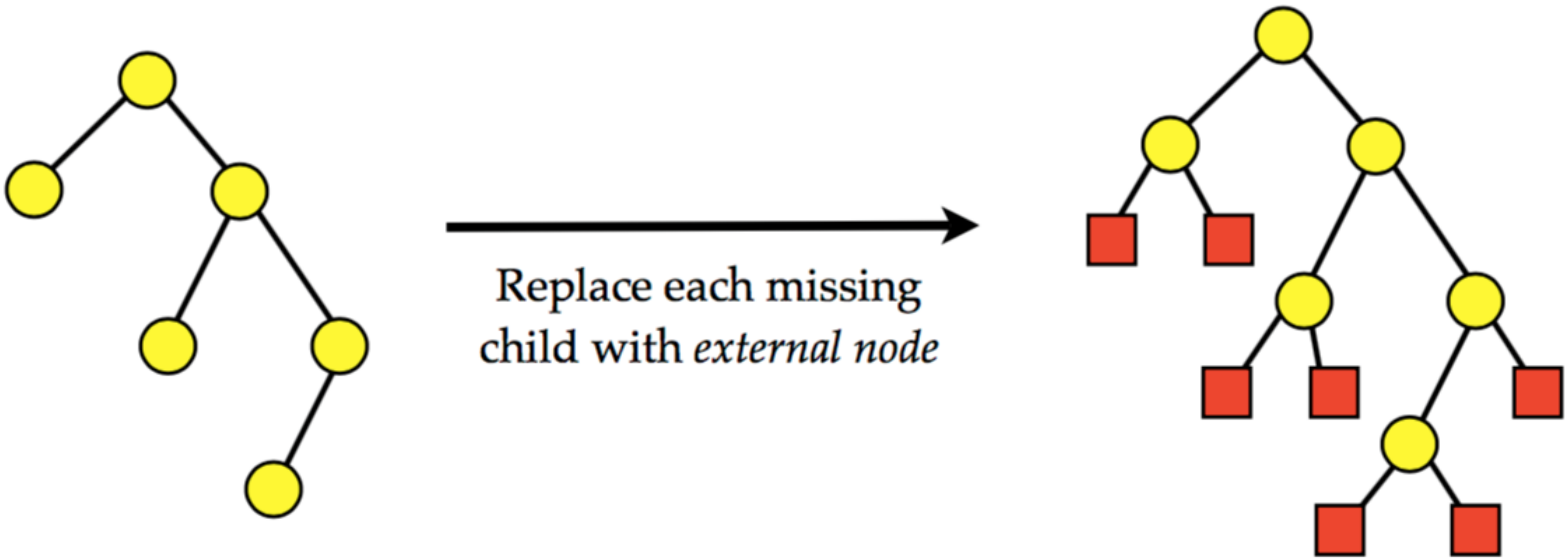
- A **binary tree** T of n nodes, $n \geq 0$,
  - ➢ is either empty, if $n = 0$
  - ➢ or consists of a root node u and two binary trees $u(1)\ and\ u(2)$ of n_1 and n_2 nodes, respectively,
  - ➢ such that $n = 1 + n_1 + n_2$

- We say that $u(1)$ is the **first or left subtree** of T, and $u(2)$ is the **second or right subtree** of T

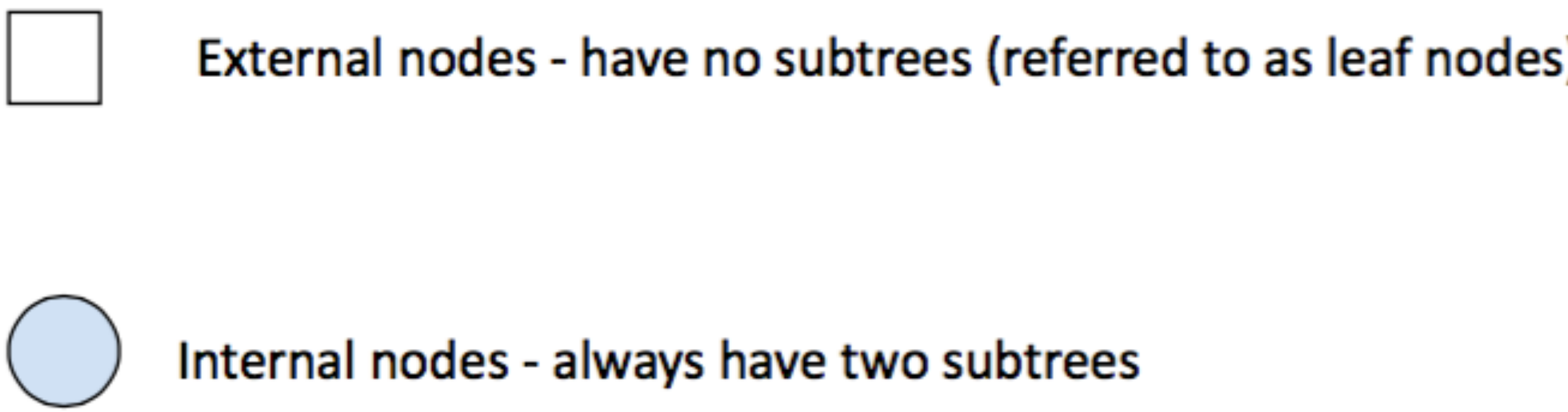
Simple Binary Tree



Extended Binary Tree

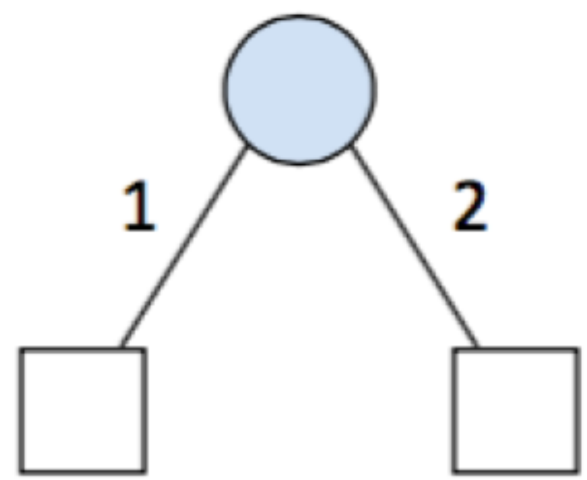


Extended Binary Tree



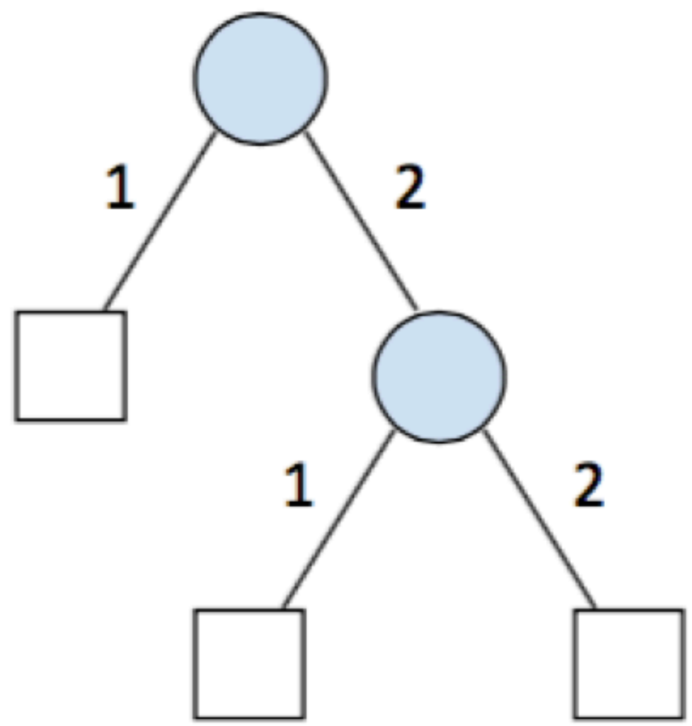
- You will see these symbols used for internal and external nodes
- External nodes can sometimes be omitted
- Implementation wise, external nodes are represented as NULL pointers

Extended Binary Tree



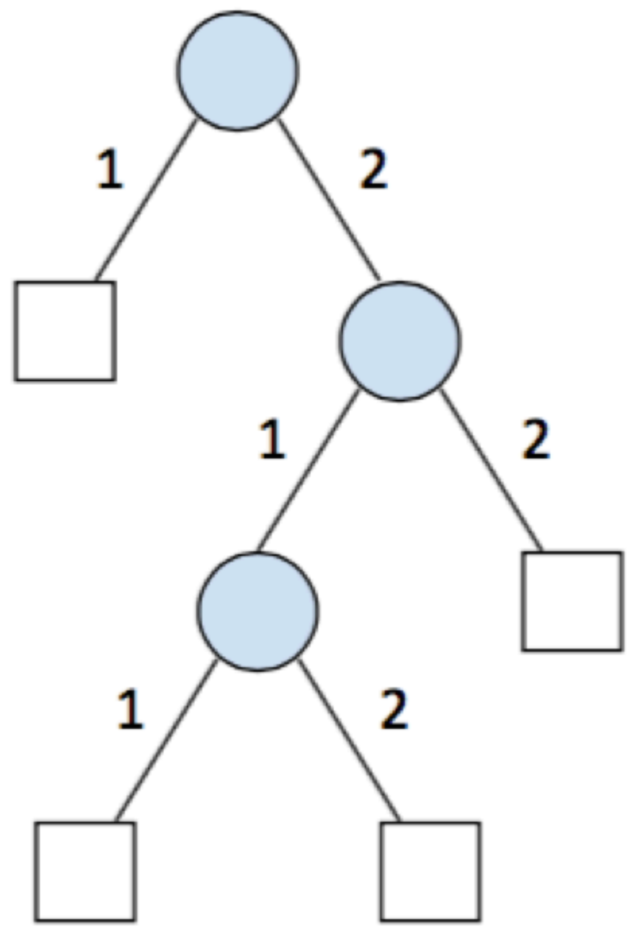
With1 Internal Node

Extended Binary Tree



With 2 Internal Nodes

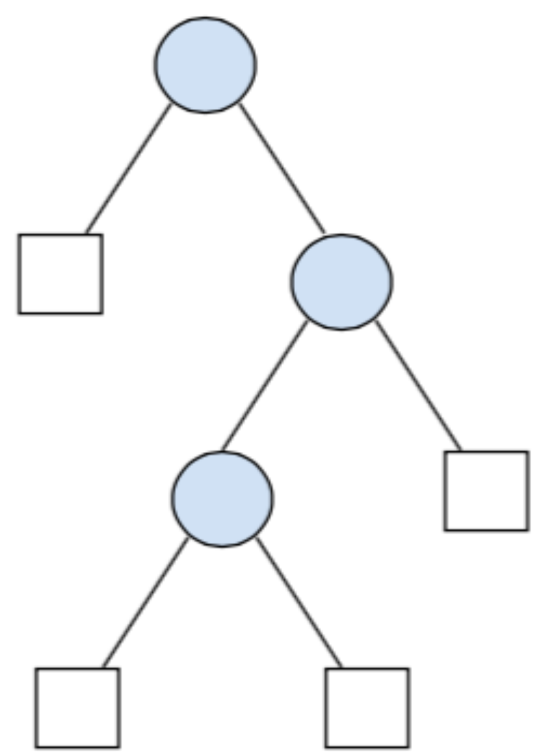
Extended Binary Tree



With 3 Internal Nodes

Number of External Nodes

- Let T be an extended binary tree with n internal nodes, $n \geq 0$,
- Then, the number of external nodes of T is $n + 1$



Proof: Number of External Nodes

- Every node has 2 children pointers, for a total of $2n$ pointers.

- Every node except the root has a parent, for a total of $n - 1$ nodes with parents.

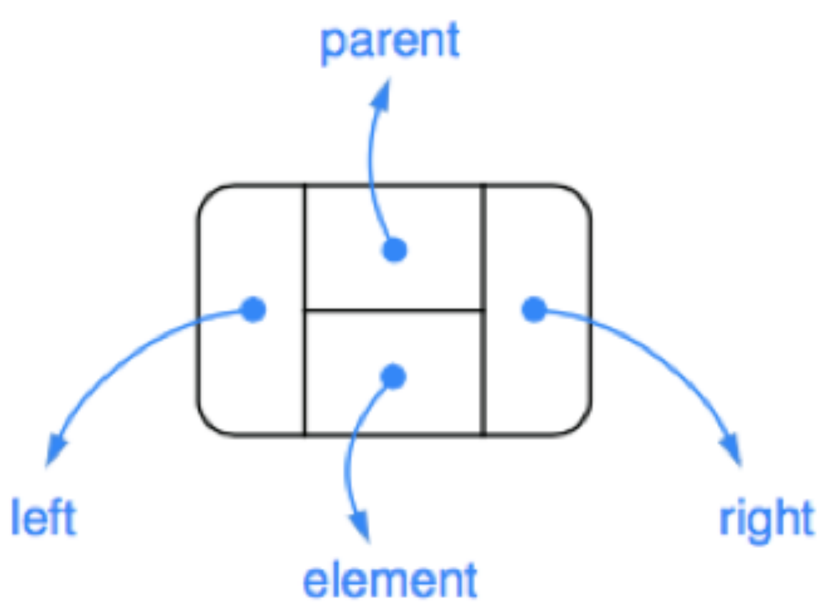
- These $n - 1$ parented nodes are all children, and each takes up 1 child pointer.

$$\text{(pointers)} - \text{(used child pointers)} = \text{(unused child pointers)}$$
$$2n - (n-1) = n + 1$$

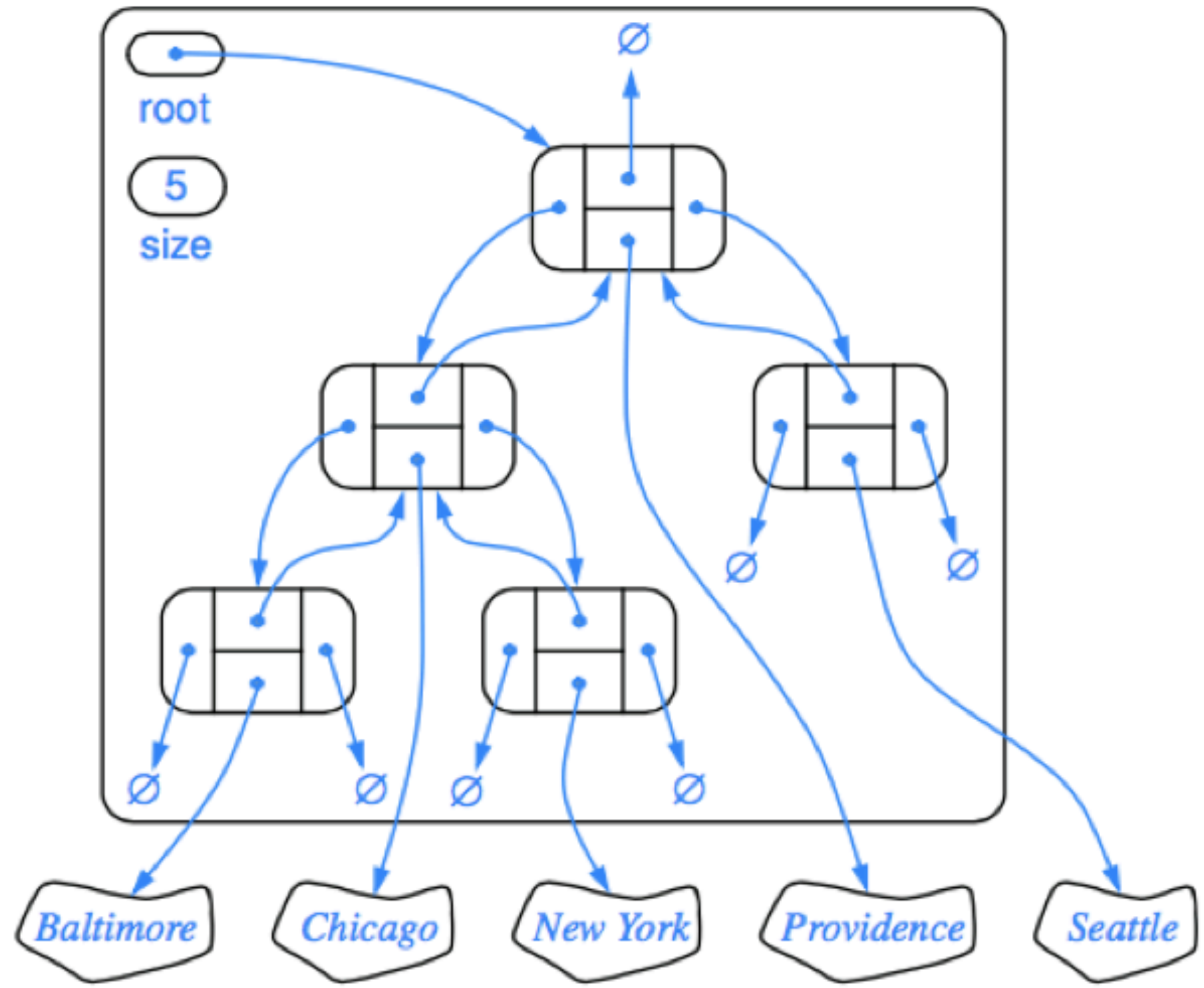
- Thus, there are $n + 1$ null pointers.

Binary Tree Representations

Linked Structure



(a)

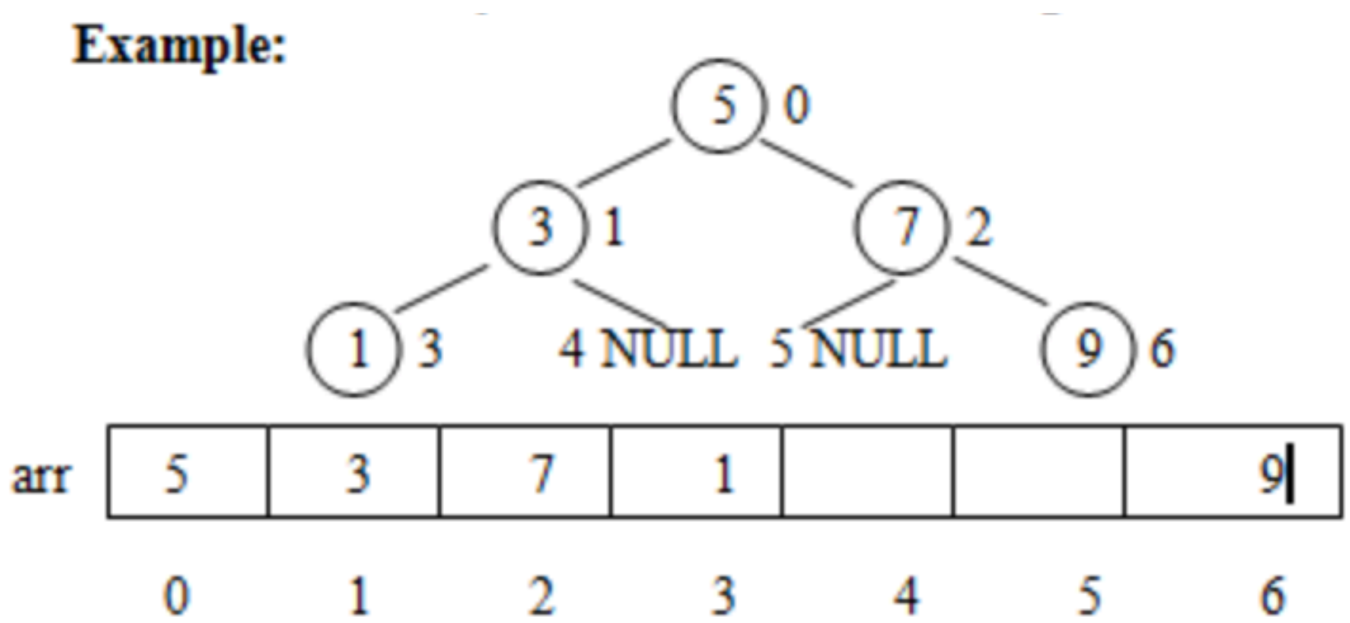


(b)

Array Structure

- Requires a mechanism for numbering the positions of T
- For every position p of T , let $f(p)$ be the integer defined as follows
    - If p is the root of T, then $f(p) = 0$
    - If p is the left child of position q, then $f(p) = 2f(q) + 1$.
    - If p is the right child of position q, then $f(p) = 2f(q) + 2$.

Array Structure



If p is the root of T, then $f(p) = 0$

If p is the left child of position q, then $f(p) = 2f(q) + 1$.

If p is the right child of position q, then $f(p) = 2f(q) + 2$.

Tree Traversal Algorithms

Tree Traversal Algorithms

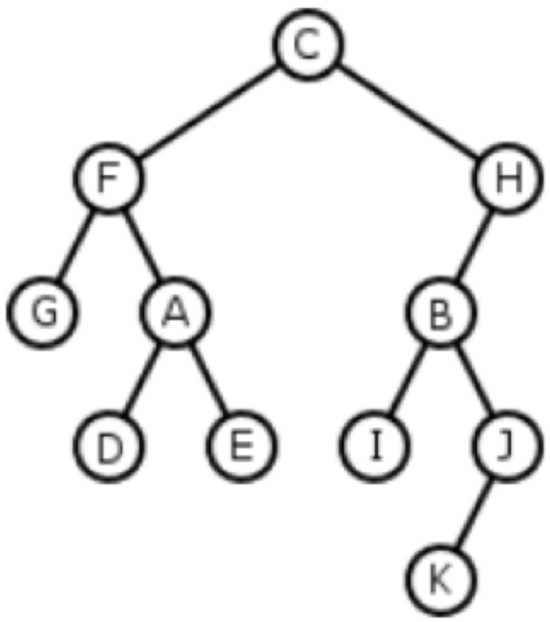
- A way of **accessing** or **visiting** all the nodes of T

Tree Traversal Algorithms

- A way of **accessing** or **visiting** all the nodes of T
- Preorder Traversal –
  - Visit the node, Preorder Left, Preorder Right
- Postorder Traversal
  - Postorder Left, Postorder Right, Visit the node
- Inorder Traversal
  - Inorder Left, Visit the node, Inorder Right

Tree Traversals

Visit = Print The Node Value



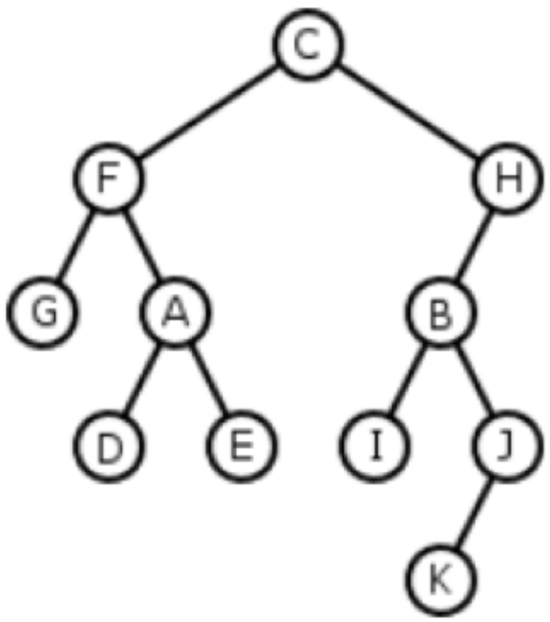
Output: C F G A D E H B I J K

Preorder Traversal

Visit the node, Preorder Left, Preorder Right

Tree Traversals

Visit = Print The Node Value



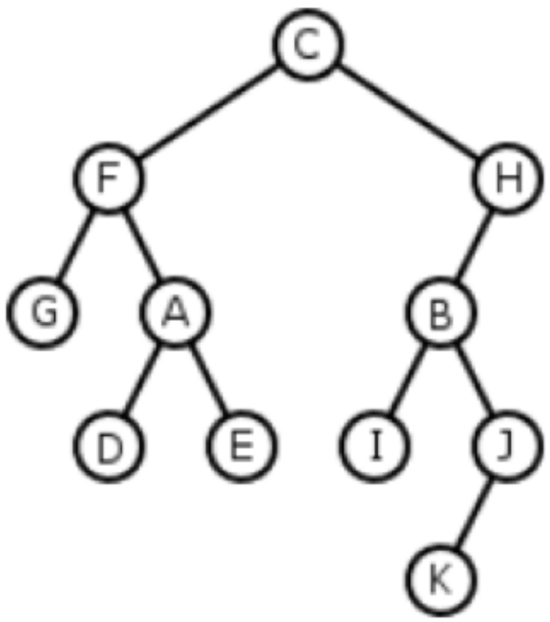
Output: G F D A E C I B K J H

Inorder Traversal

Inorder Left, Visit the node, Inorder Right

Tree Traversals

Visit = Print The Node Value



Output: ?

Postorder Traversal

Postorder Left, Postorder Right, Visit the node

Tree Traversals: Final Comments

- Also called Tree-Walks
- Take $O(n)$

Binary Search Trees

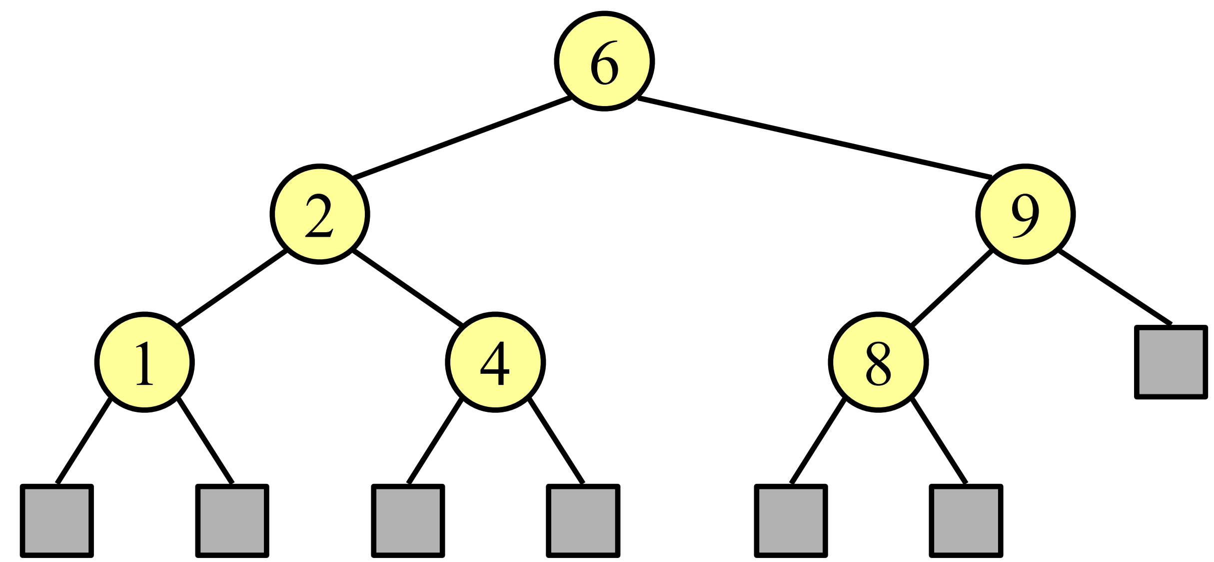
Searching in a BST

- Why is it called a binary search tree?

    - Data is stored in such a way, that it can be more efficiently found than in an ordinary binary tree

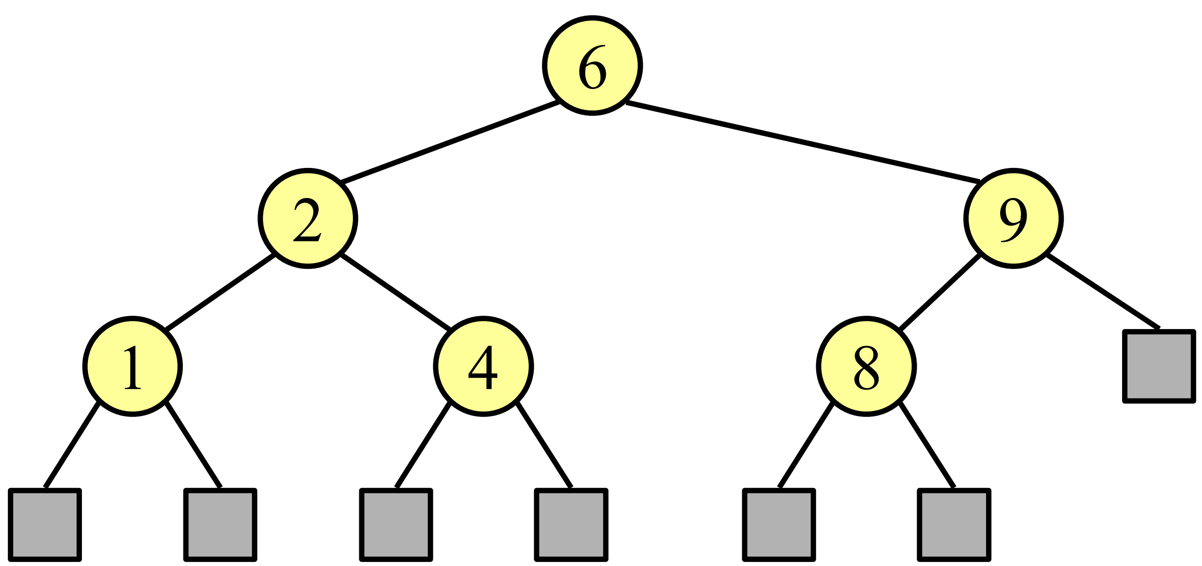
Binary Search Trees

- A special type of binary tree
  - it represents information in an ordered format
  - A binary search tree is a binary tree in which a node's value is
  - > every value in its left subtree, and
  - < every value in its right subtree

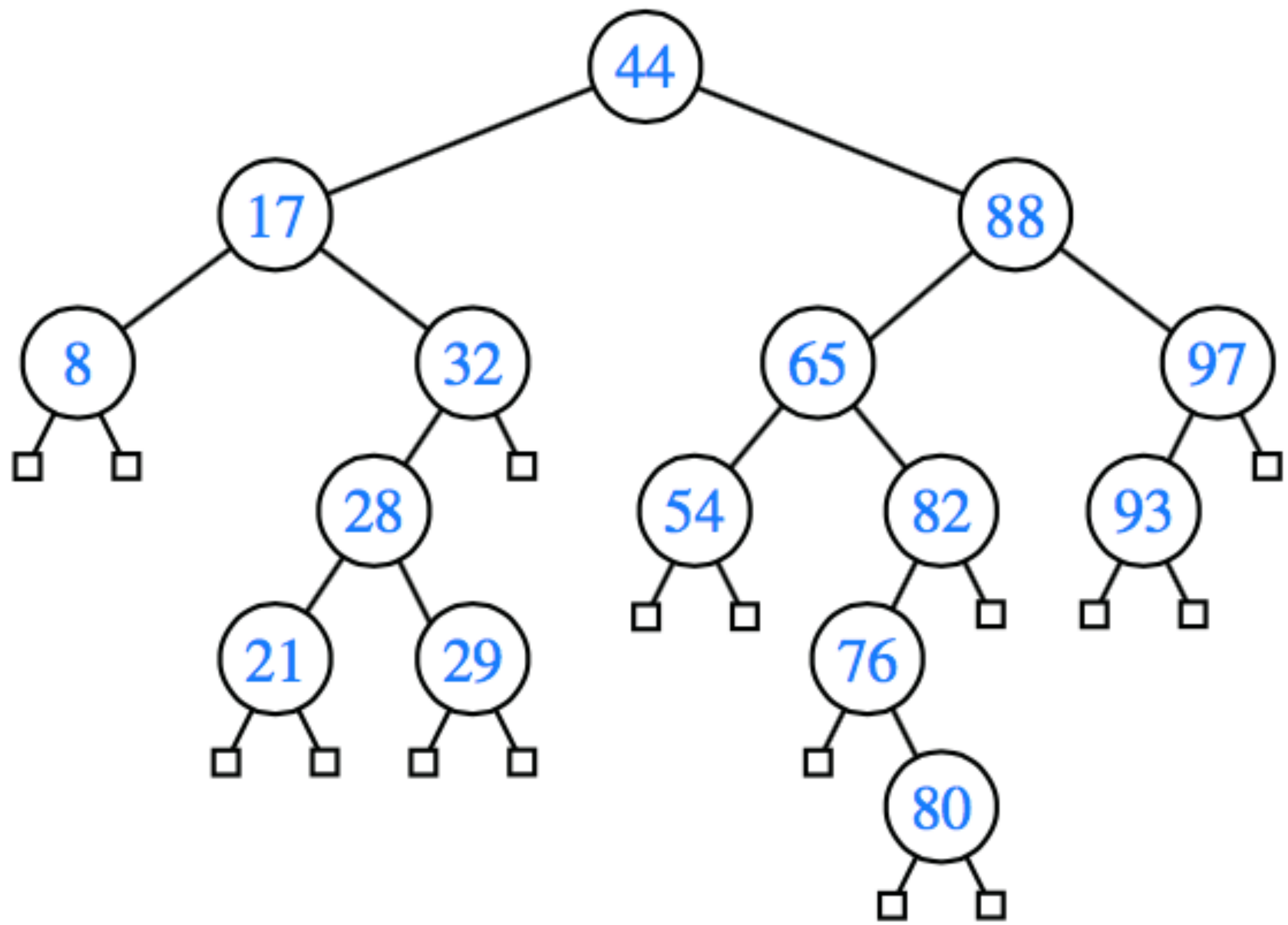


Binary Search Tree Property

Let x be a node in a binary search tree. If y is a node in the left subtree of x, then $y.key \leq x.key$. If y is a node in the right subtree of x, then $y.key \geq x.key$.



Example



- What is in the leftmost node?
- What is in the rightmost node?

BST Operations

- search
- add an element (requires that the BST property be maintained)
- remove an element (requires that the BST property be maintained)
- Remove/find the maximum element
- Remove/find the minimum element

Searching in a BST

- Recursive Algorithm to search for an item in a BST

```
TREE-SEARCH(x, k)
if x == NIL or k == x.key
    return x
if k < x.key
    return TREE-SEARCH(x.left, k)
else return TREE-SEARCH(x.right, k)
```

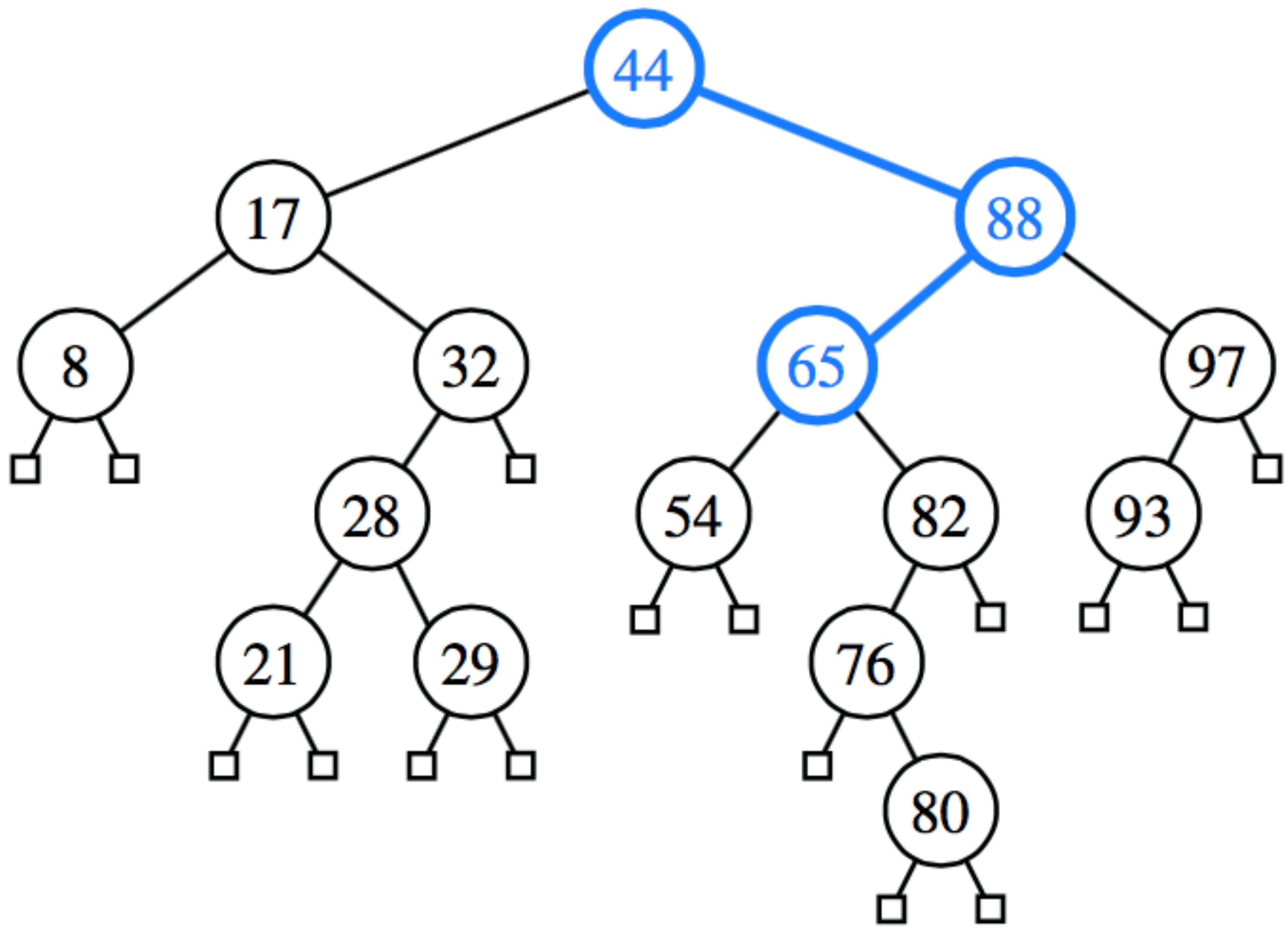
Searching in a BST

- Iterative Algorithm to search for an item in a BST

ITERATIVE-TREE-SEARCH(x, k)

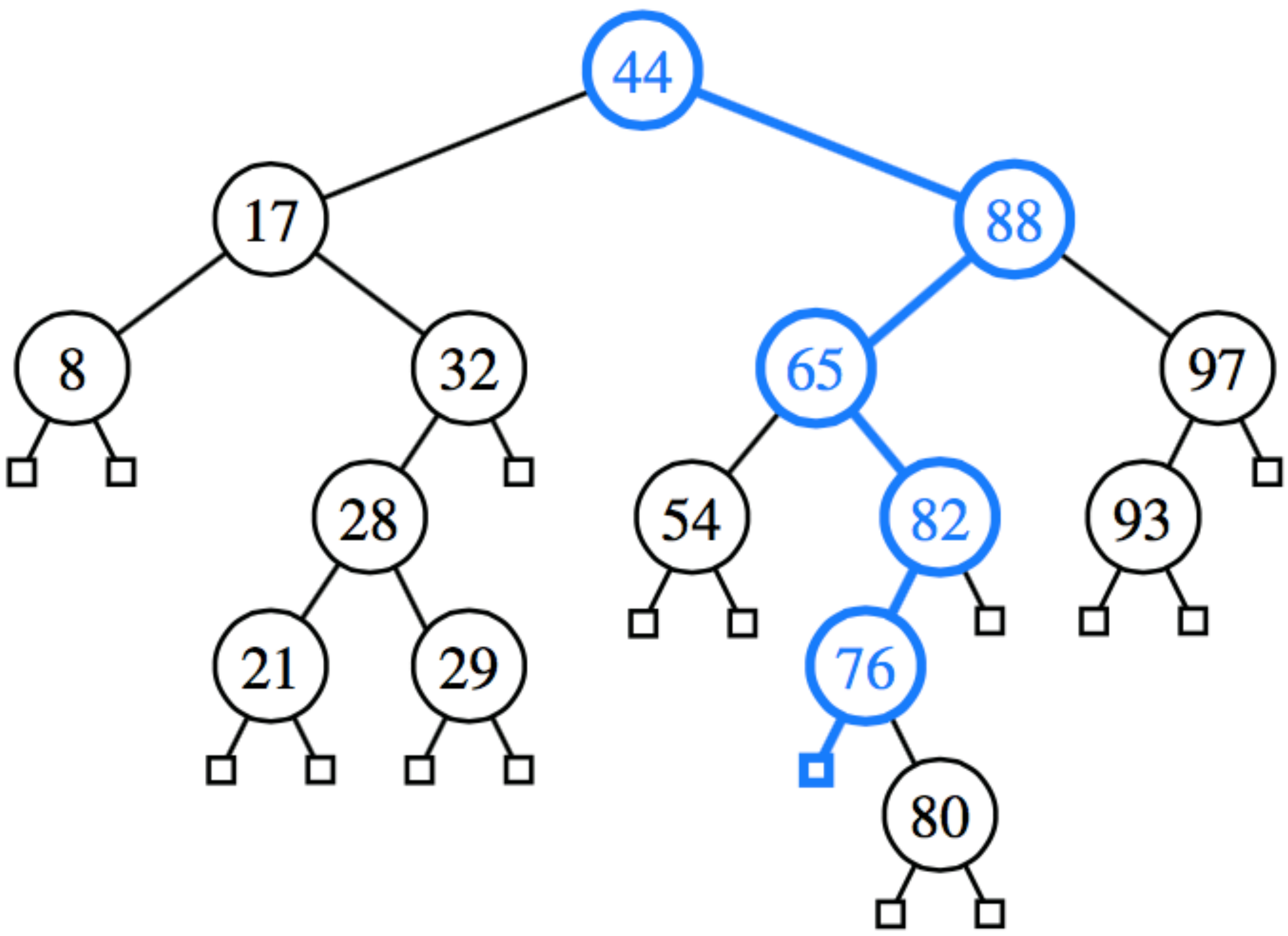
```
while x ≠ NIL and k ≠ x.key
    if k < x.key
        x = x.left
    else x = x.right
return x
```

Searching in a BST



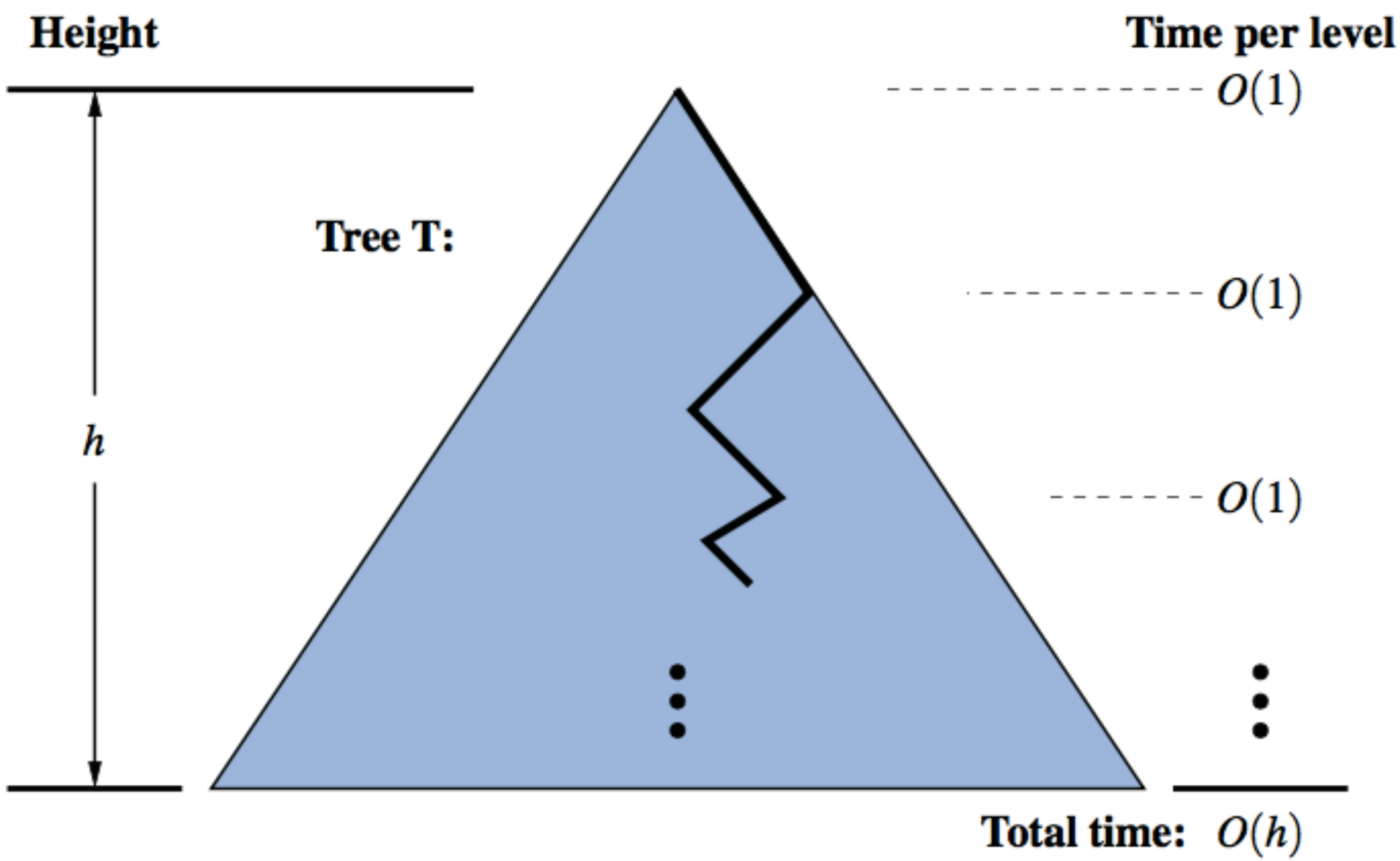
A successful search for key 65 in a binary search tree

Searching in a BST



A unsuccessful search for key 68 that terminates at the leaf to the left of key 76

Analysis of BST Searching



Searching in BST

- Searching in a BST having n of height h takes $O(h)$ time

Searching for Min in a BST

TREE-MINIMUM(x)

```
while x.left ≠ NIL
    x = x.left
return x
```

Time Complexity: $O(h)$

Searching for Max in a BST

TREE-MAXIMUM(x)

while $x.right \neq$ NIL
    $x = x.right$
return x

Time Complexity: $O(h)$

Insertions in a BST

- To ***add*** an item to a BST:

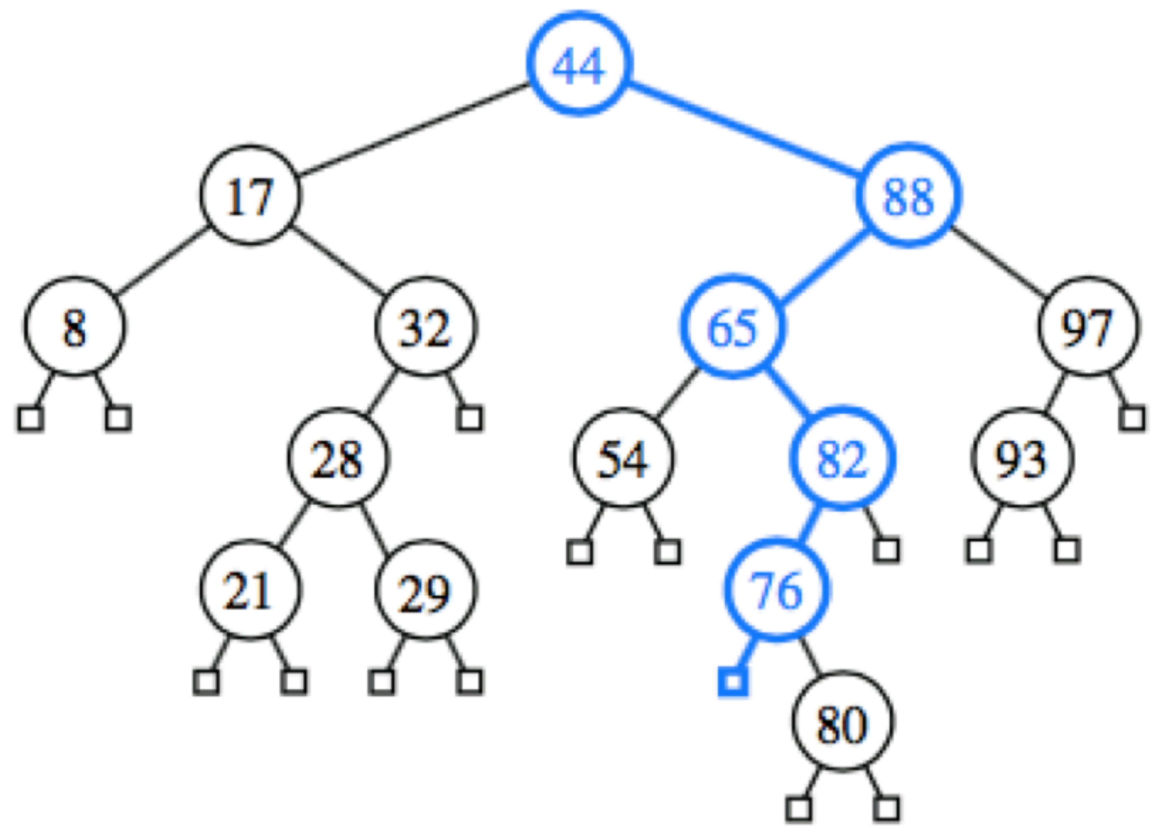
    1. Follow the algorithm for searching, until there is no child
    2. Insert at that point

- So, new node will be added as a leaf

- (We are assuming no duplicates allowed)

Insertions in a BST

- Inserting 68 in the following tree



(a)

Insertions in a BST

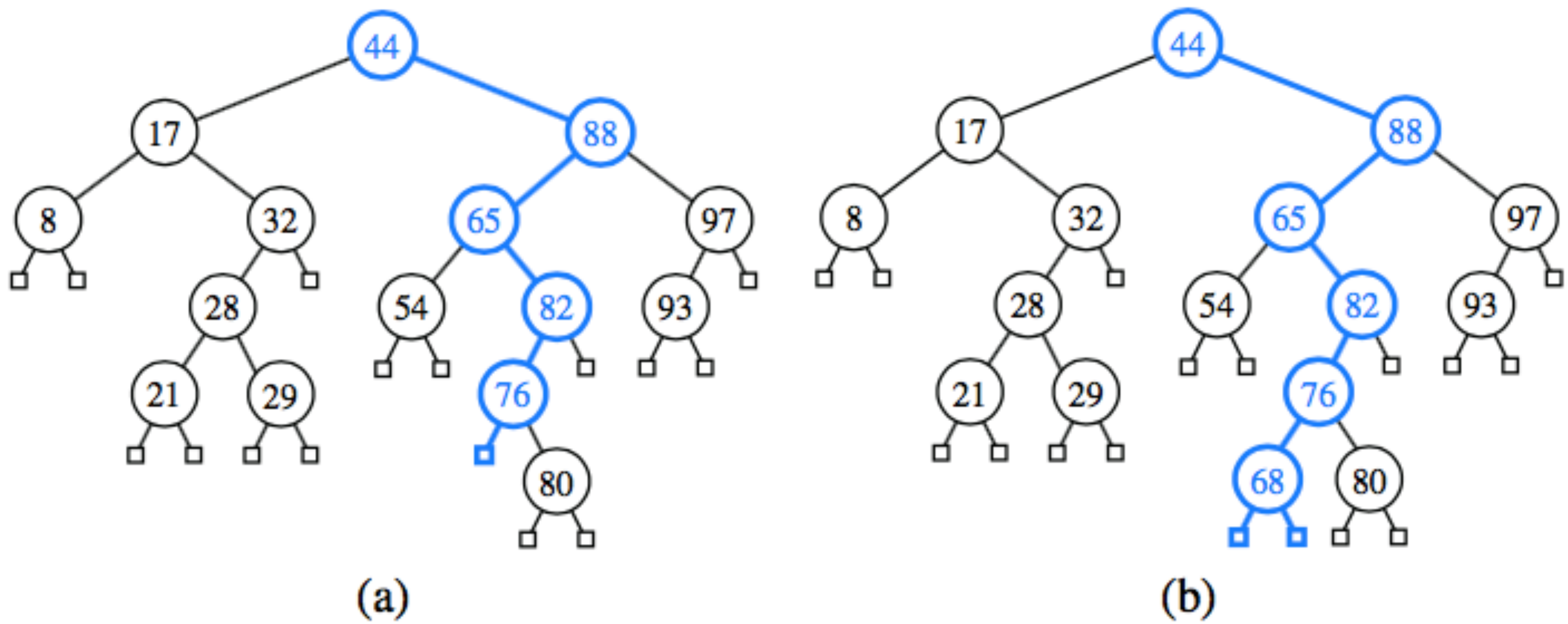


Figure 11.4: Insertion of an entry with key 68 into the search tree of Figure 11.2. Finding the position to insert is shown in (a), and the resulting tree is shown in (b).

Insertion in BST

- First Search in $O(h)$ → Then Insert in $O(1)$

- Thus time complexity is $O(h)$

Deletions in a BST

method remove (key)

1. if the tree is empty return false

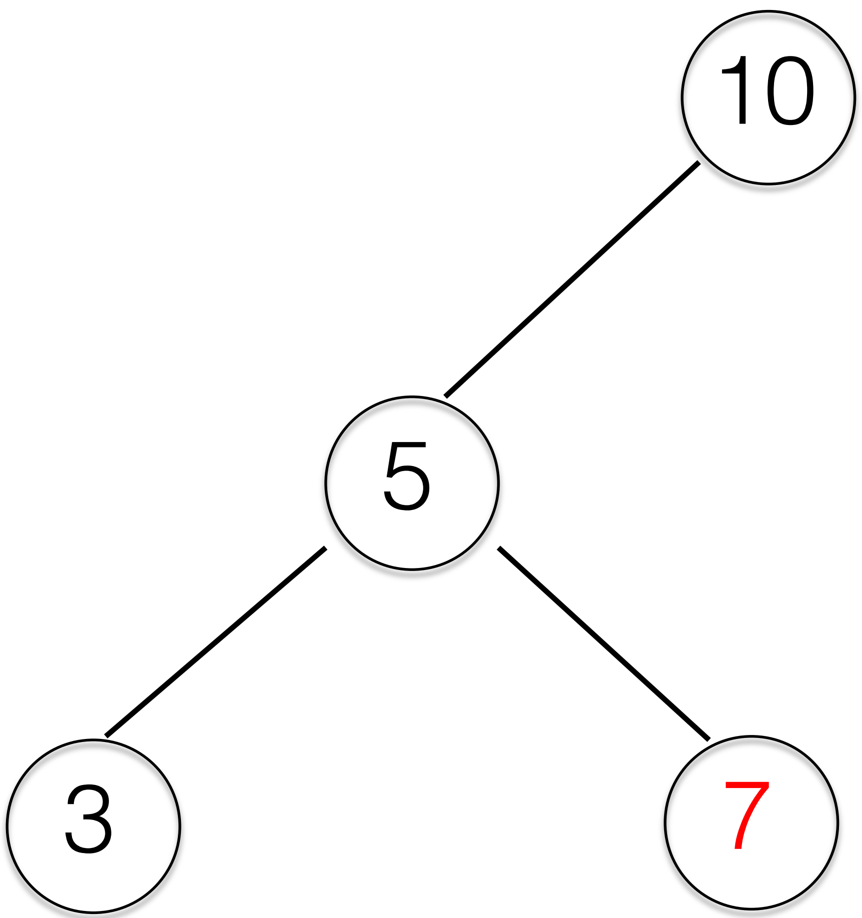
2. Attempt to locate the node containing the target using the binary search algorithm
   if the target is not found return false
   else the target is found, so remove its node:

// Now there can be 3 cases

Deletions in a BST

// The easiest case, the node has no children – is a leaf

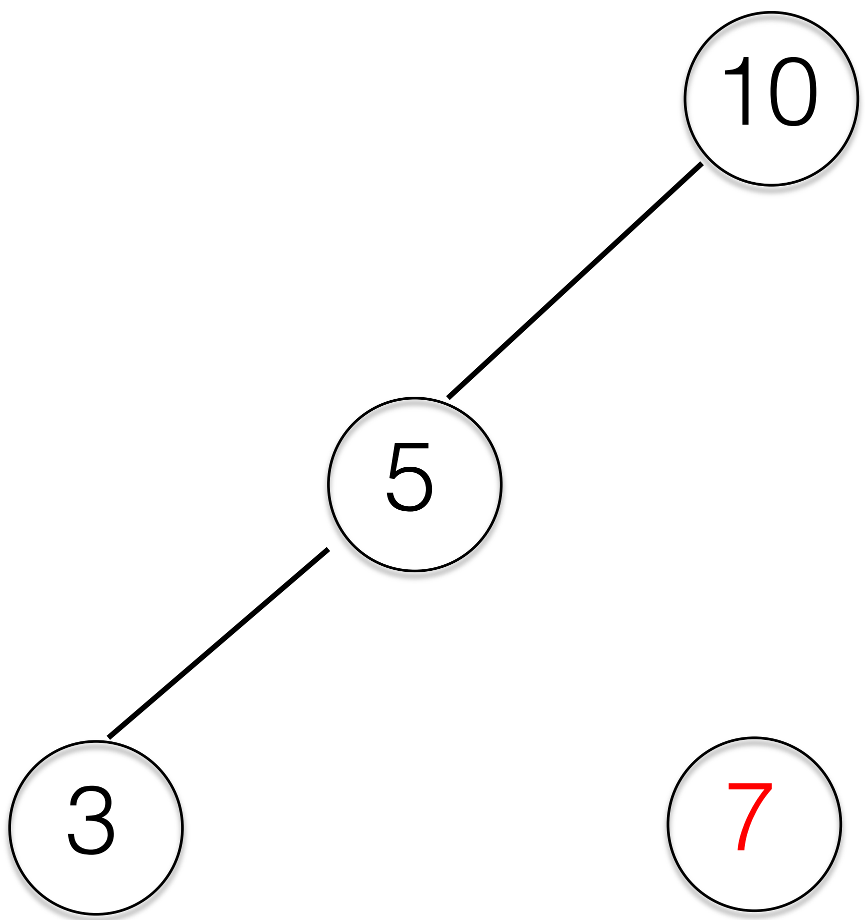
Case 1: the node has no children



Let's delete the node with key 7

Deletions in a BST

// **Case 1:** **the node has no children**

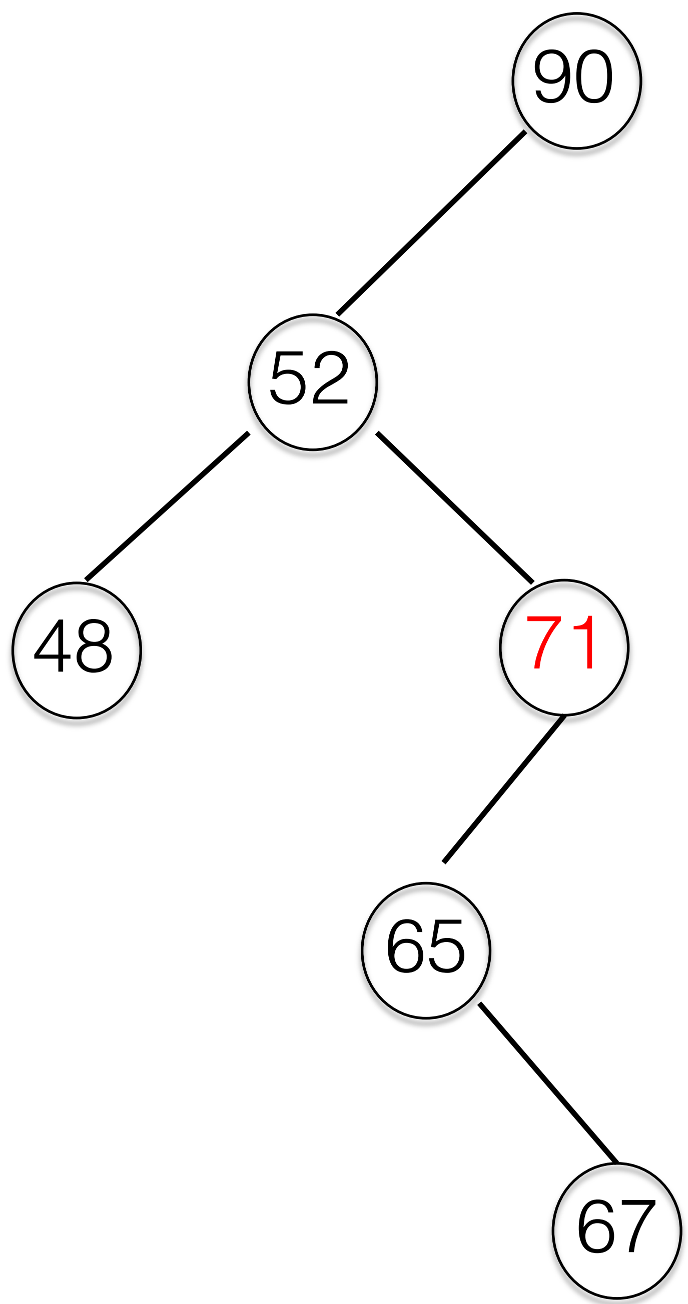


Replace the link in the parent with null!

Awaiting garbage
collection

Deletions in a BST

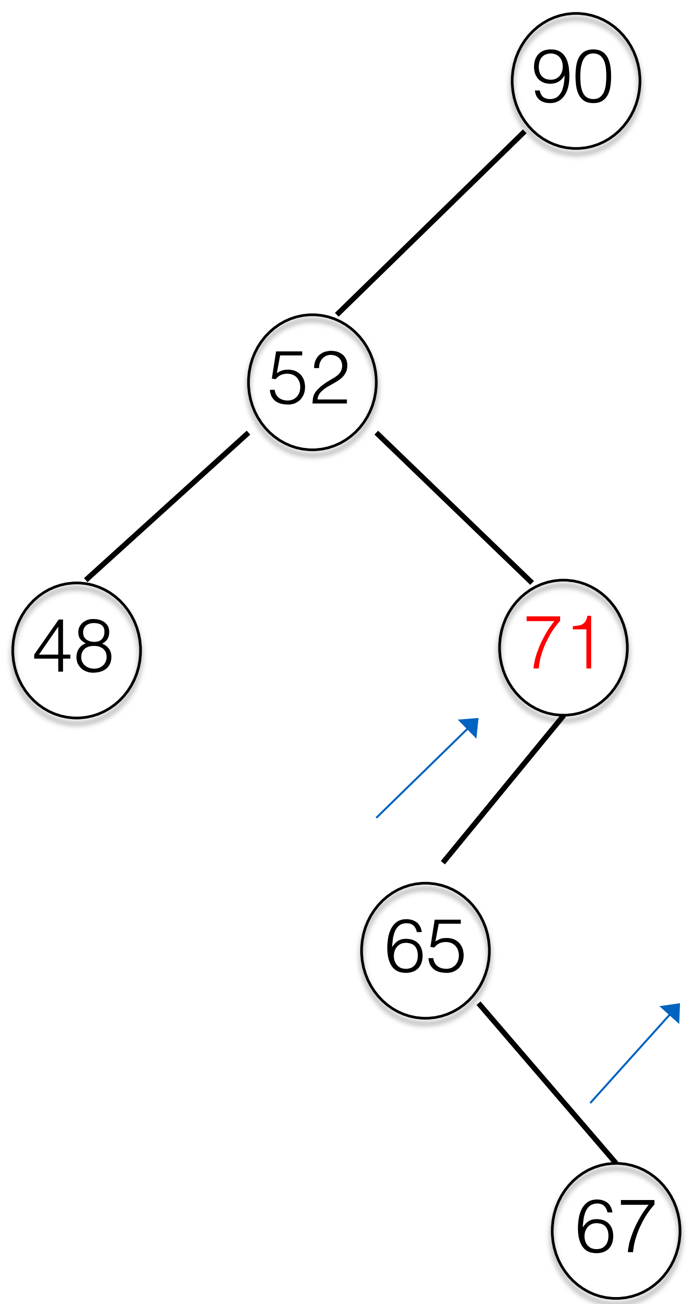
Case 2: **the node has only one child**



Let's delete the node with key 71

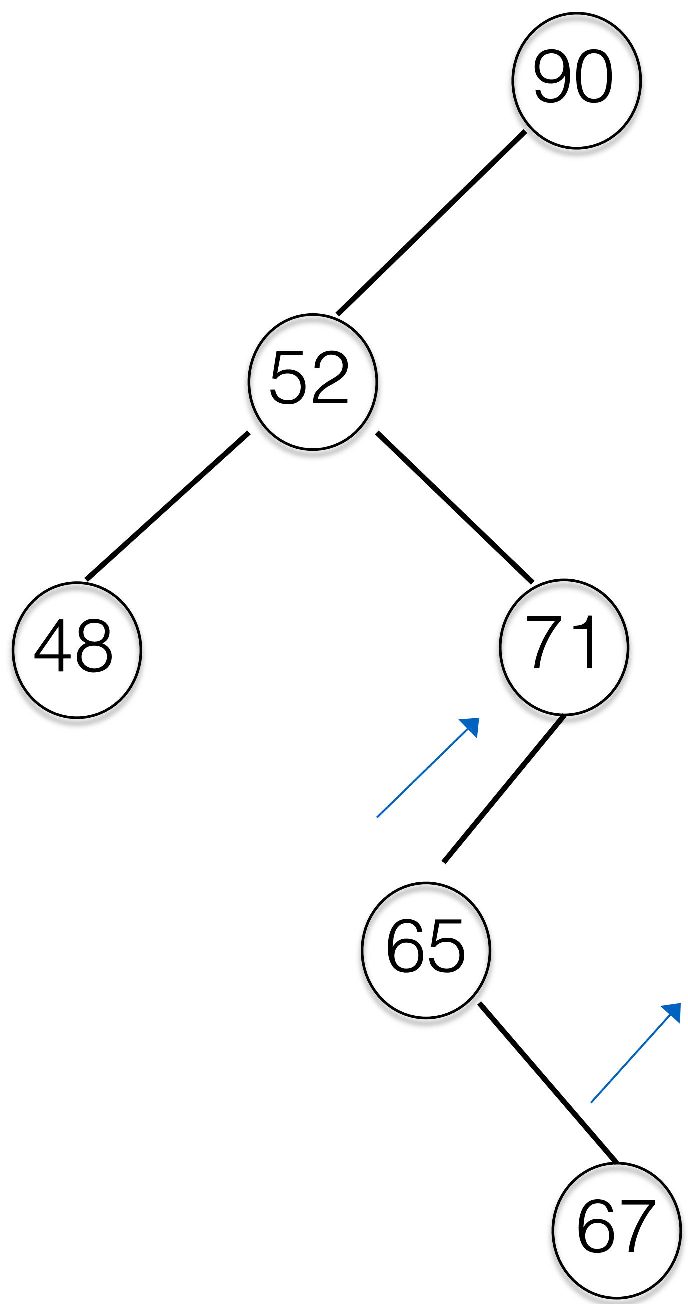
Deletions in a BST

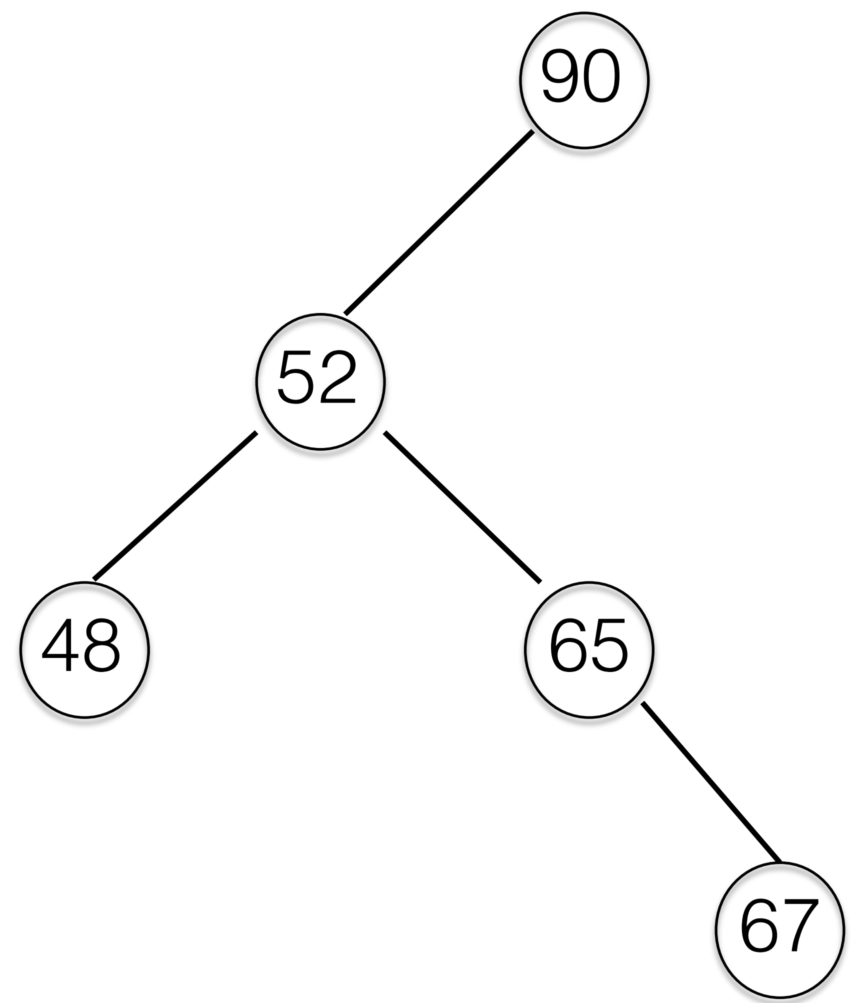
Case 2: the node has only one child



Deletions in a BST

Case 2: **the node has only one child**



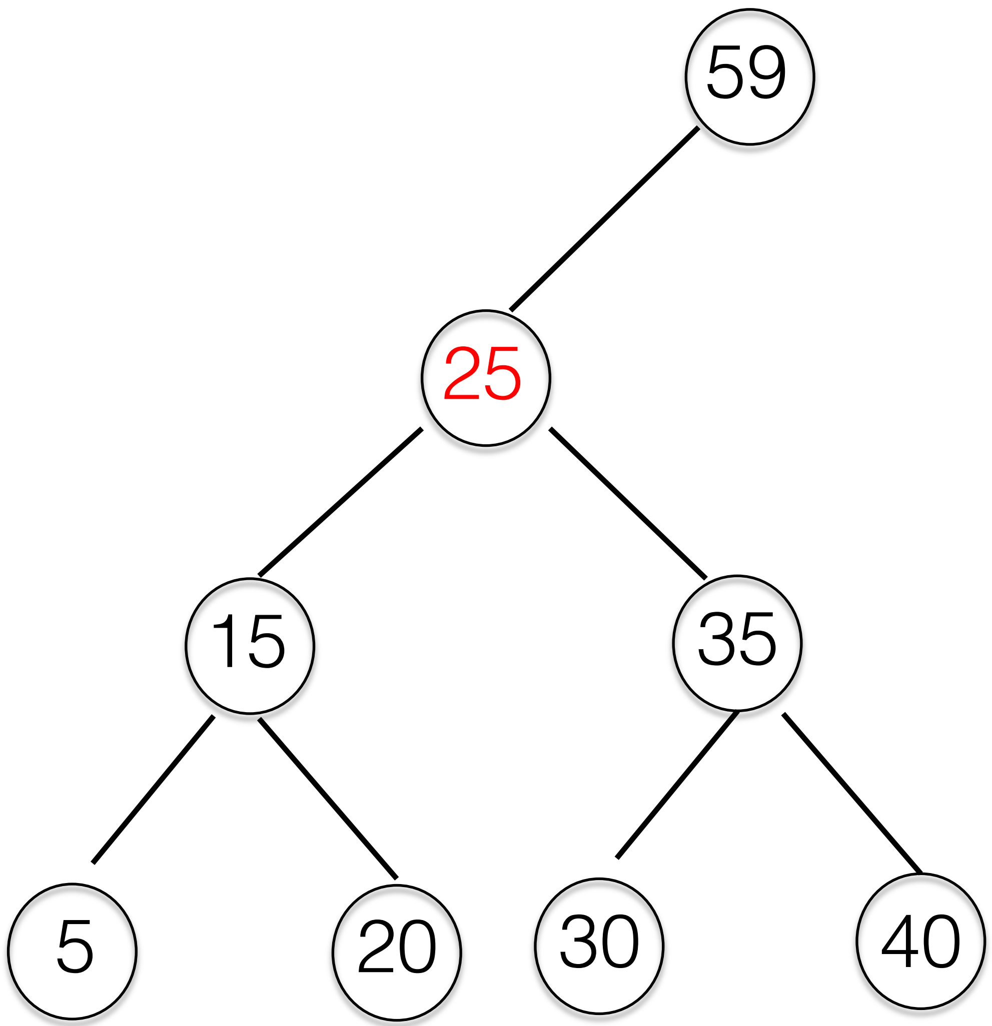


Deletions in a BST

// The tricky case is when the node has two children
// deleting this node will leave two children in trouble

Case 3: if the node has a two children

Deletions in a BST

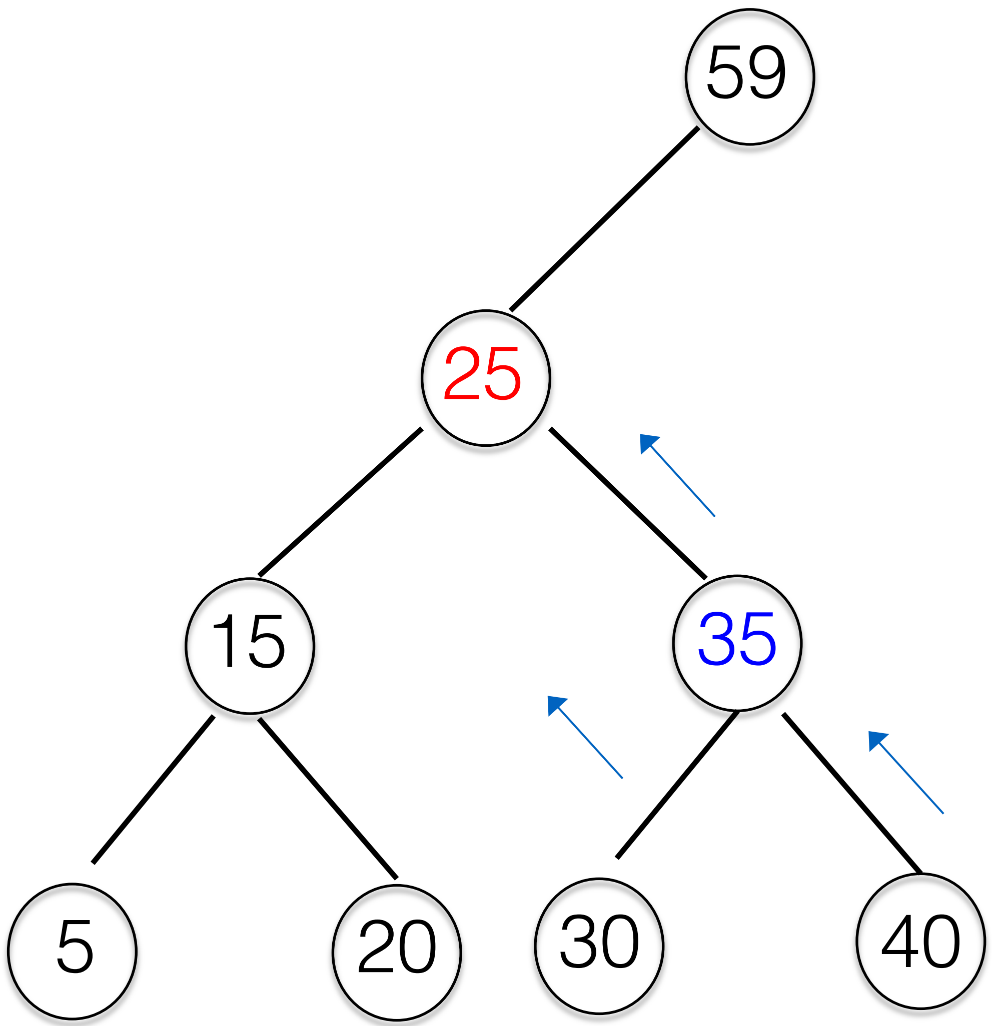


Case 3: if the node has a two children

Let's delete 25 – Two choices

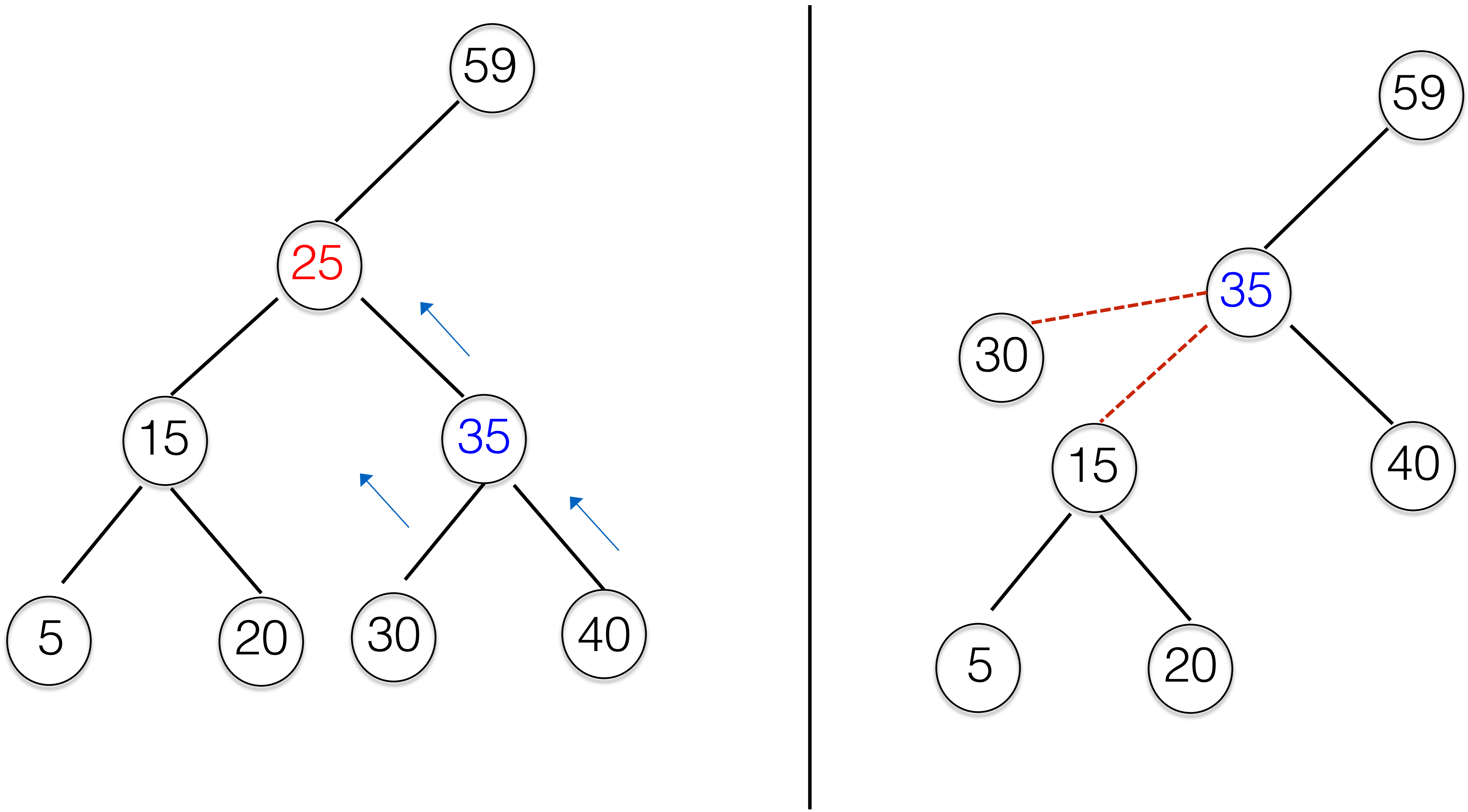
- Replace with root of left sub-tree, OR
- Replace with the root of right sub-tree

Deletions in a BST



- Replace with the root of right sub-tree

Deletions in a BST



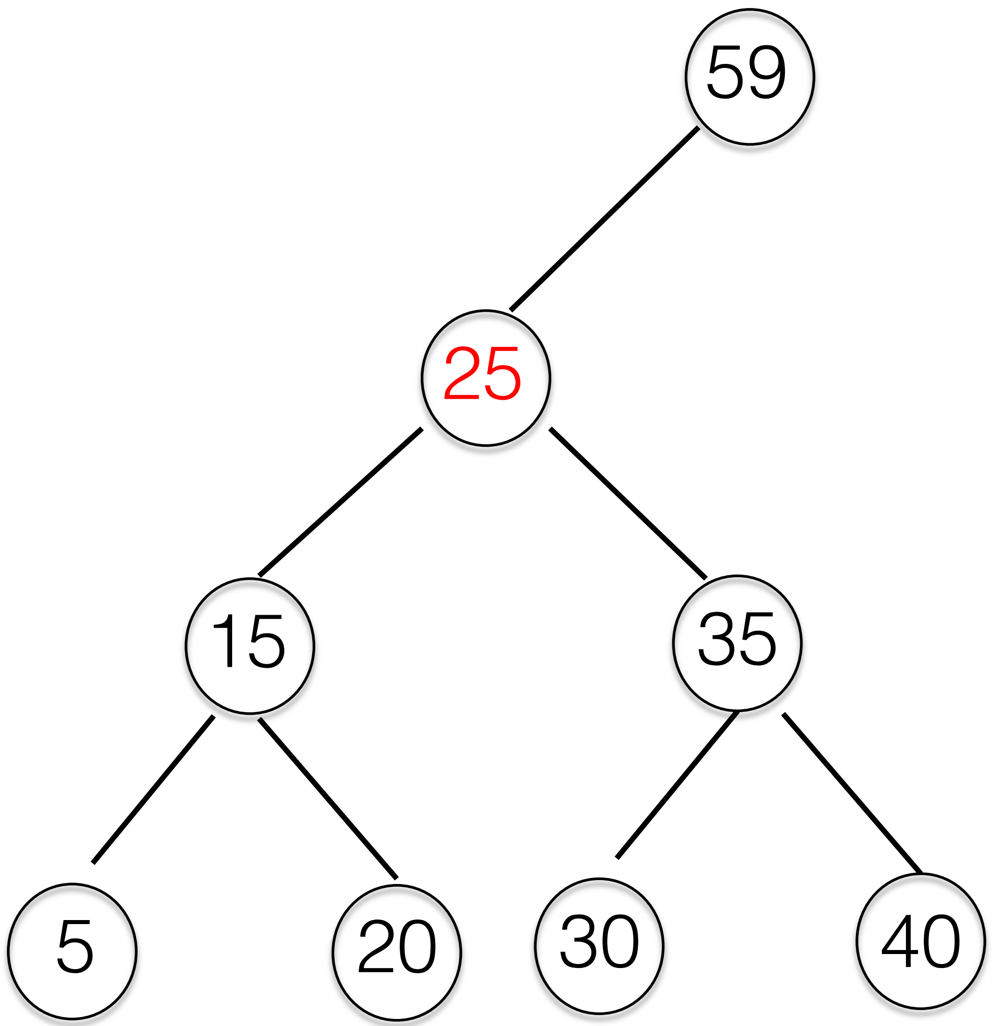
Deletions in a BST

Case 3: if the node has a two children

// Thus a trick is needed

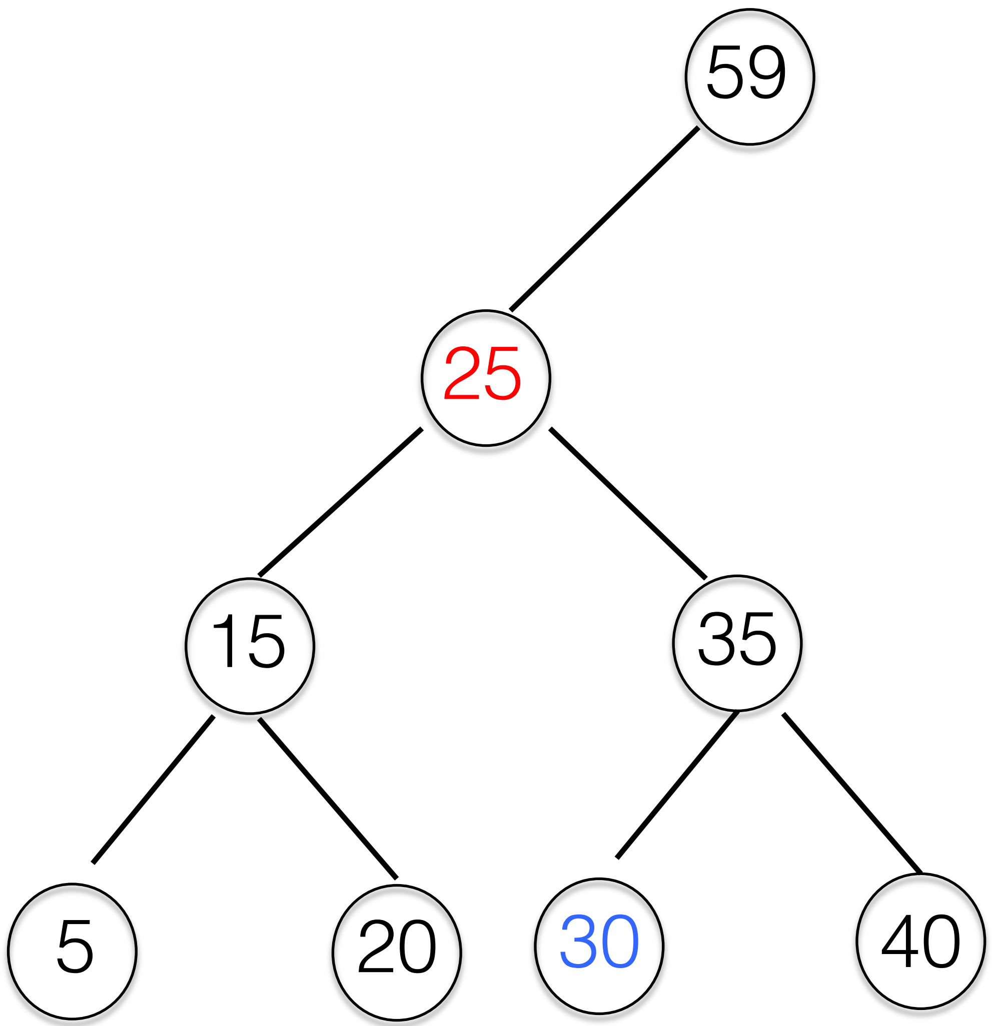
Replace the node with its **predecessor or successor from the inorder traversal** of the tree, and delete that node instead.

Inorder Successor



What is the in-order successor of node 25?

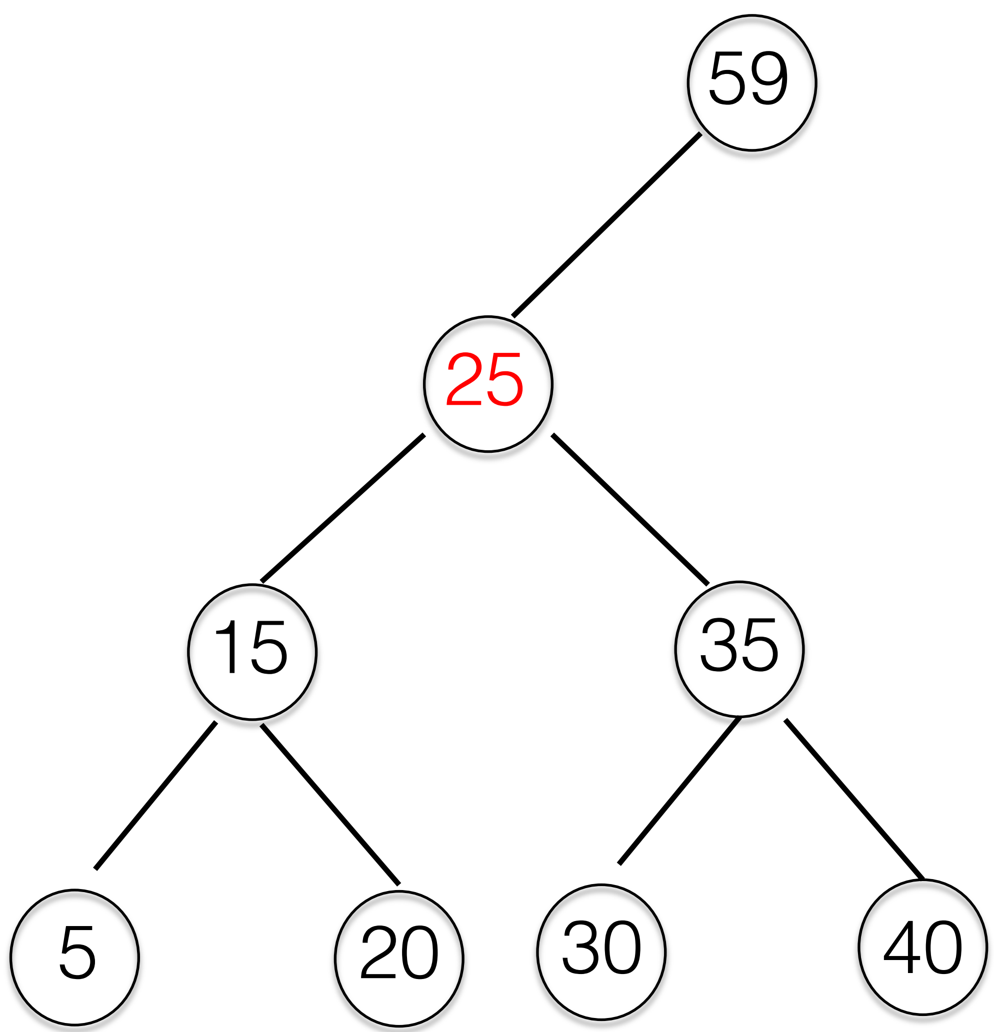
Inorder Successor



What is the in-order successor of node 25?

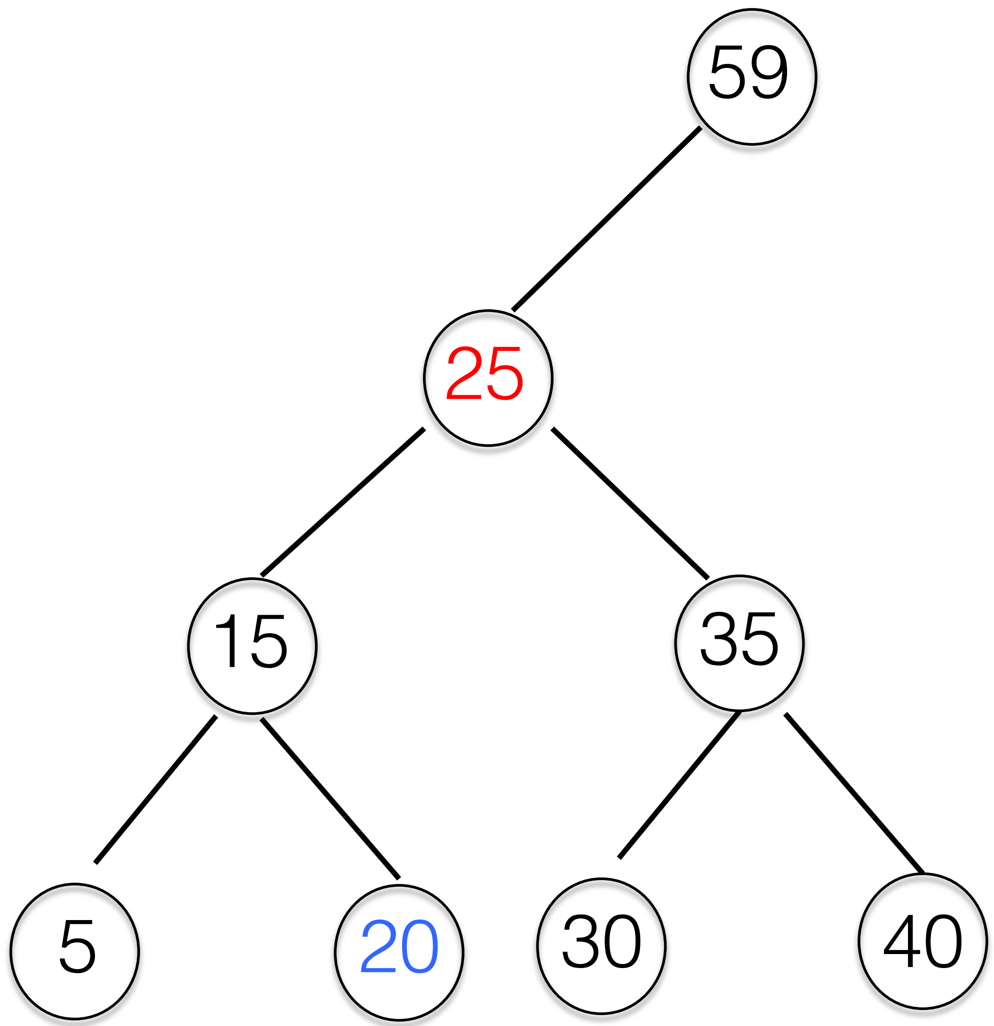
30

Inorder Predecessor



What is the in-order predecessor of node 25?

Inorder Predecessor



What is the in-order predecessor of node 25?

20

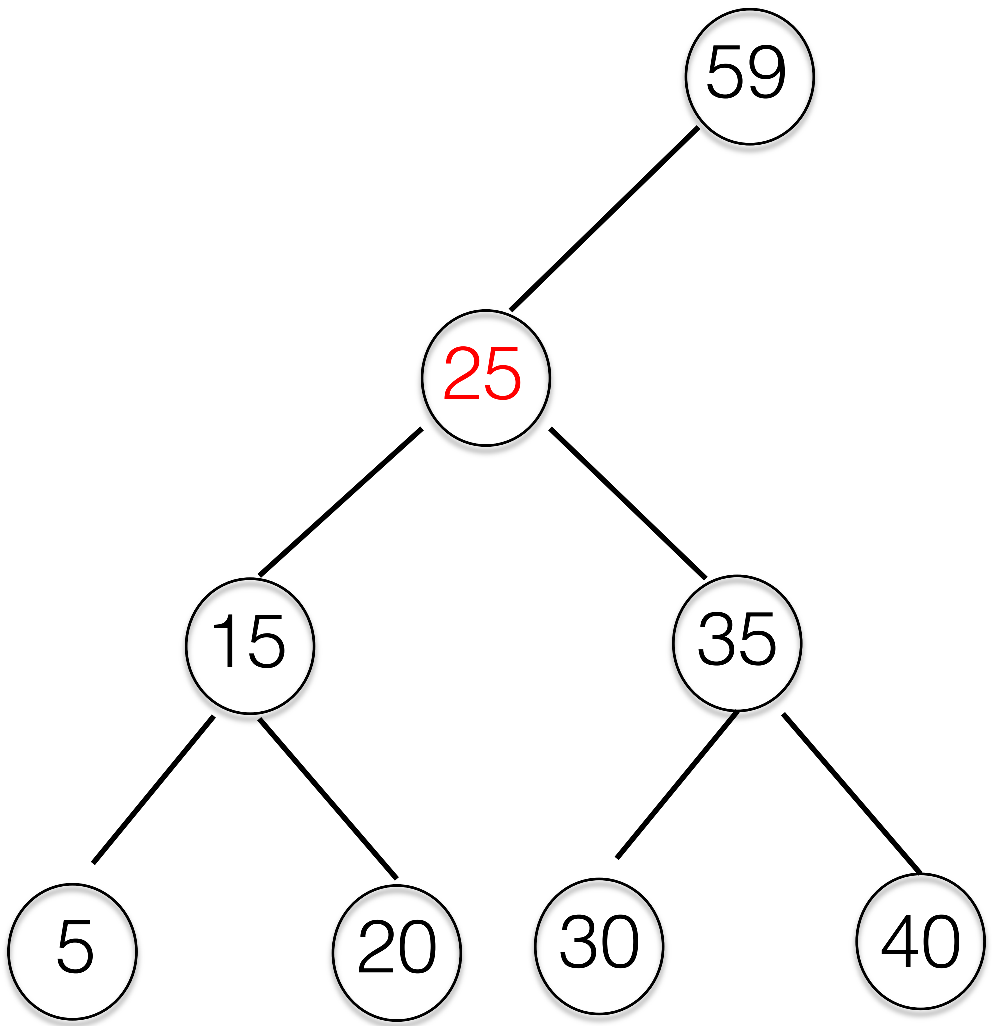
Deletions in a BST

// Thus a trick is needed

Case 3: if the node has a two children

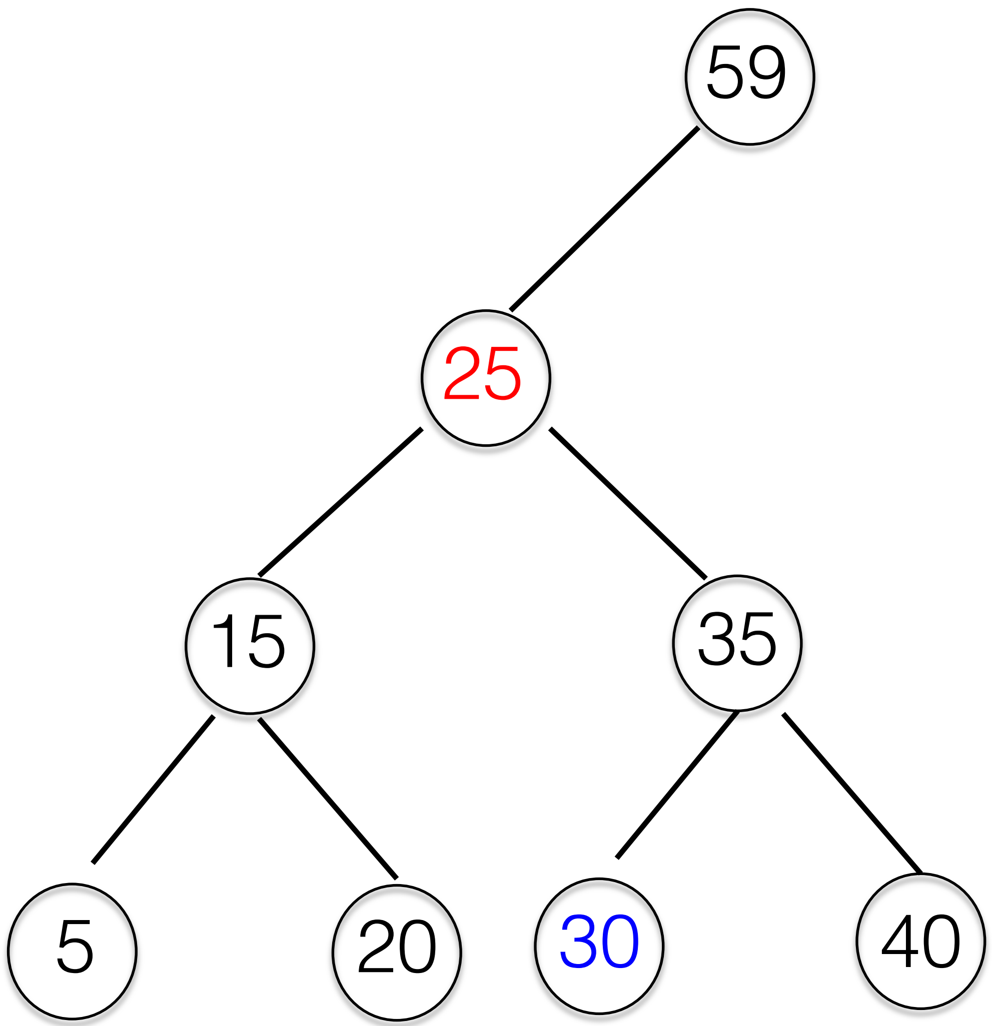
Replace the node with its **predecessor or successor from the inorder traversal** of the tree. and delete that node instead.

Deletions in a BST



Let's delete 25

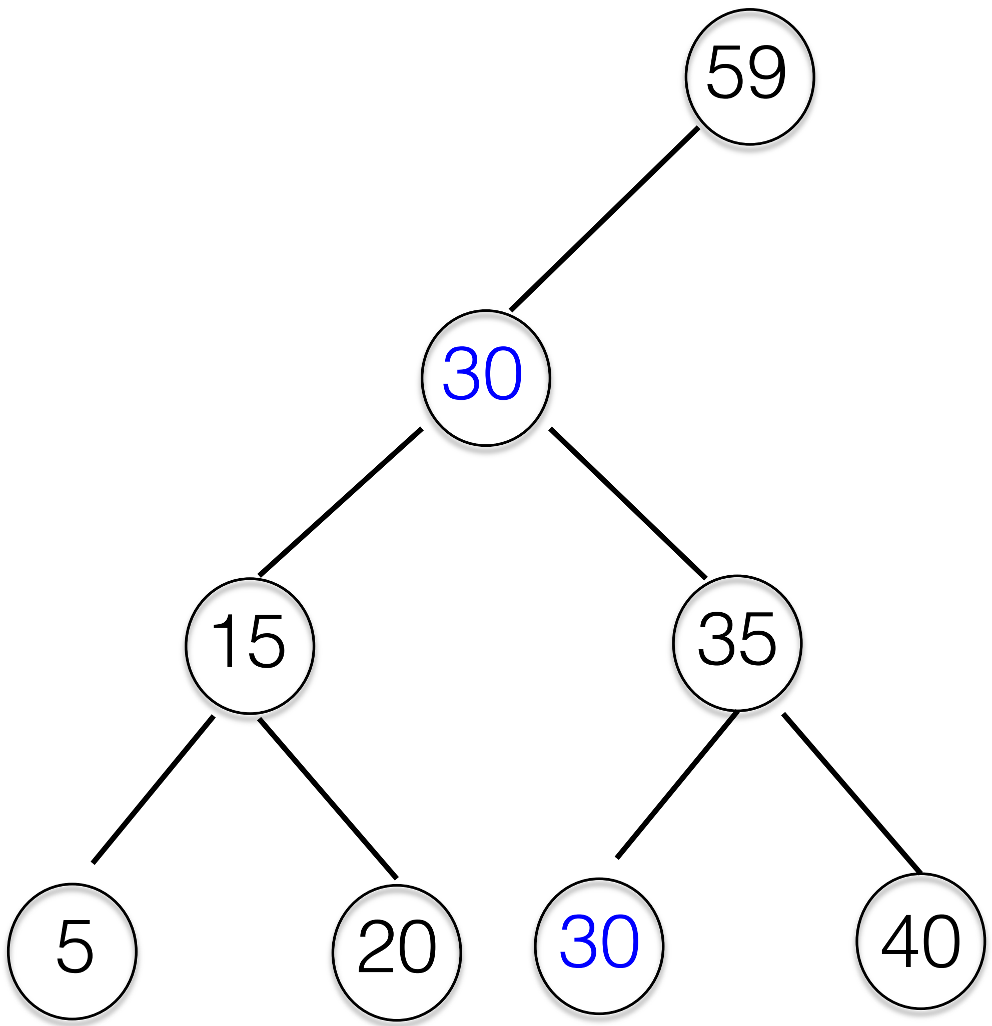
Deletions in a BST



Let's delete 25

- Identify its inorder successor

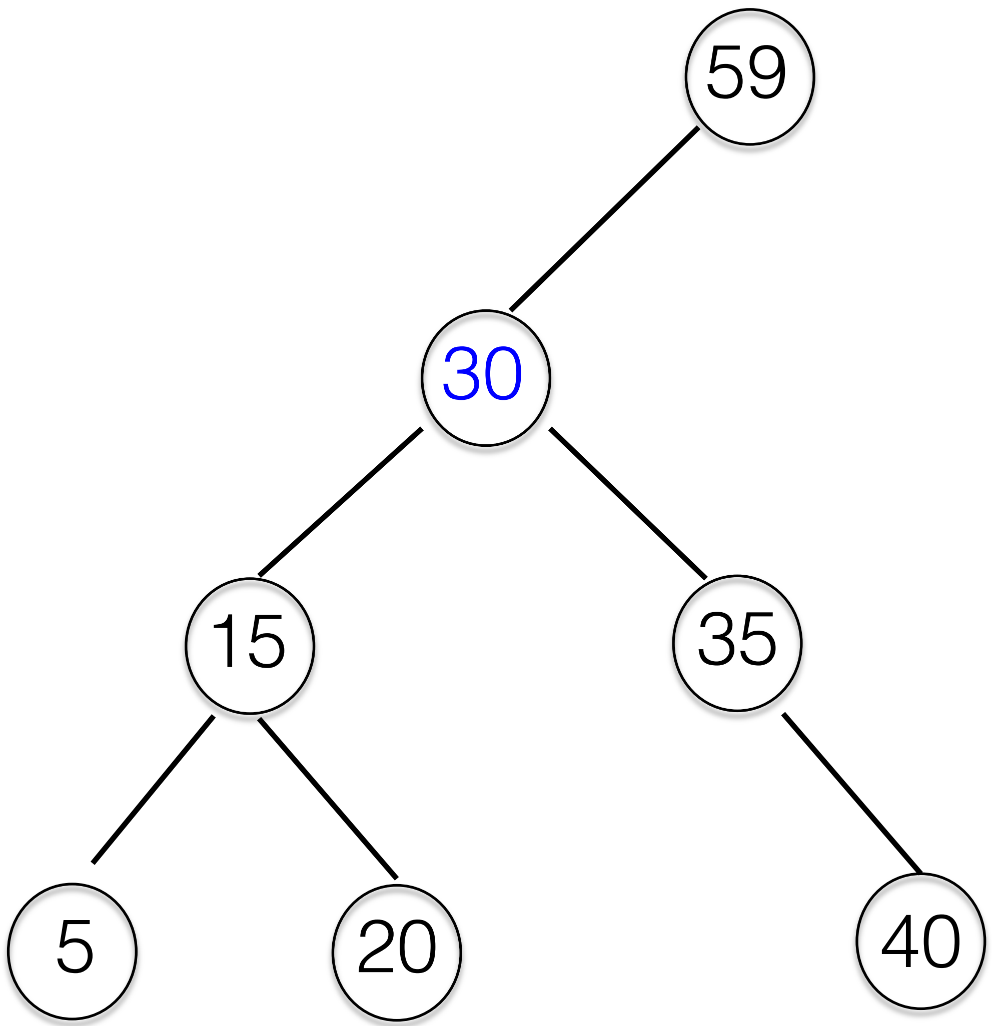
Deletions in a BST



Let's delete 25

- Replace it with its successor

Deletions in a BST



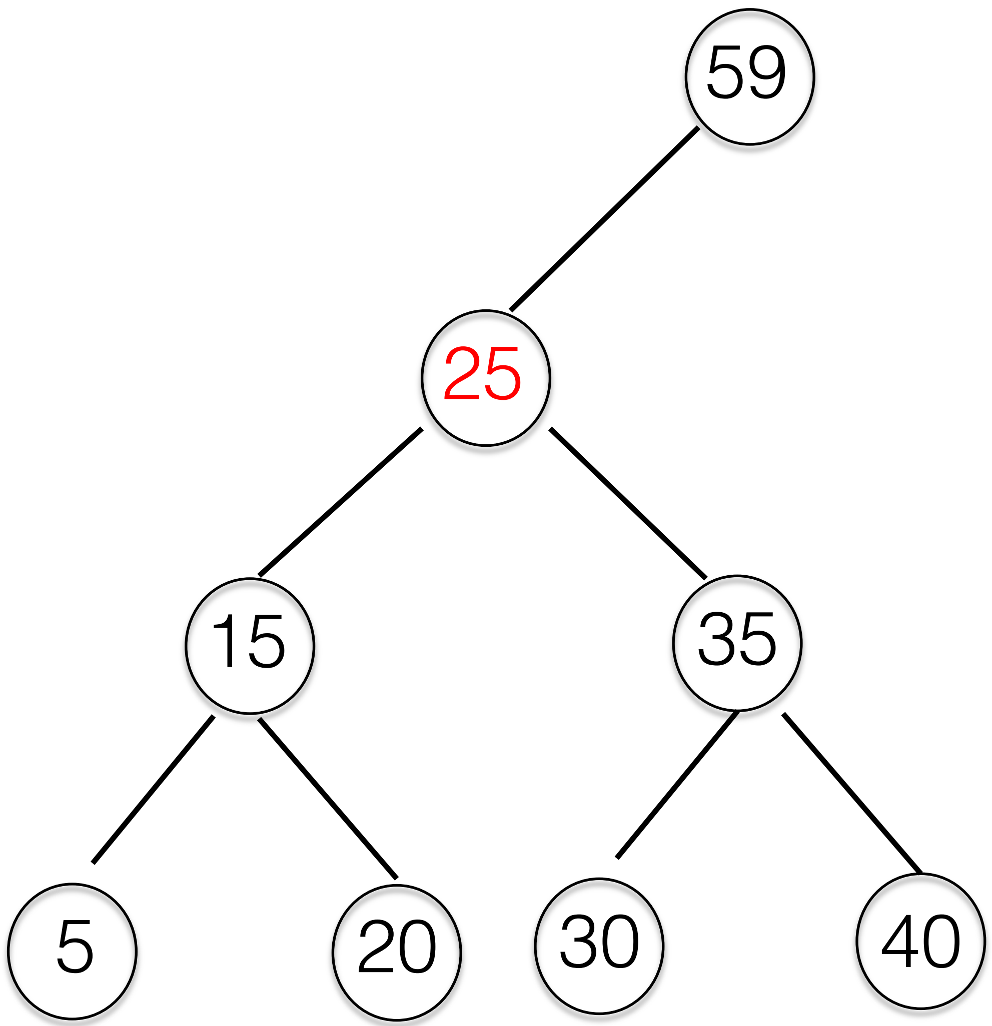
Let's delete 25

- Delete the successor

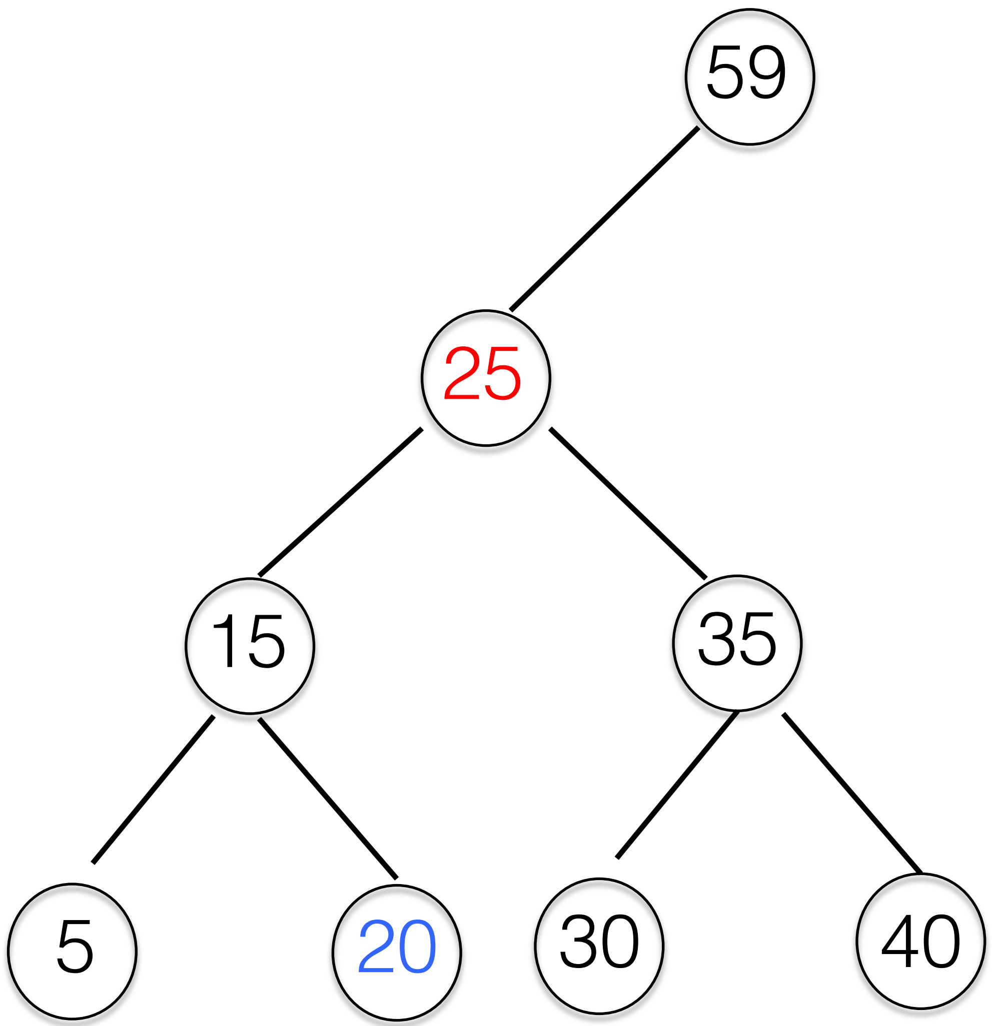
OR, use the Predecessor

Deletions in a BST

Let's delete 25



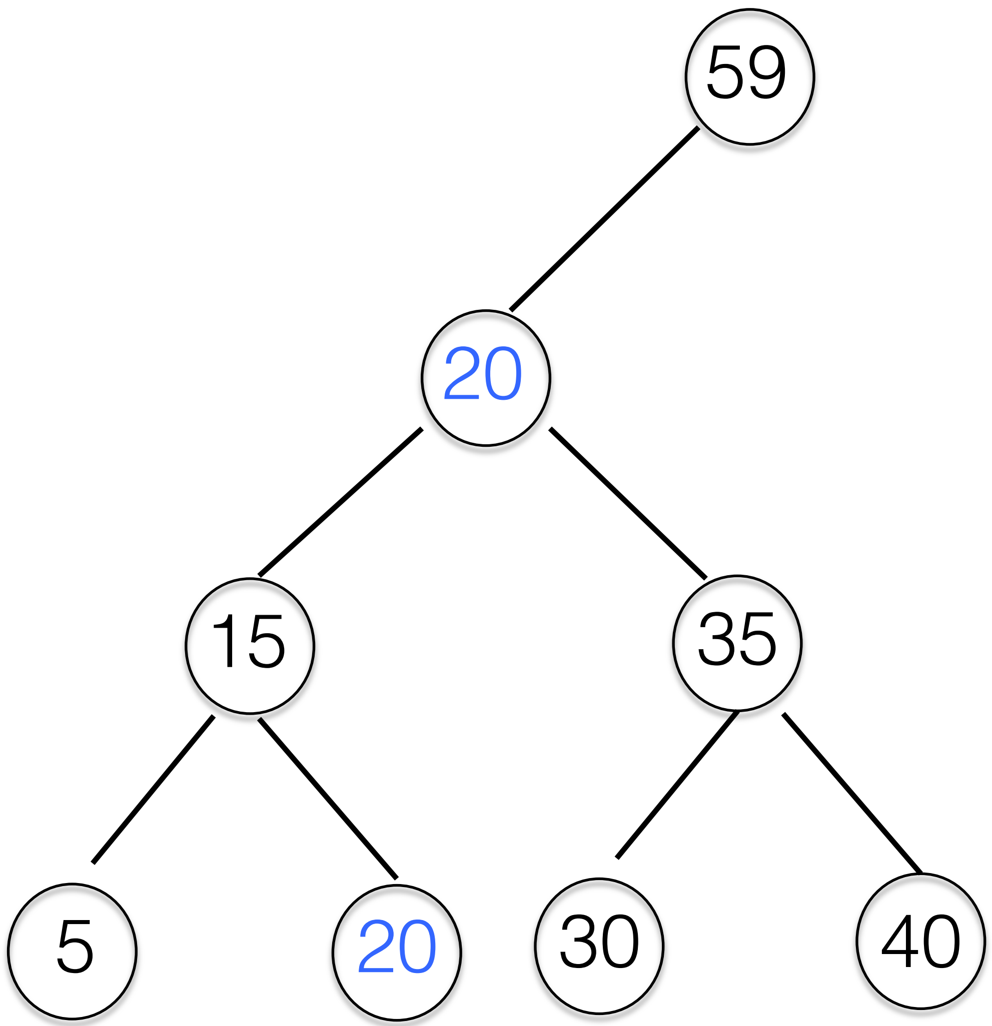
Deletions in a BST



Let’s delete 25

- Identify its predecessor

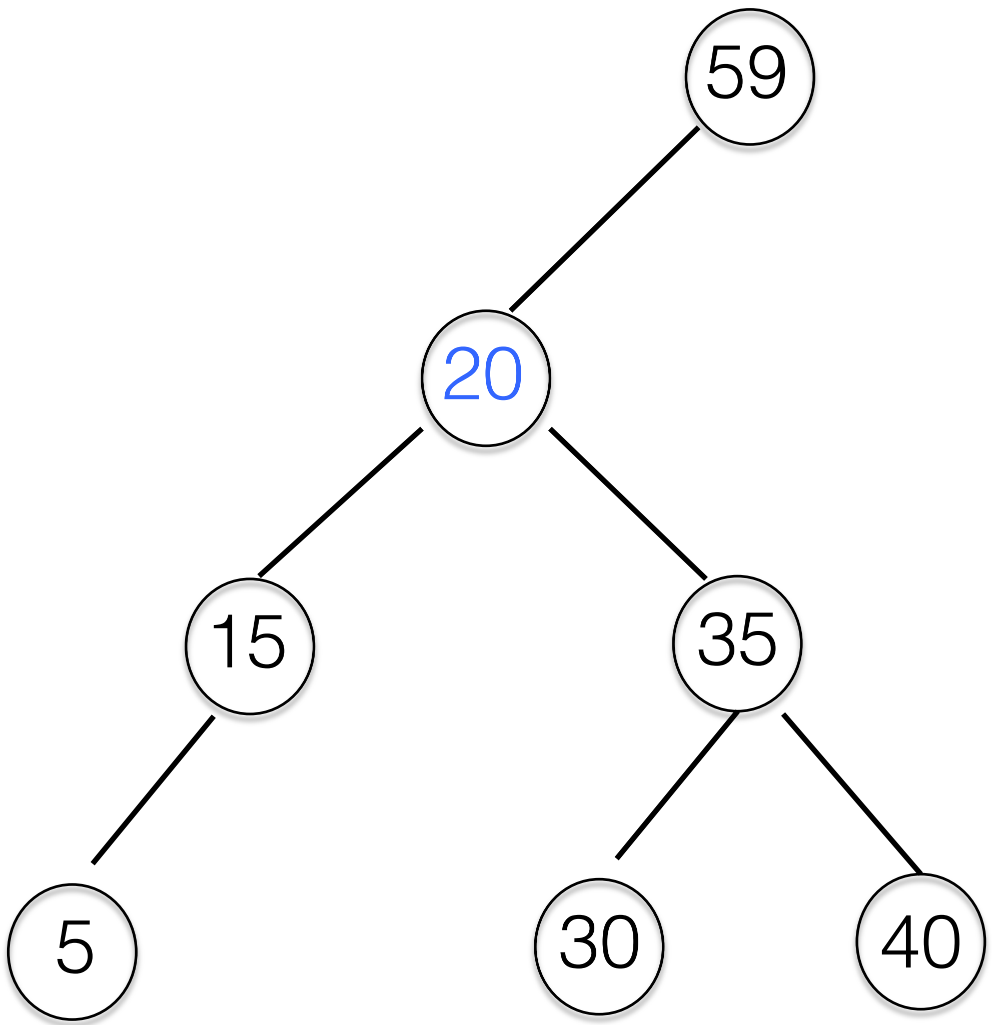
Deletions in a BST



Let's delete 25

- Replace the node with predecessor

Deletions in a BST



Let's delete 25

- Delete the predecessor

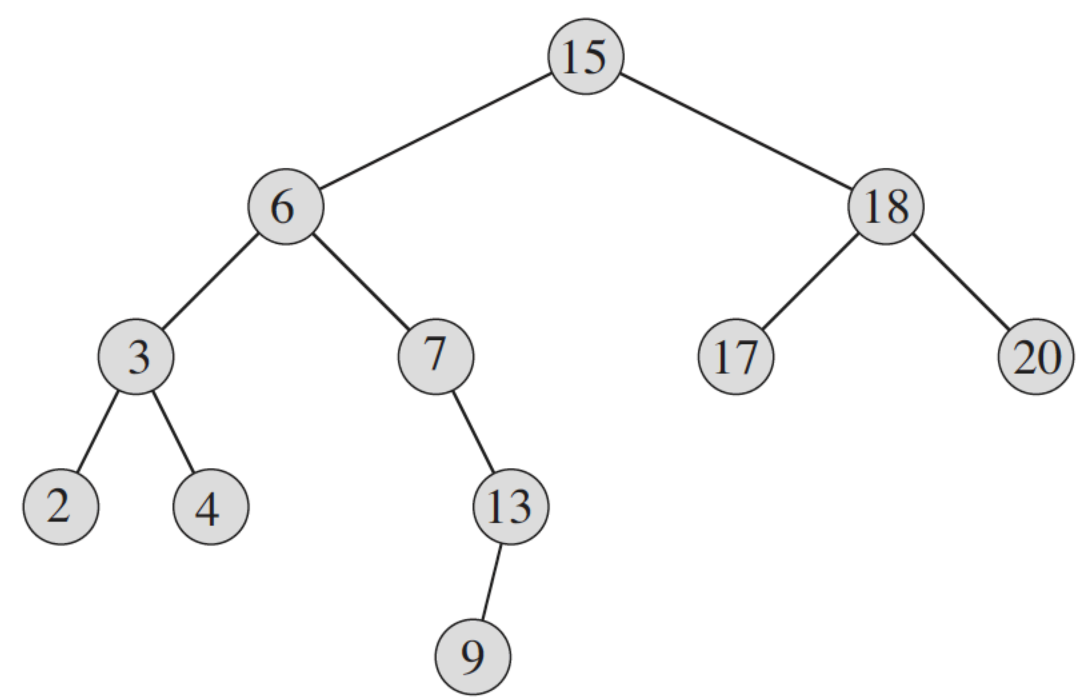
Deletion in BST

- First Search in $O(h)$ → Then Delete
- But delete requires In Order Traversal $O(n)$
- So can we find a way to find successor or predecessor without performing the actual traversal?

Finding Successor

TREE-SUCCESSOR(x)

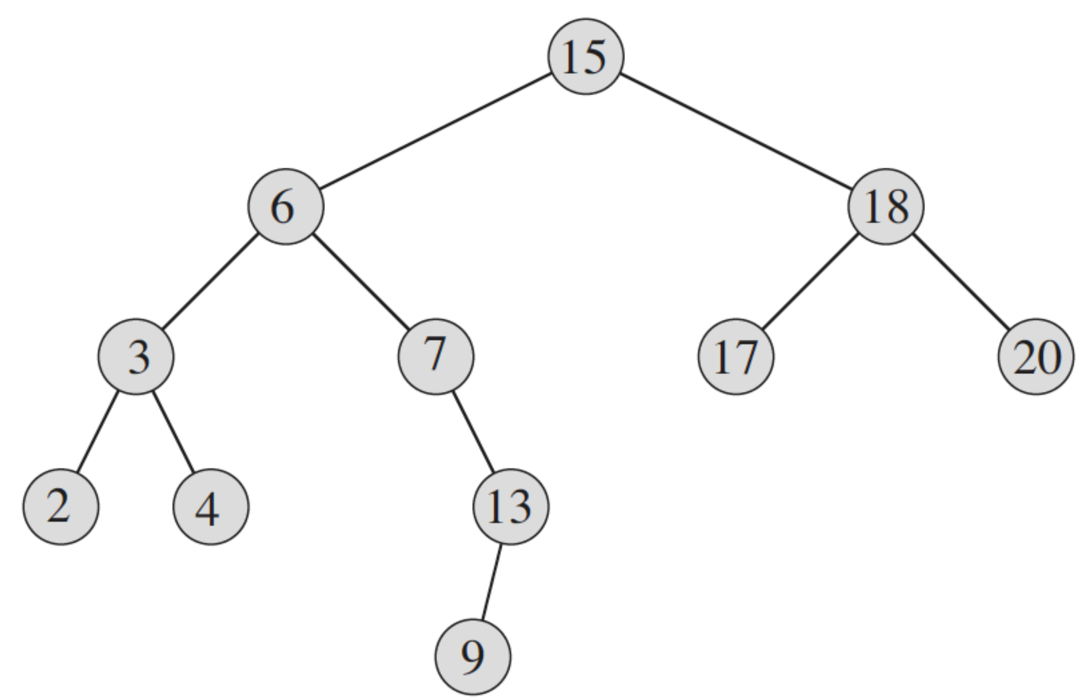
```
if x.right ≠ NIL
    return TREE-MINIMUM(x.right)
```



Successor of 15 is 17

Finding Successor

```
TREE-SUCCESSOR(x)
if x.right ≠ NIL
    return TREE-MINIMUM(x.right)
y = x.p
while y ≠ NIL and x == y.right
    x = y
    y = y.p
return y
```



Successor of 13 is 15

Finding Successor

```
TREE-SUCCESSOR(x)
if x.right ≠ NIL
    return TREE-MINIMUM(x.right)
y = x.p
while y ≠ NIL and x == y.right
    x = y
    y = y.p
return y
```

$O(h)$

$O(h)$

Deletion in BST

- First Search in $O(h)$ → Then Delete
- But delete requires successor $O(h)$
- Thus the time complexity is $O(h)$

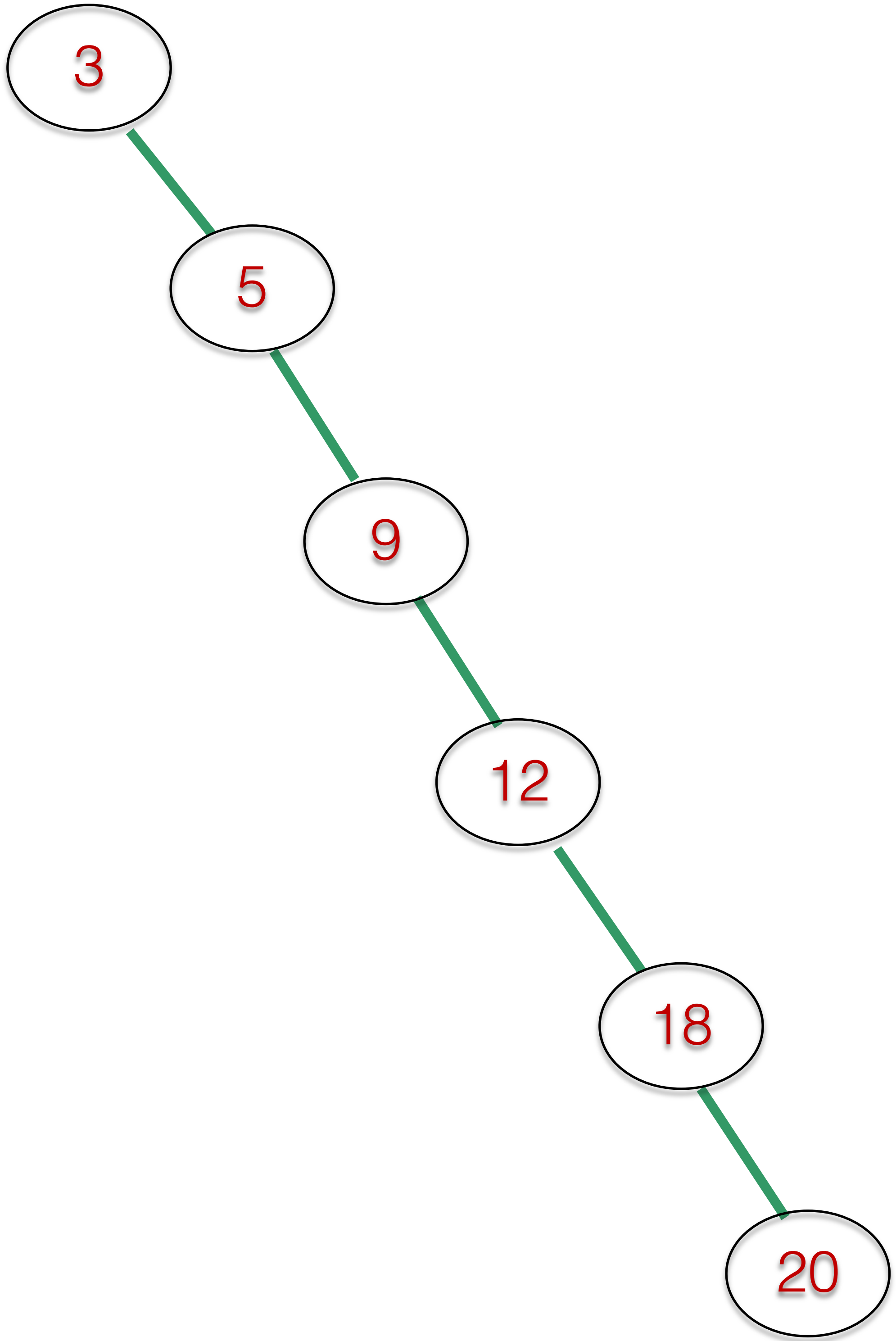
BST Analysis

- All the important operations of BST are $O(h)$
- So the question now is: $h = O(?)$

Discussion

- Look at what happens if we insert the following numbers in this order:

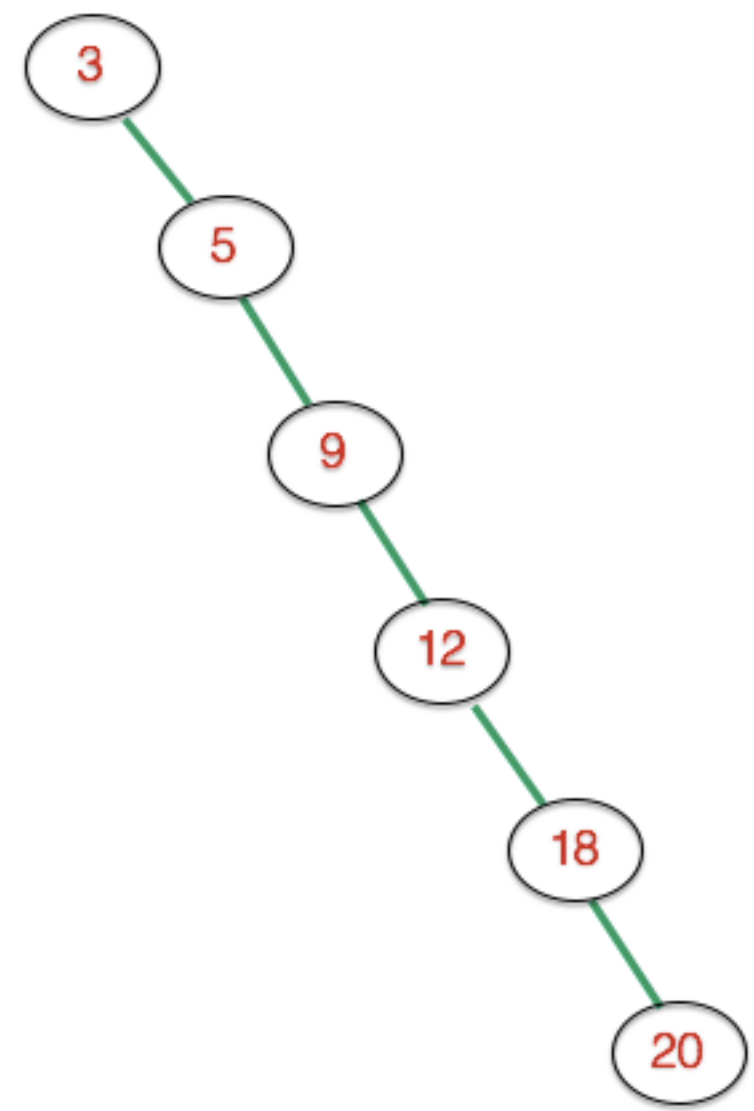
3 5 9 12 18 20



3 5 9 12 18 20

Degenerate Binary Trees

- The resulting tree is called a ***degenerate*** binary tree



- Note that it looks more like a linked list than a tree!

- Thus, $h = O(n)$

Degenerate Binary Trees

- How to avoid ***degenerate*** binary tree?
  - Random Binary Search Trees
  - Balanced Binary Search Trees

Random BST

- A tree that arises from inserting the keys in *random* order into an initially empty tree

- Theorem

  - The *expected* height of a randomly built binary search tree is $O(\log n)$

Did we achieve today's objectives?

- What is a Tree (as a data structure)?
- Learn about different types of trees and the associated properties, definitions and terminologies
- Special emphasis on Binary Search Trees
    - How does a BST work?
    - How to traverse in a BST?
    - Time complexity of a BST