Data Structures & Algorithms

Adil M. Khan
Professor of Computer Science
Innopolis University

Six ways to make people like you – Dale Carnegie

1. *Become genuinely interested in other people*
2. *Smile*

Recap

- What is sorting?
- Why learn about sorting?
- Properties of sorting algorithms
- Three simple sorting algorithms – Identifying their best and worst cases and 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 a BST

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
  - First need to find where the object will go
  - 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
- On the other hand, finding a particular element is slow
  - Must start at the beginning of the linked list
  - Visit each element until you find the one you're looking for
- What if we made the linked list ordered? Will it help?

Trees to Rescue

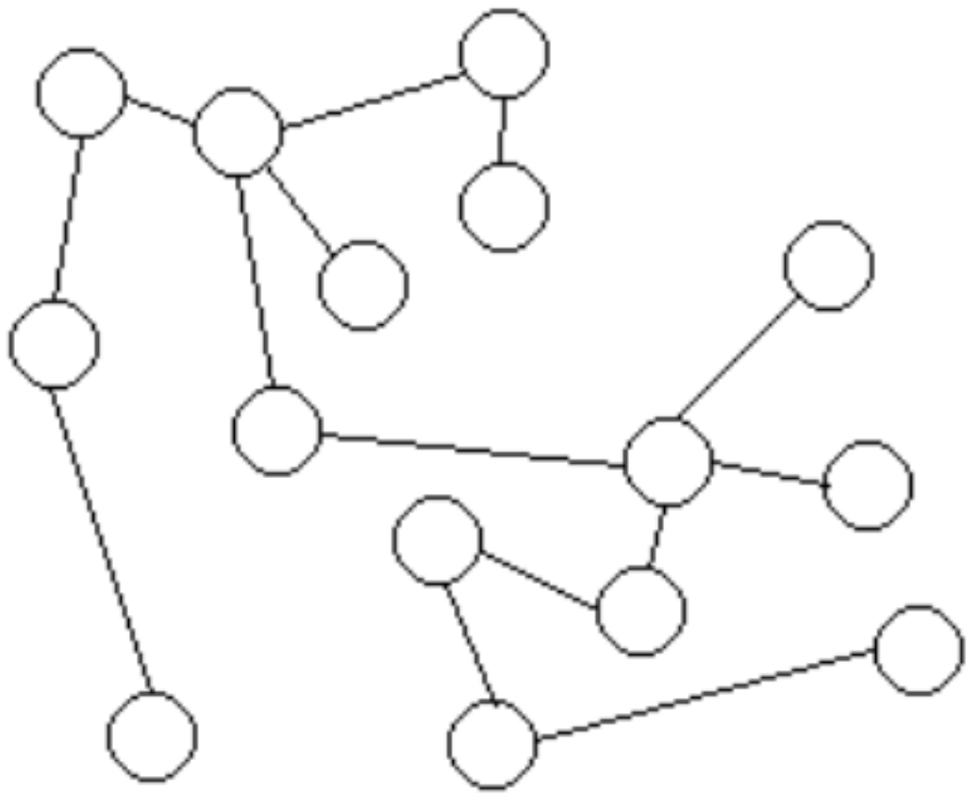
- It would be nice to have a data structure
  - With quick insertions and deletions of a linked list
  - And, the quick searching of an ordered array

Trees to Rescue

- Trees provide both these characteristics
- Our main focus will be a binary tree
- But let’s first start discussing trees in general

Tree

- Consists of nodes connected by edges



Node

(a.k.a. vertex) a data element in the tree

Edge

(a.k.a. branch, or link) a connection between two nodes

Empty tree

has zero nodes

Size

the size of a tree is the number of nodes

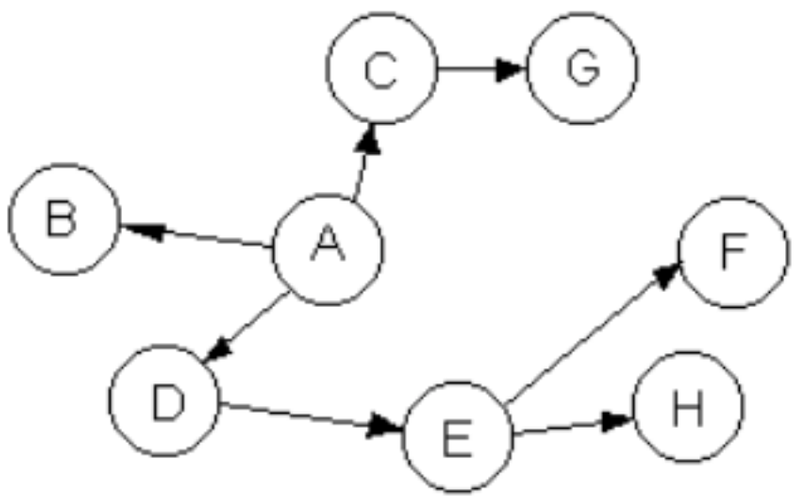
Path

a sequence $n_0e_1n_1e_2n_2...e_kn_k$ where $k \geq 0$ and e_i connectes n_{i-1} and n_i.

Oriented Trees

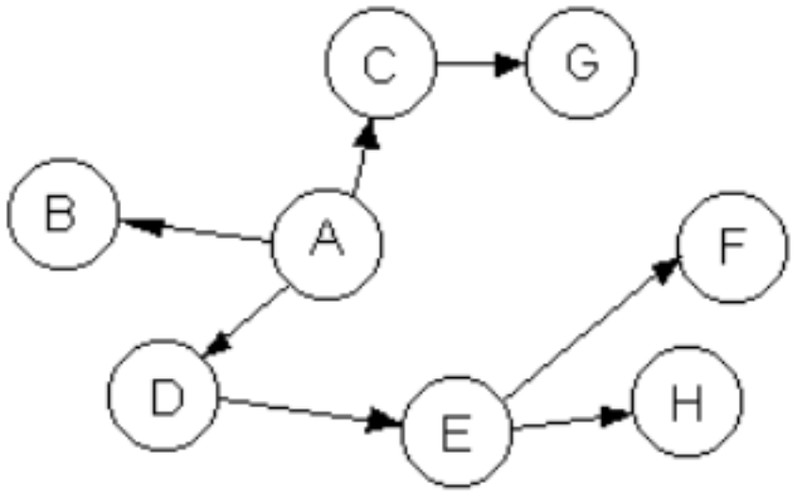
Oriented Trees

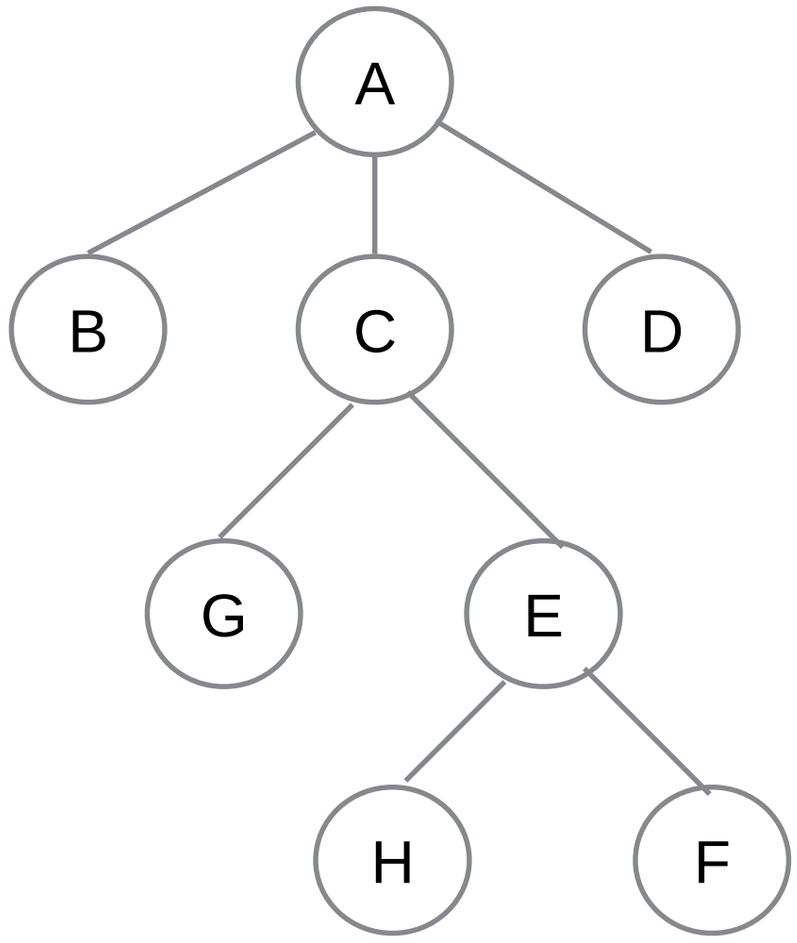
- A tree used to represent a hierarchical data.
- All edges are directed outward from a distinguished node called the root node



Oriented Trees

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





Definitions

Parent

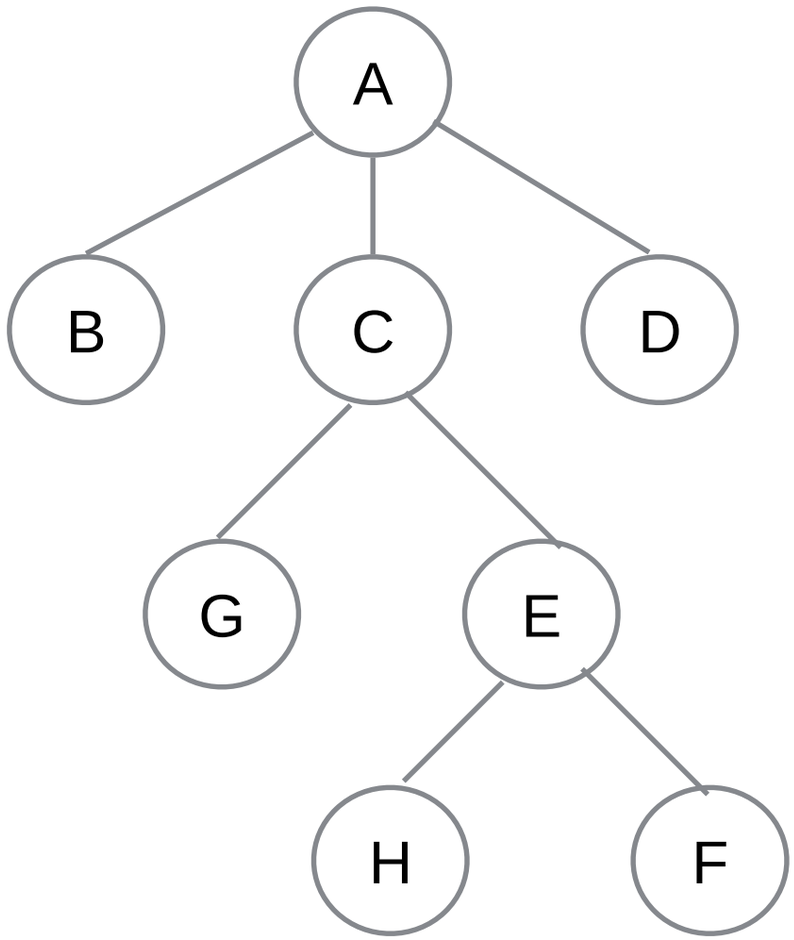
A is the parent of B and C and D

Children

The children of A are B and C and D

Siblings

B and C are siblings



Definitions

Root node

The only node without a parent

Leaf node

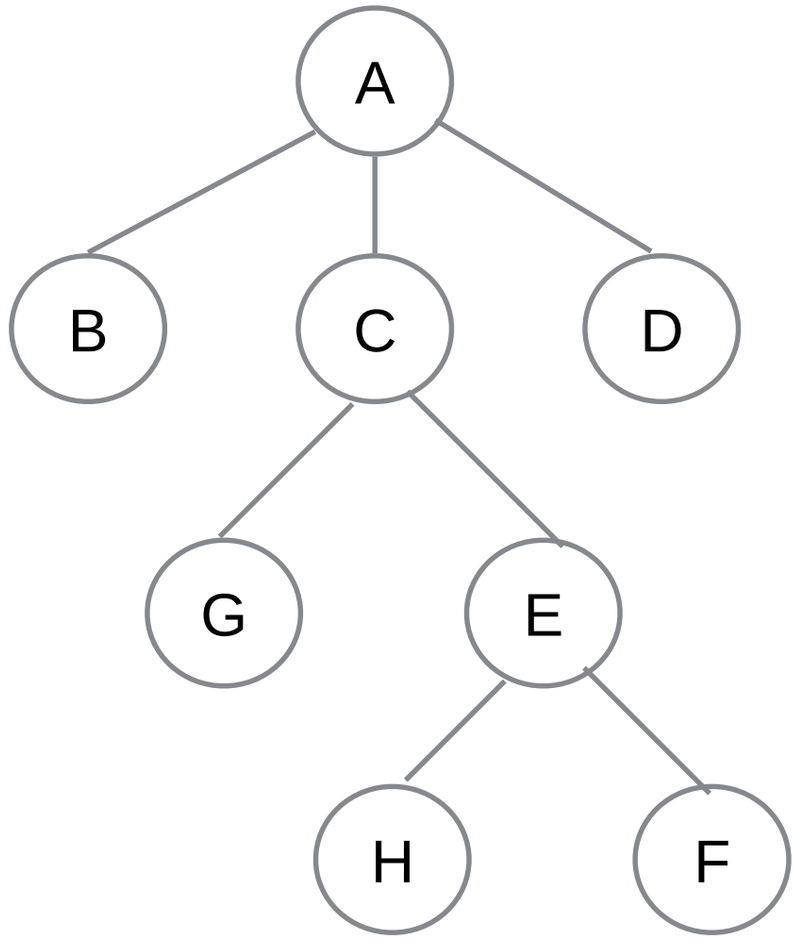
A node without children

Degree(n)

The number of children of n

Degree(t)

The greatest degree of the nodes in t



Definitions

Level(n)

Level(n) = if n is the root then 0 else 1 + Level(Parent(n))

Depth(n)

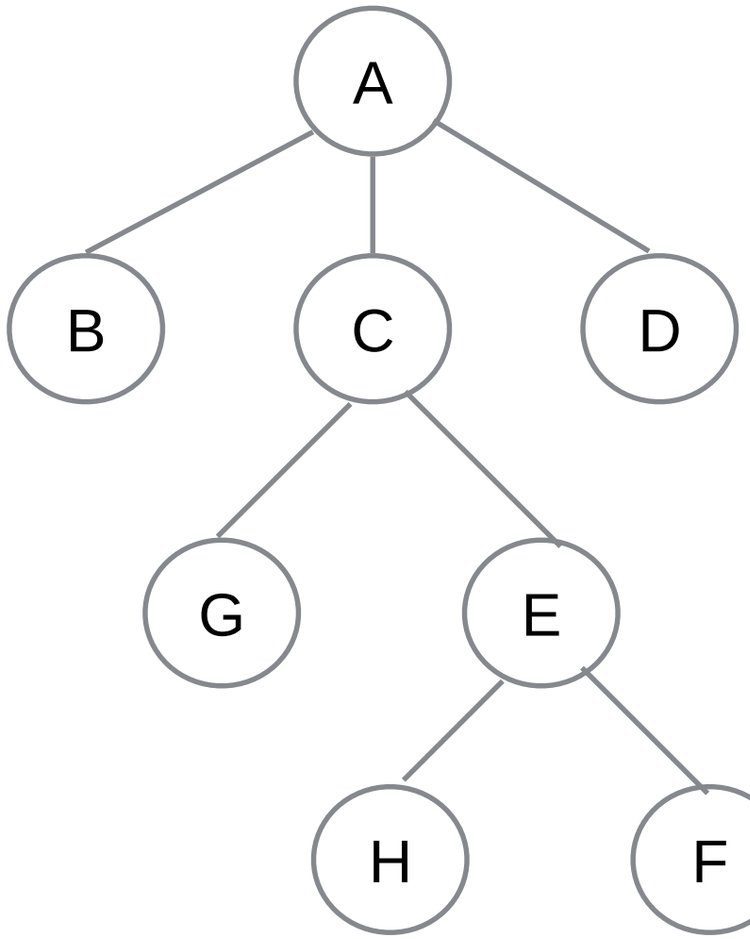
Same as Level(n)

Height(n)

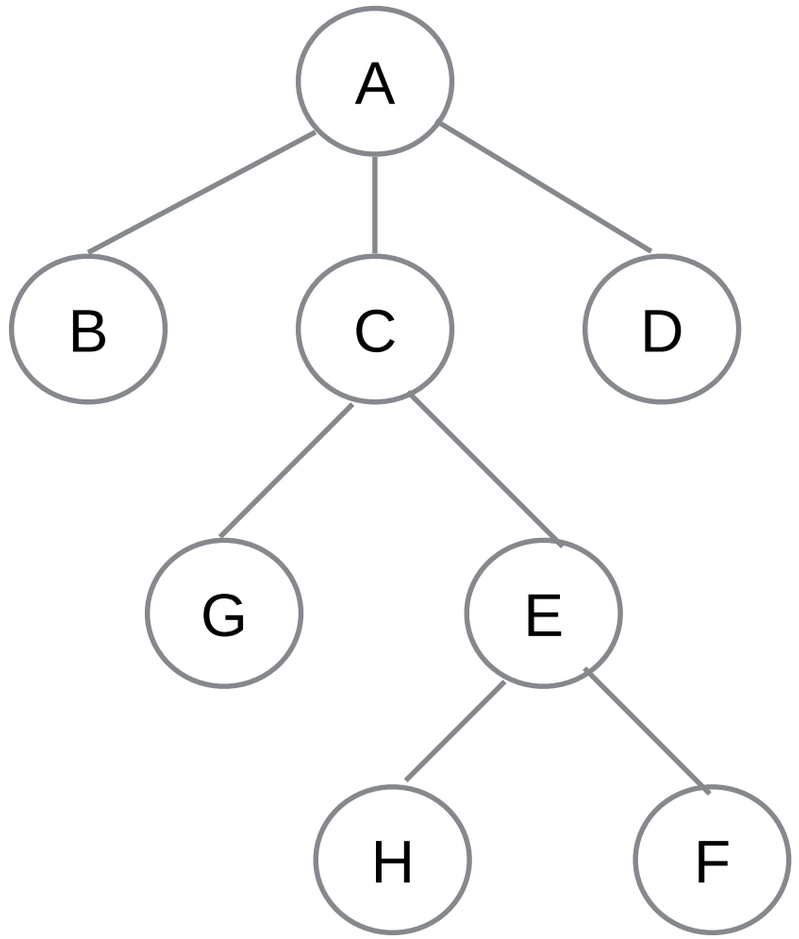
Maximum length over all paths from n to a leaf

Height(t)

The height of t's root node



Definitions



Ancestors(n)

Ancestors(n) = if n is the root then { } else {Parent(n)} union Ancestors(Parent(n))

Descendants(n)

The set of all nodes reachable from n following the edges leaving it in the direction of the arrows

Definitions

PathLength(t)

The sum of all the depths of all the nodes in t

Empty Tree

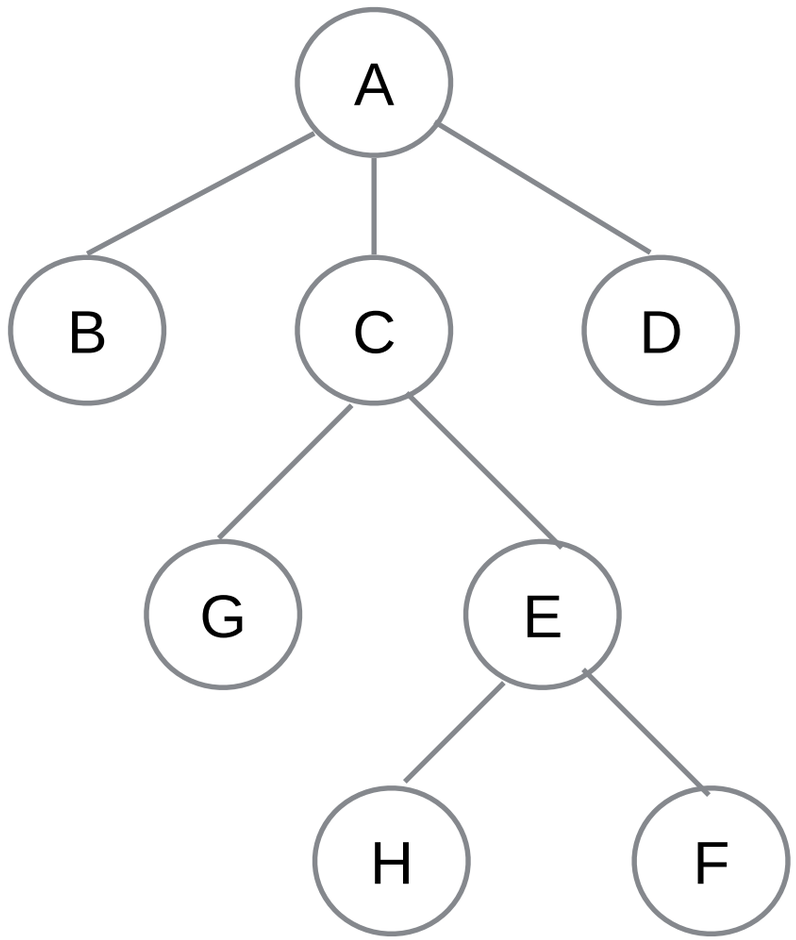
A tree with zero nodes

Singleton Tree

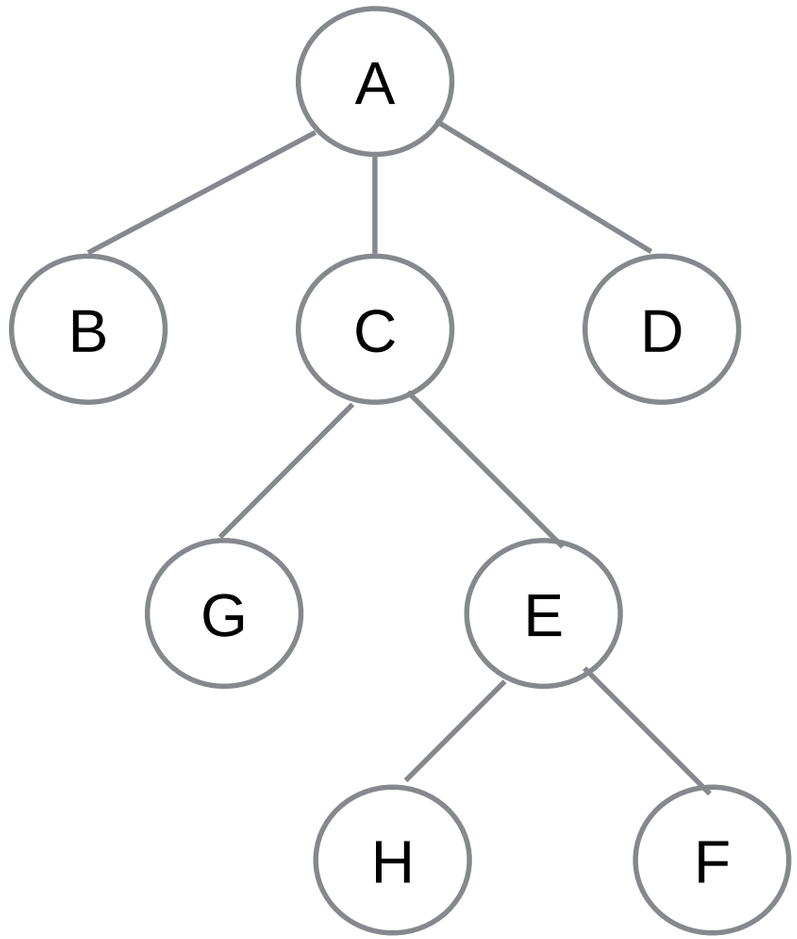
A tree with only one node

Subtree(n)

The subtree rooted at n



Definitions



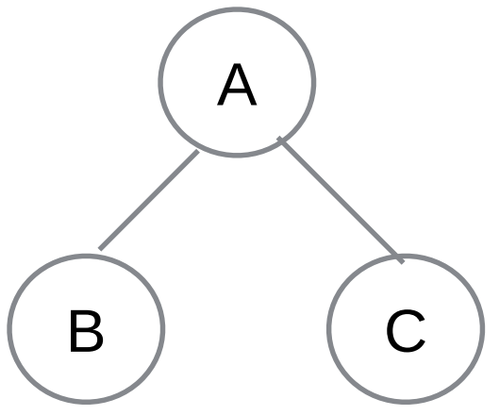
k-node

A node with k children. A 0-node is a leaf. A 1-node has exactly one child. A 2-node has exactly two children. Etc.

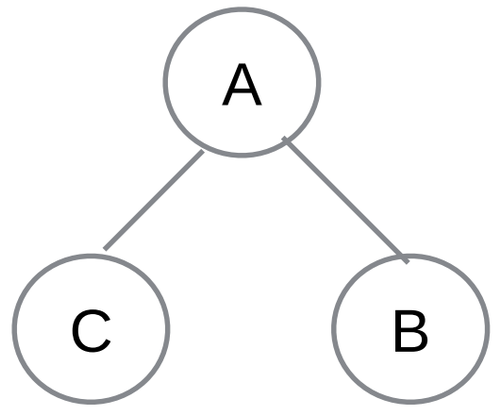
Ordered Trees

Ordered Trees

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



T1



T2

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

Types of Ordered Trees

- Fibonacci Tree
- Binomial tree
- k-ary Tree

k-ary Trees

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

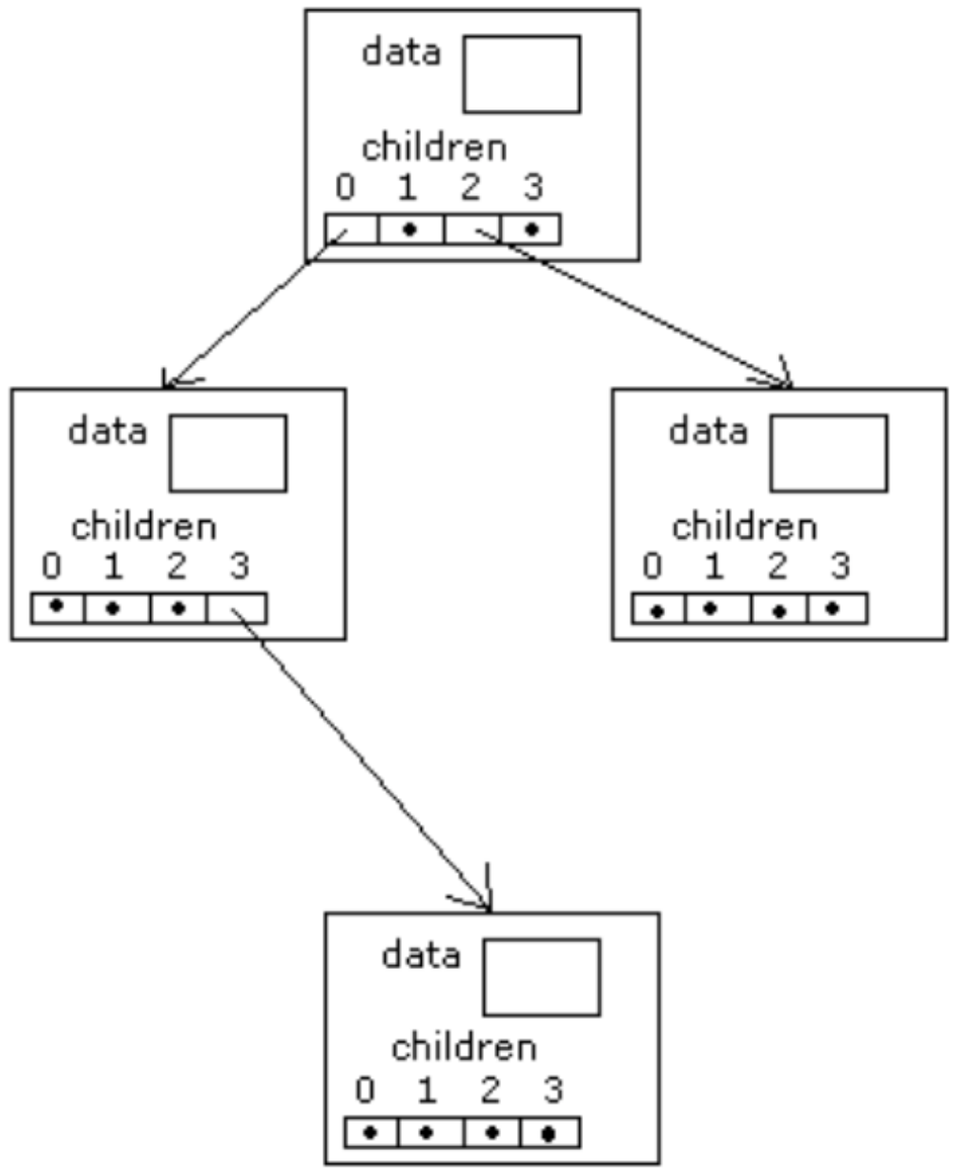
- Maximum number of children for a node is k

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**

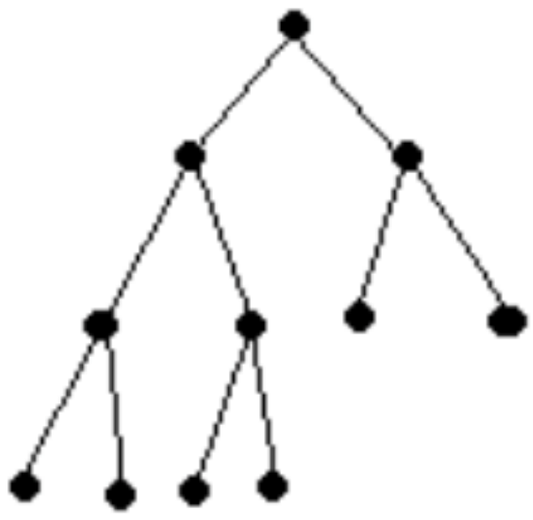
k-ary Tree Representation

```
class Node {

        private Object data;

        private Node[] children;

        .

        .

        .

}
```



Complete k-ary Trees

- Filled out on every level, expect perhaps on the last one
- All nodes on the last level, should be as far to left as possible



Complete trees can be packed into an array with no wasted space

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

Binary Tree

External nodes - have no subtrees (referred to as leaf nodes)

Internal nodes - always have two subtrees

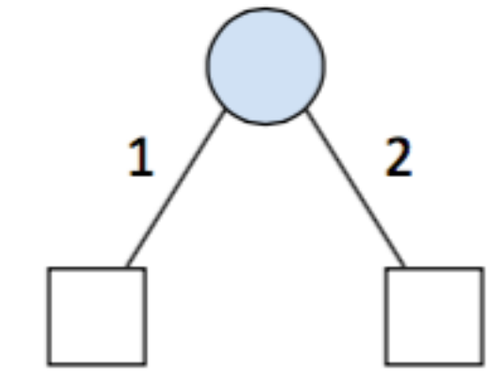
Figures mostly used for internal and external nodes

External nodes are usually omitted

Binary Tree

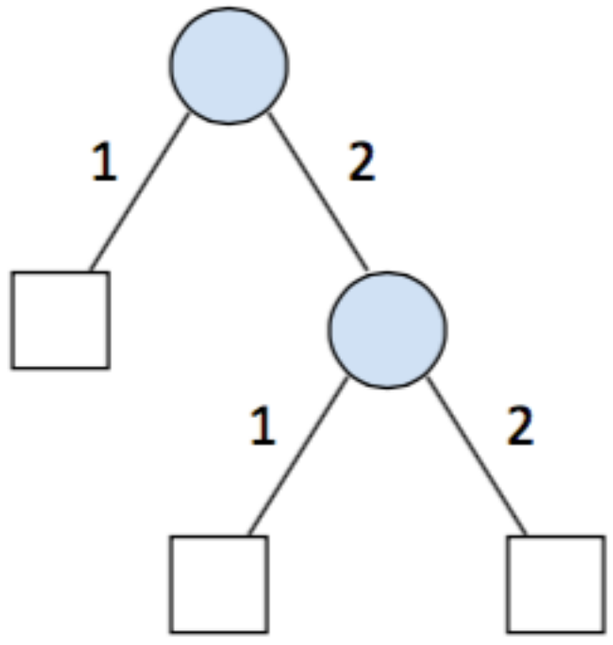
Binary Tree of zero nodes

Binary Tree



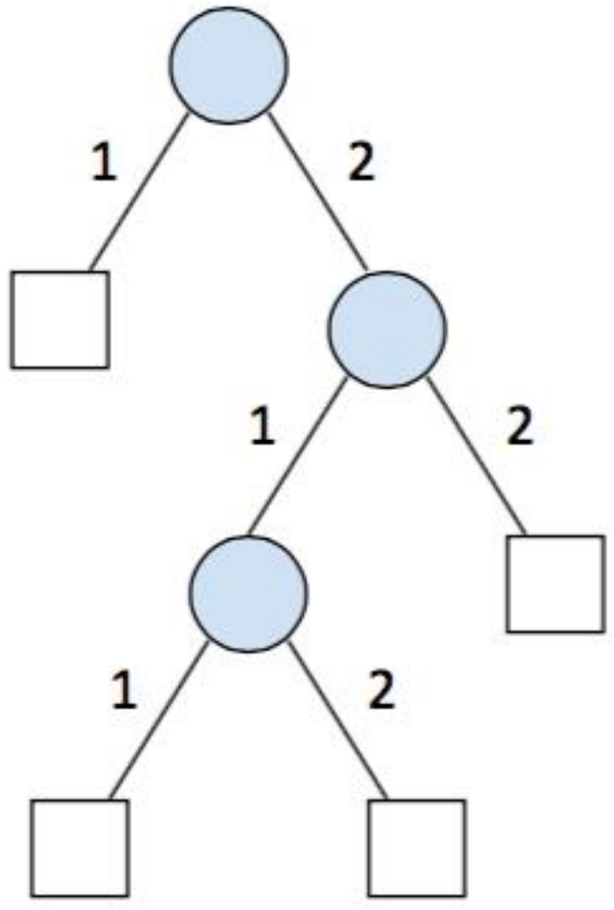
Binary Tree of 1 nodes

Binary Tree



Binary Tree of 2 nodes

Binary Tree



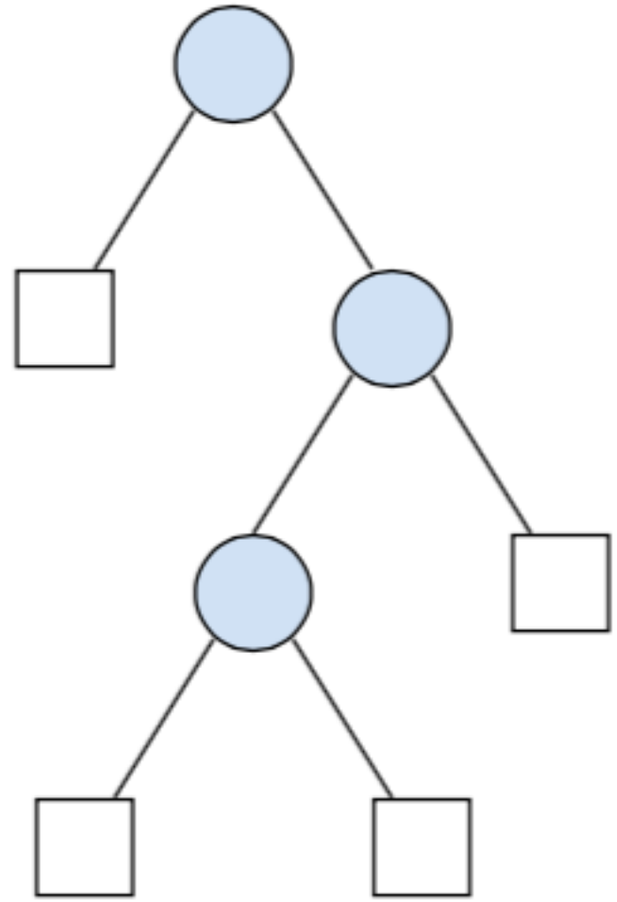
Binary Tree of 3 nodes

Why Binary Trees

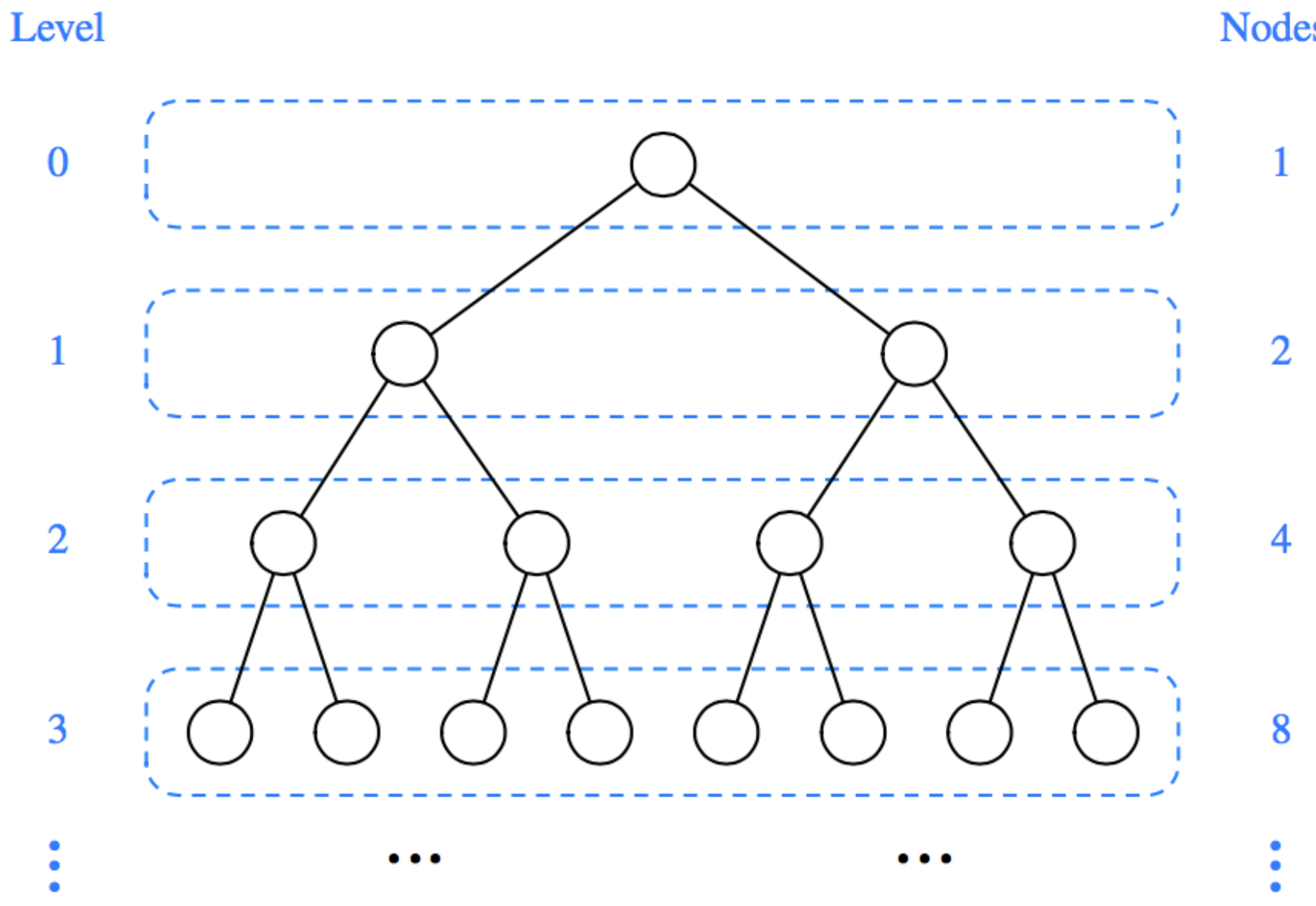
- Binary trees are a bit simpler and easier to understand than trees with a large or unbounded number of children
- Binary tree nodes have an elegant linked representation with "left" and "right" subtrees
- Binary trees form the basis for efficient representations of sets, dictionaries, and priority queues

Binary Tree Properties

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



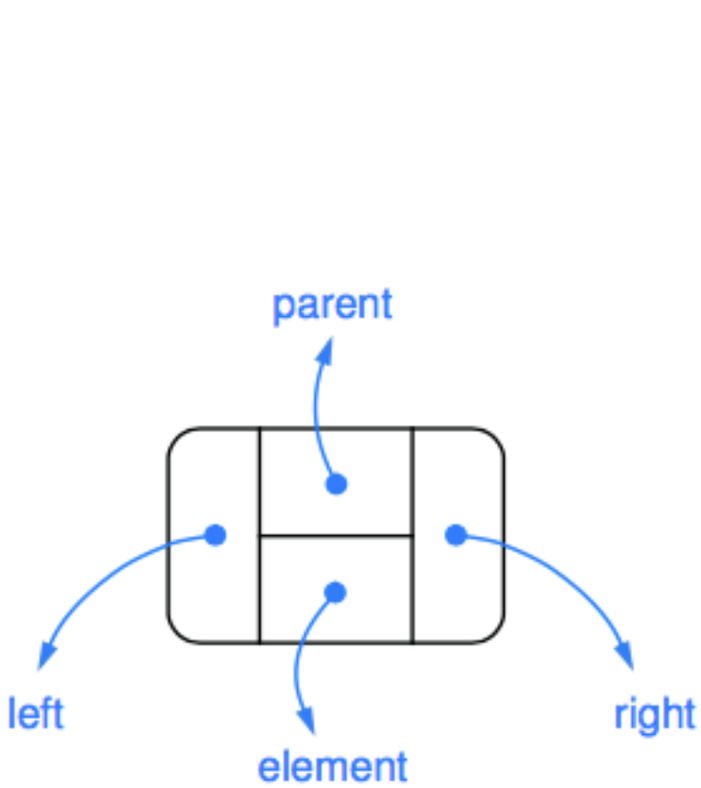
Binary Tree Properties



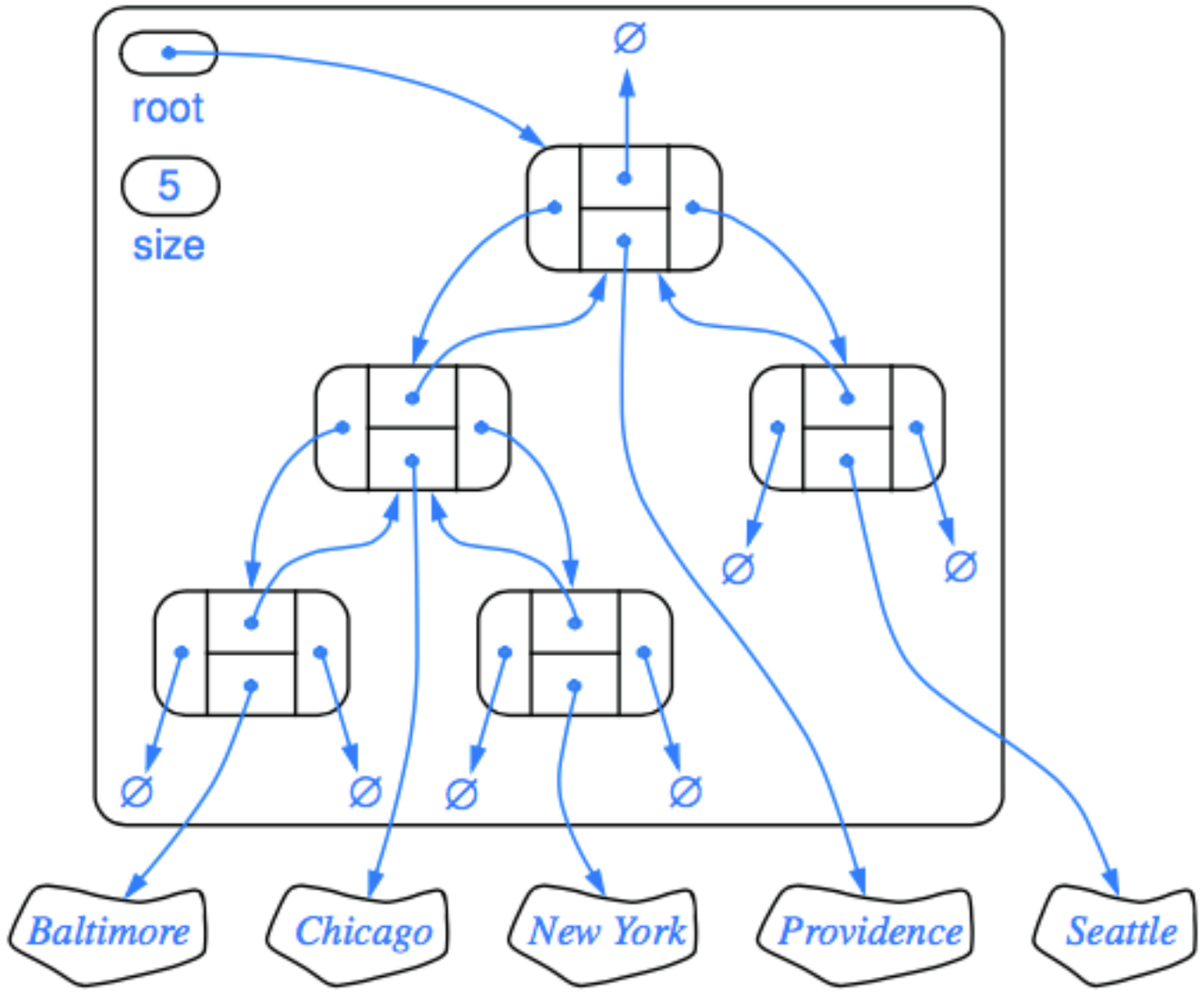
Maximum number of nodes in the levels of a binary tree

Binary Tree Representations

- Linked Structure



(a)



(b)

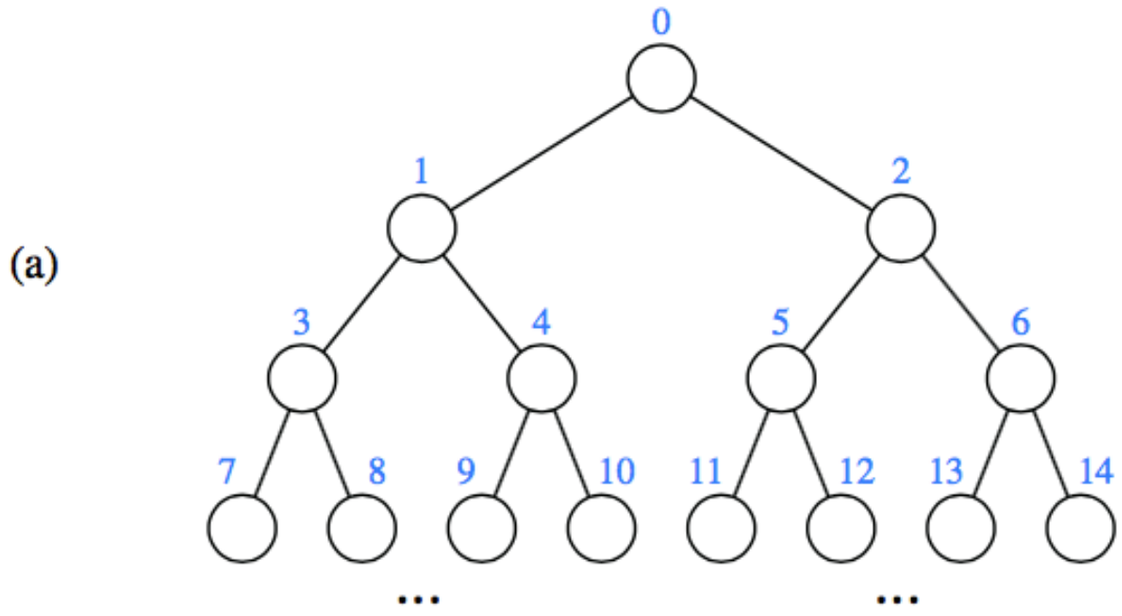
Binary Tree Representations

- Array Based 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 $f(p) = 2f(q) + 2$.

Binary Tree Representations

- Array Based Structure

(a) general scheme



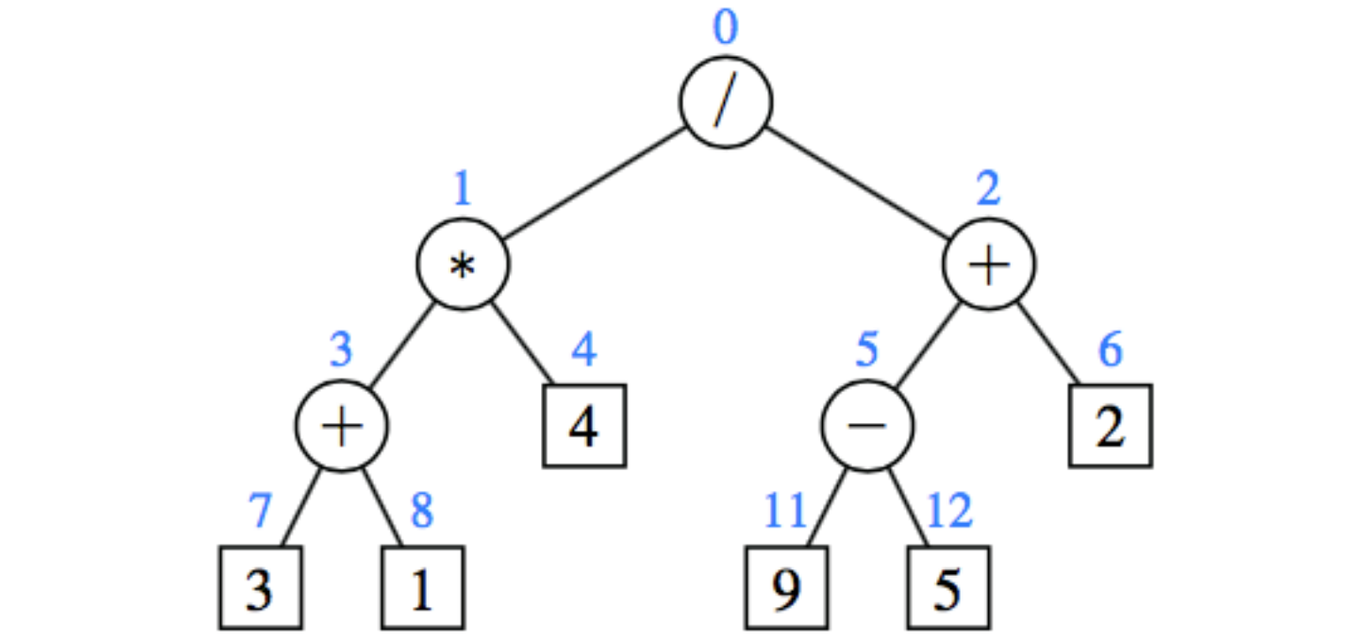
If p is the root of T, then $f(p) = 0$

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 $f(p) = 2f(q) + 2$.

Binary Tree Representations

- Array Based Structure



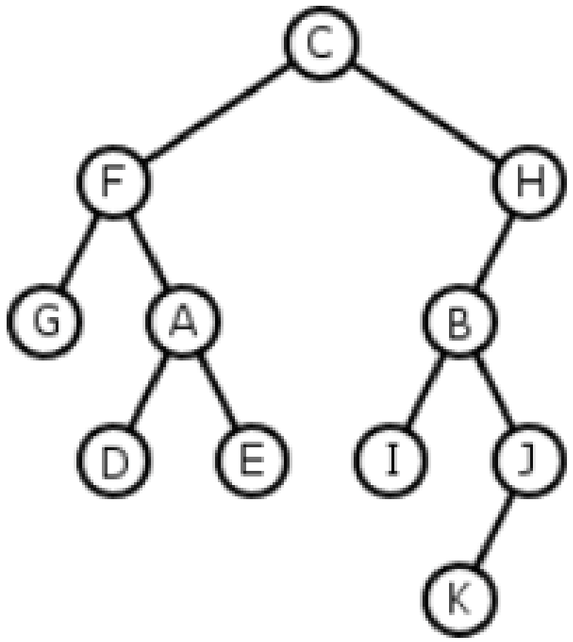
/	*	+	+	4	−	2	3	1			9	5		
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

Tree Traversal Algorithms

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



- **Preorder:** C F G A D E H B I J K
- **Inorder:** G F D A E C I B K J H
- **Postorder:** G D E A F I K J B H C

Preorder Traversal

```
Algorithm preorder(p):
    perform the "visit" action for position p     { this happens before any recursion }
    for each child c in children(p) do
        preorder(c)                               { recursively traverse the subtree rooted at c }
```

Code Fragment 8.12: Algorithm preorder for performing the preorder traversal of a subtree rooted at position p of a tree.

Postorder Traversal

```
Algorithm postorder(p):
    for each child c in children(p) do
        postorder(c)                    { recursively traverse the subtree rooted at c }
    perform the "visit" action for position p    { this happens after any recursion }
```

Code Fragment 8.13: Algorithm postorder for performing the postorder traversal of a subtree rooted at position p of a tree.

Inorder Traversal

```
Algorithm inorder(p):
    if p has a left child lc then
        inorder(lc)                    { recursively traverse the left subtree of p }
    perform the "visit" action for position p
    if p has a right child rc then
        inorder(rc)                    { recursively traverse the right subtree of p }
```

Code Fragment 8.15: Algorithm inorder for performing an inorder traversal of a subtree rooted at position p of a binary tree.

Binary Search Trees

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 every node holds a value > every value in its left subtree and < every value in its right subtree

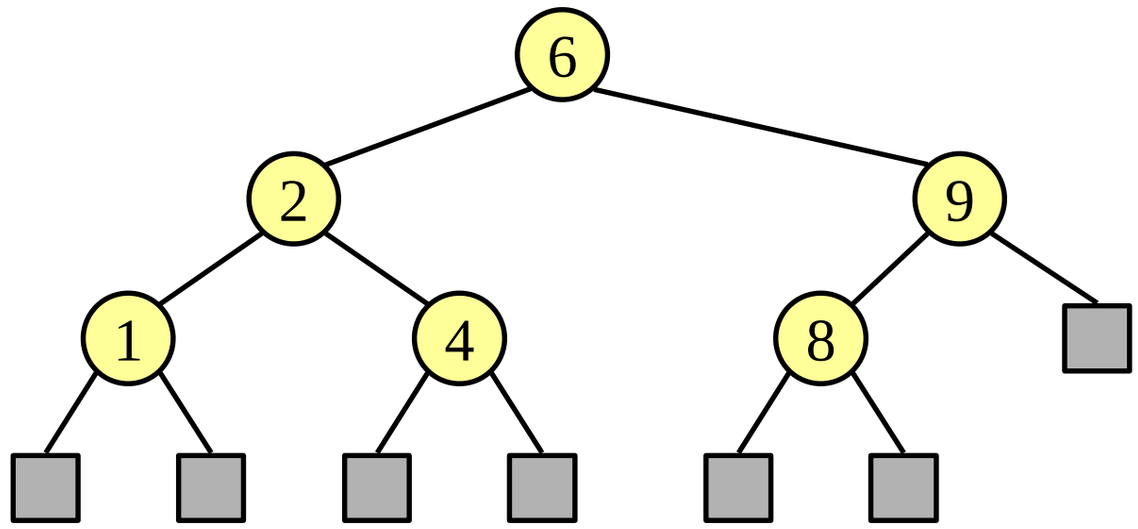
Assuming no duplicates are allowed

Binary Search Trees

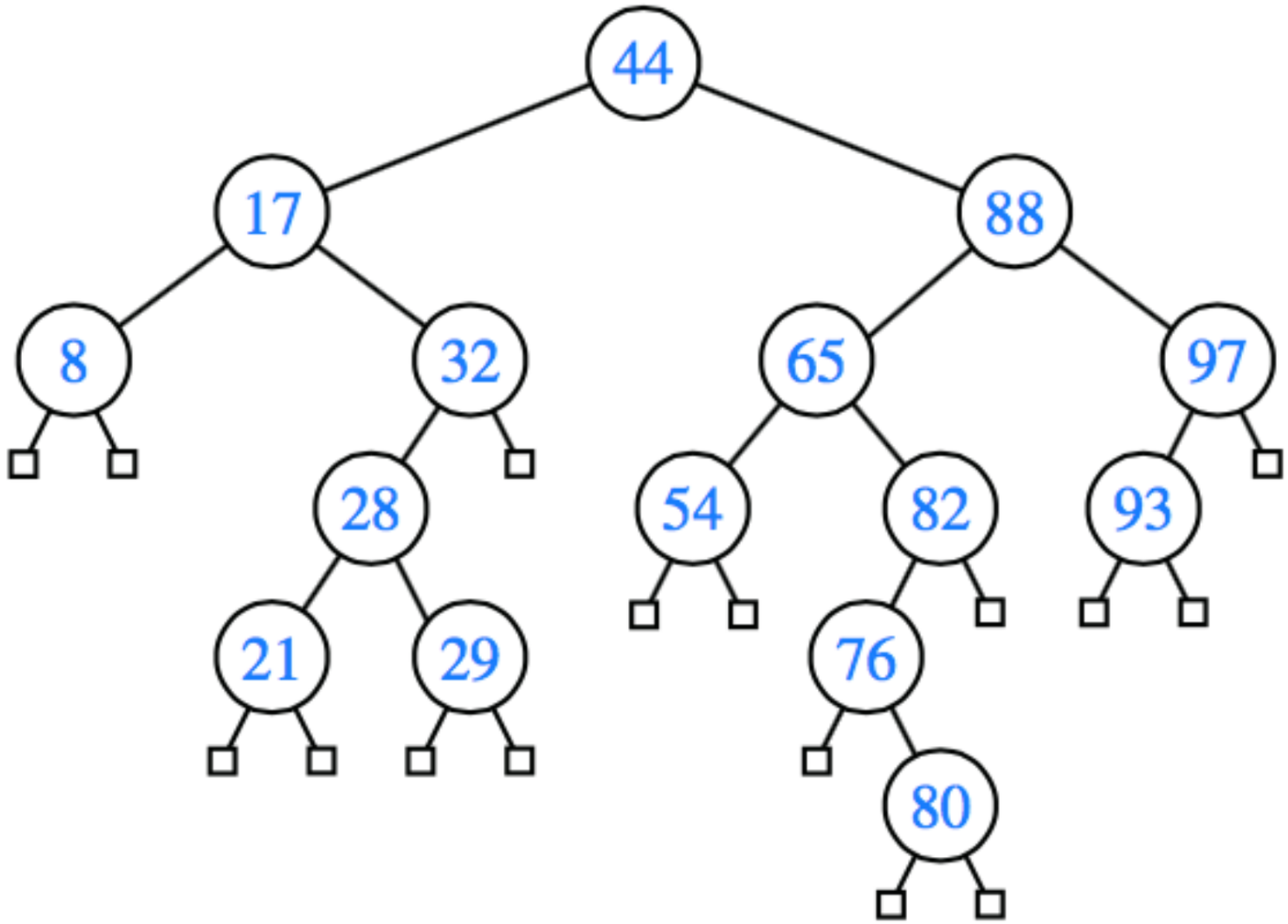
- Let u, v, and w be three nodes such that u is in the left subtree of v and w is in the right subtree of v. We have

$$key(u) < key(v) < key(w)$$

- An inorder traversal of a binary search trees visits the keys in increasing order



Example



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

BST Operations

- A binary search tree is a special case of a binary tree
  - So, it has all the operations of a binary tree

- It also has operations specific to a BST:
  - 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

- 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

Searching in a BST

- Algorithm to search for an item in a BST

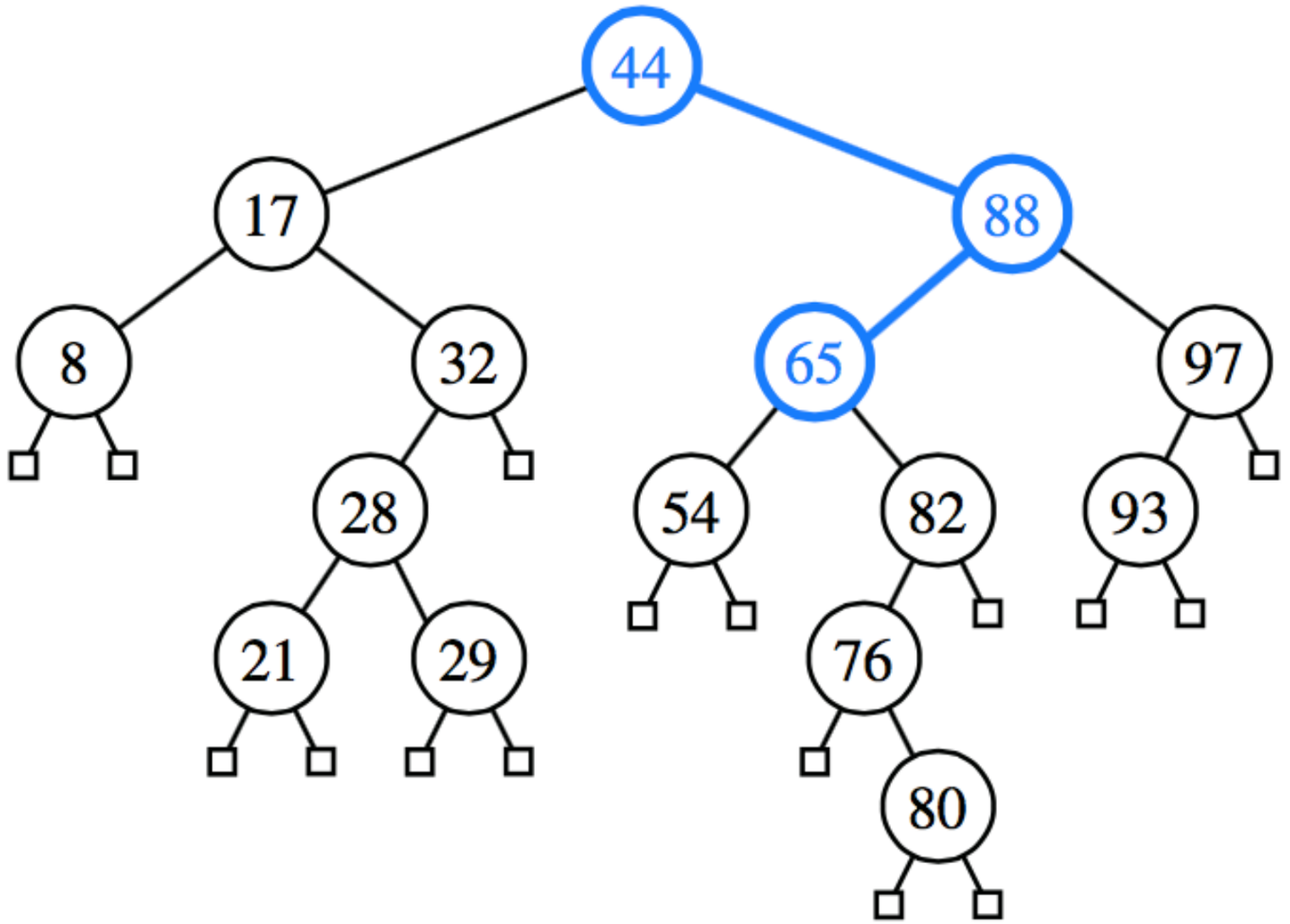
  - Compare data item to the root of the (sub)tree
  - If data item = data at root, found
  - If data item < data at root, go to the left; if there is no left child, data item is not in tree
  - If data item > data at root, go to the right; if there is no right child, data item is not in tree

Searching in a BST

```
Algorithm TreeSearch(p, k):
    if p is external then
        return p                                   {unsuccessful search}
    else if k == key(p) then
        return p                                   {successful search}
    else if k < key(p) then
        return TreeSearch(left(p), k)              {recur on left subtree}
    else {we know that k > key(p)}
        return TreeSearch(right(p), k)             {recur on right subtree}
```

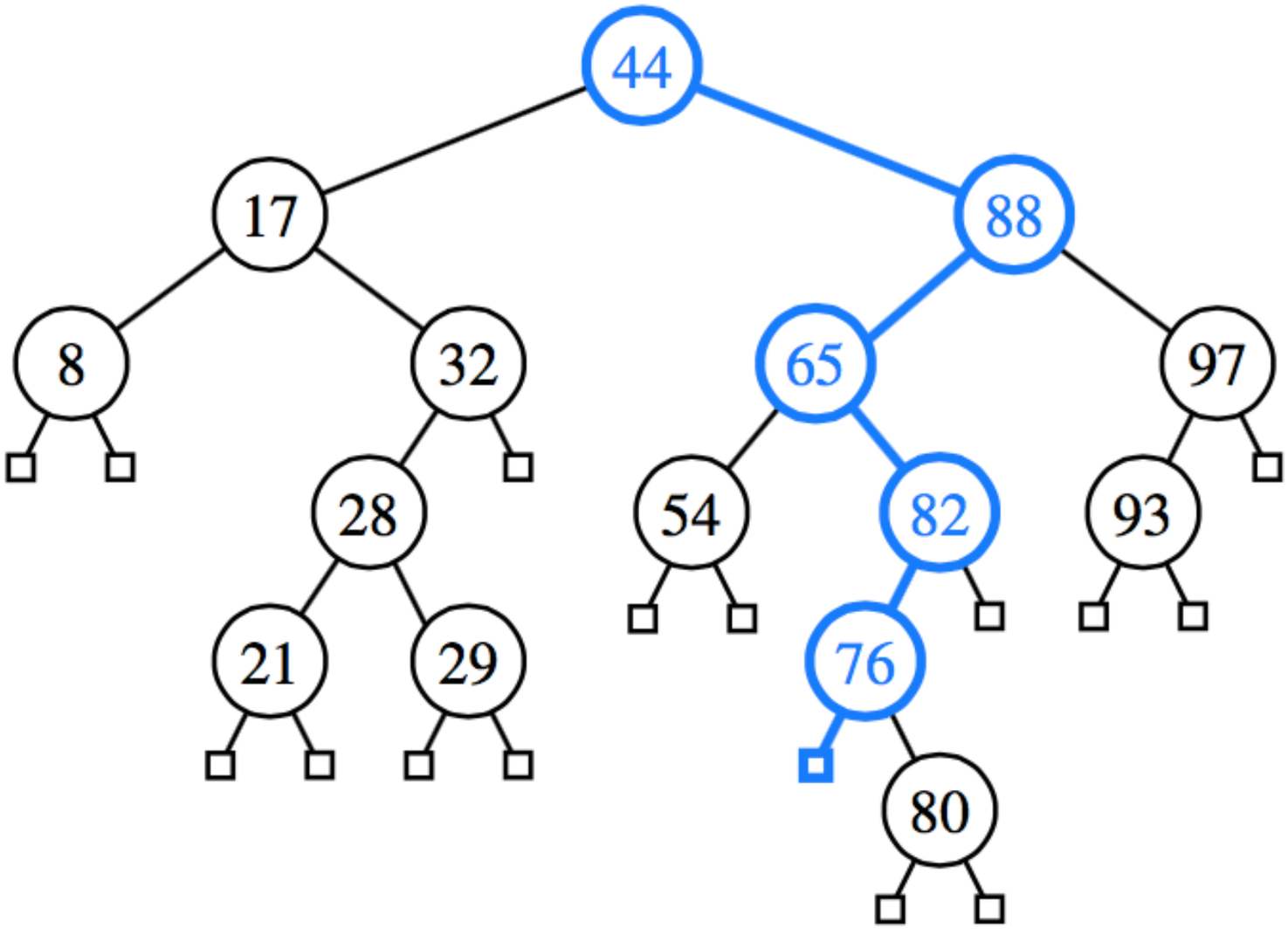
Code Fragment 11.1: Recursive search in a binary search tree.

Searching in a BST



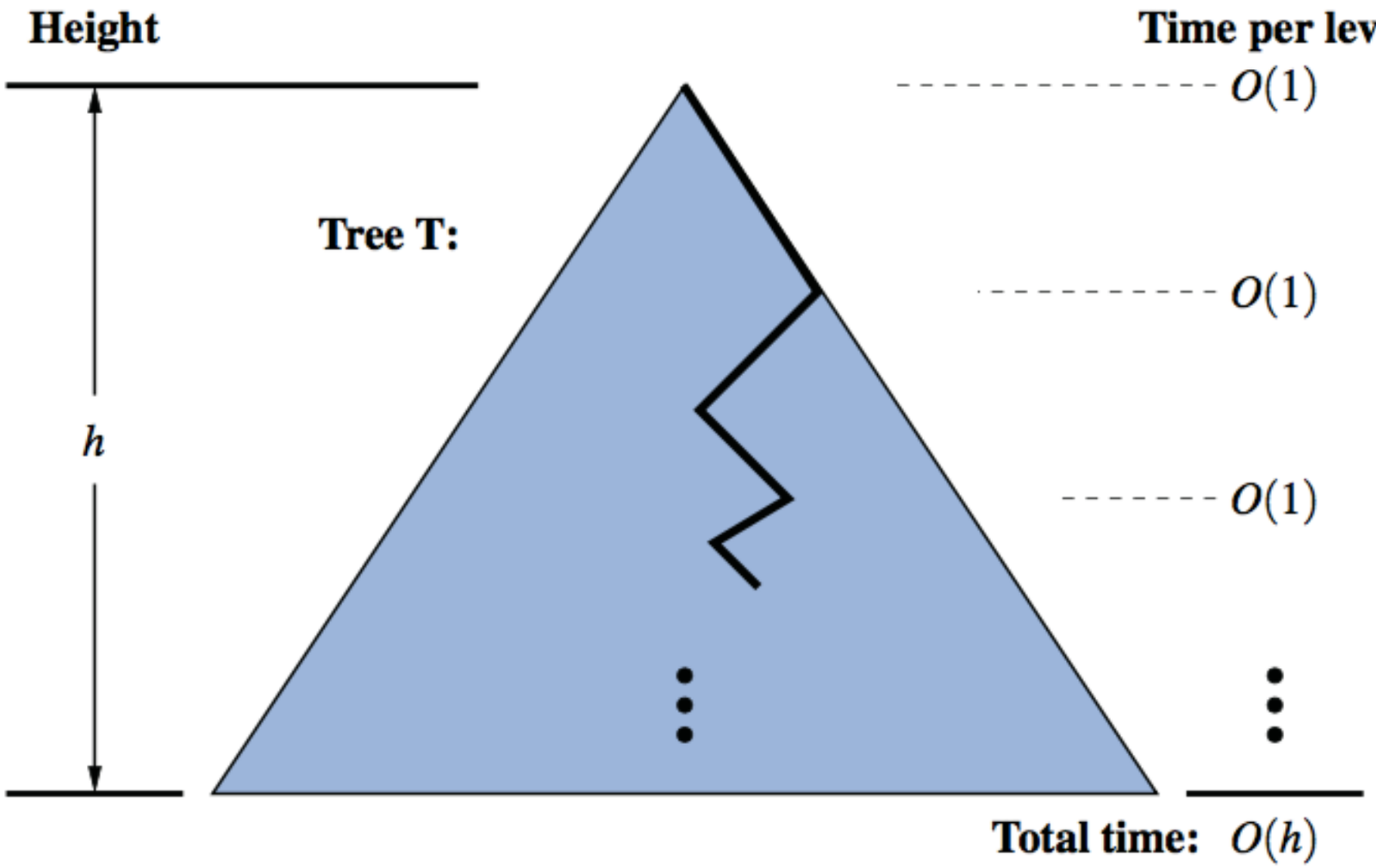
A successful search for key 65 in a binary search tree

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



Insertions in a BST

- To ***add*** an item to a BST:
  - Follow the algorithm for searching, until there is no child
  - Insert at that point

- So, new node will be added as a leaf

- (We are assuming no duplicates allowed)

Insertions in a BST

Algorithm TreeInsert(k, v):
  Input: A search key k to be associated with value v
  p = TreeSearch(root(), k)
  if k == key(p) **then**
    Change p's value to (v)
  else
    expandExternal(p, (k, v))

Code Fragment 11.2: Algorithm for inserting a key-value pair into a map that is represented as a binary search tree.

Insertions in a BST

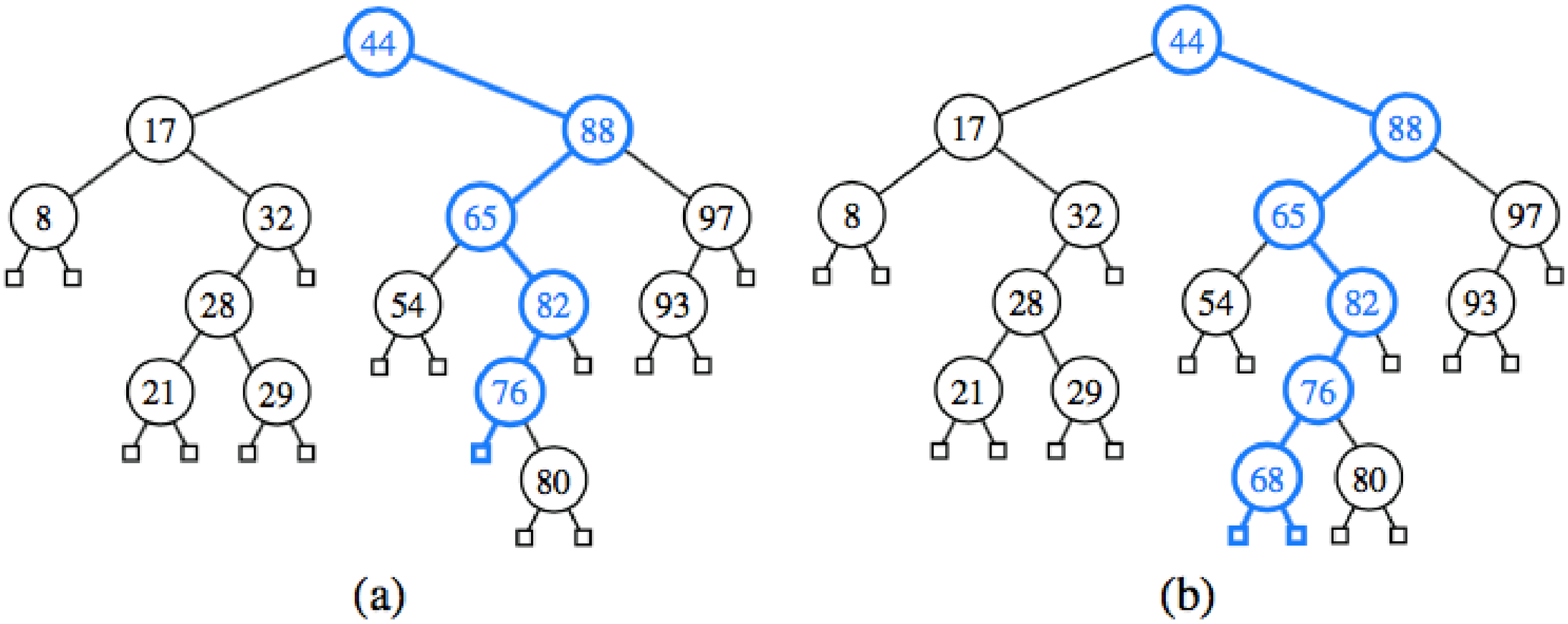


Figure 11.4: Insertion of an entry with key 68 into the search tree of Figure 11.2. Finding the position to insert is shown in (a), and the resulting tree is shown in (b).

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:

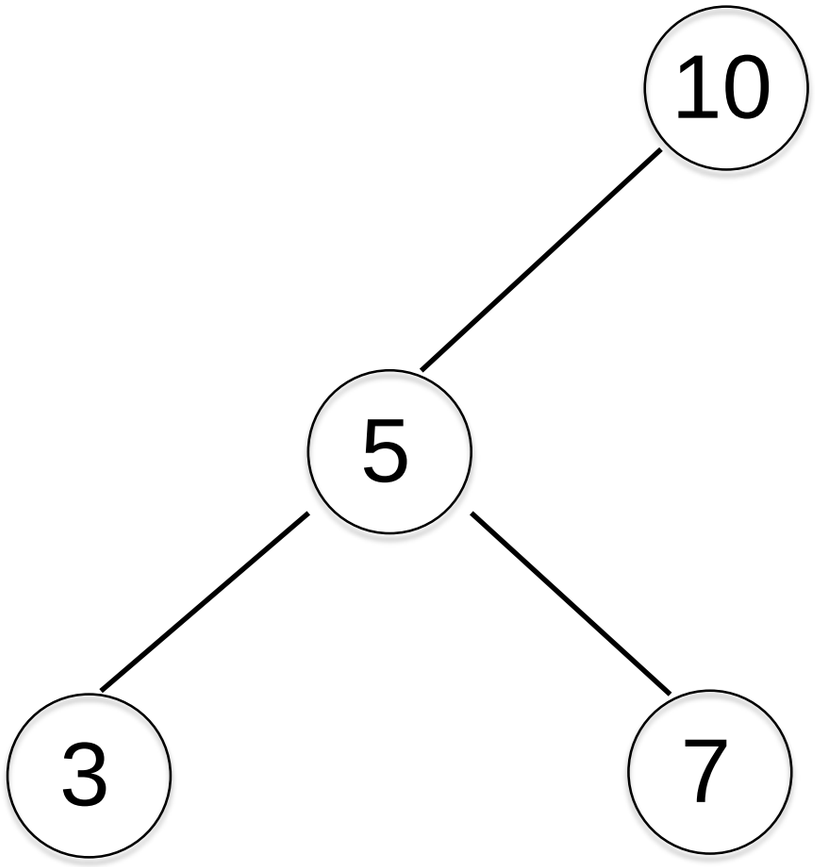
// Now there can be 4 cases

// The easiest case, the node has no children – is a leaf
Case 1: if the node has 2 empty subtrees
replace the link in the parent with null

Deletions in a BST

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

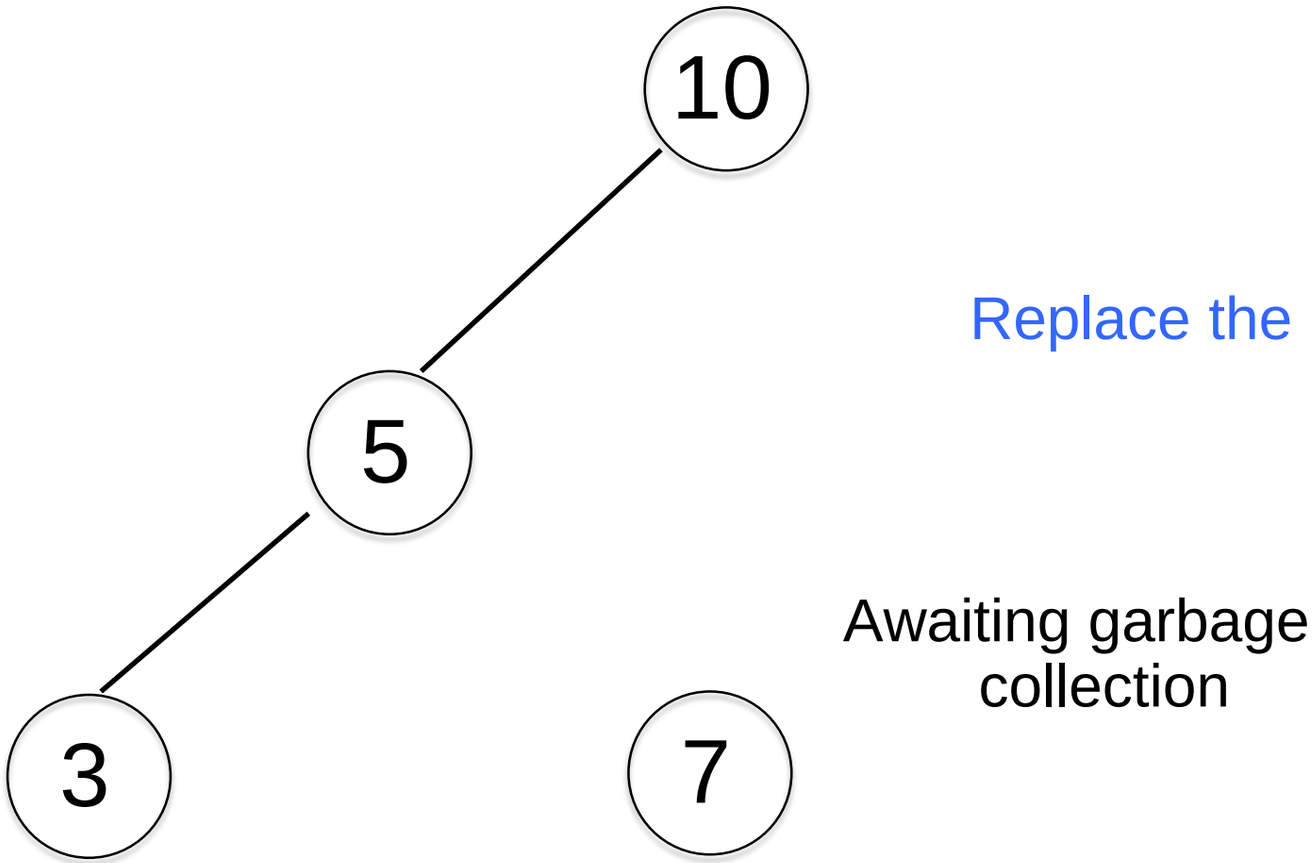
Case 1: the node has no children



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

// Next are the cases with one child

Case 2: if the node has no left child

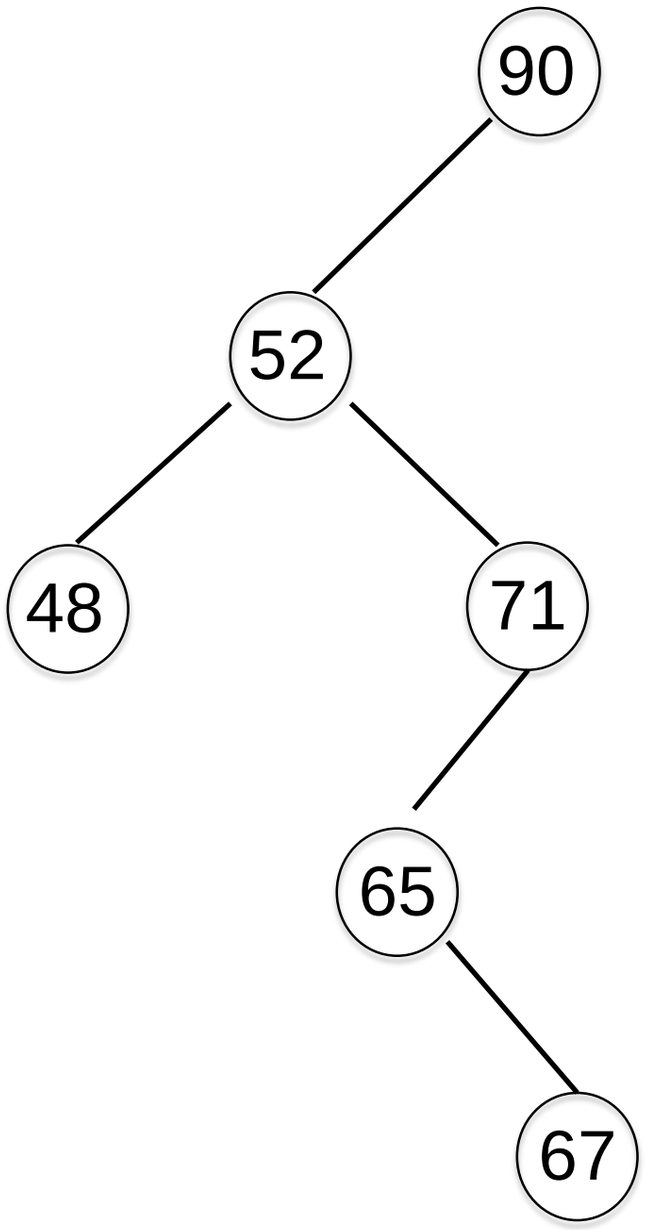
- link the parent of the node
- to the right (non-empty) subtree

Case 3: if the node has no right child

- link the parent of the target
- to the left (non-empty) subtree

Deletions in a BST

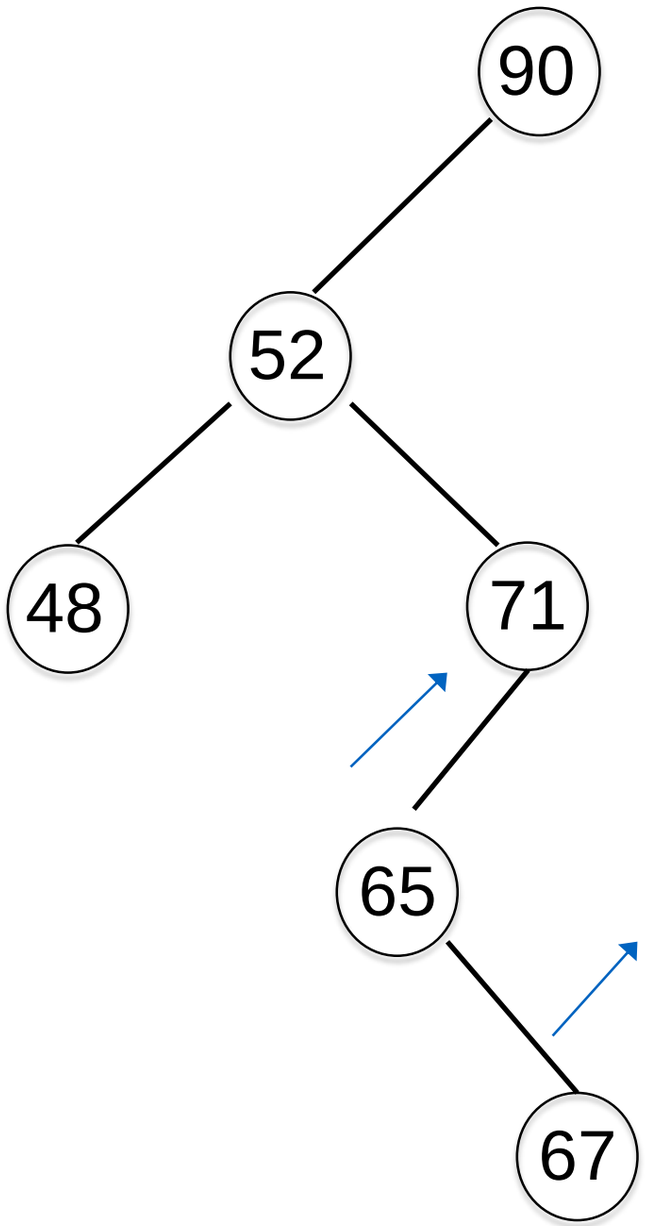
Case 2: the node has only a left child



Let’s delete the node with key 71

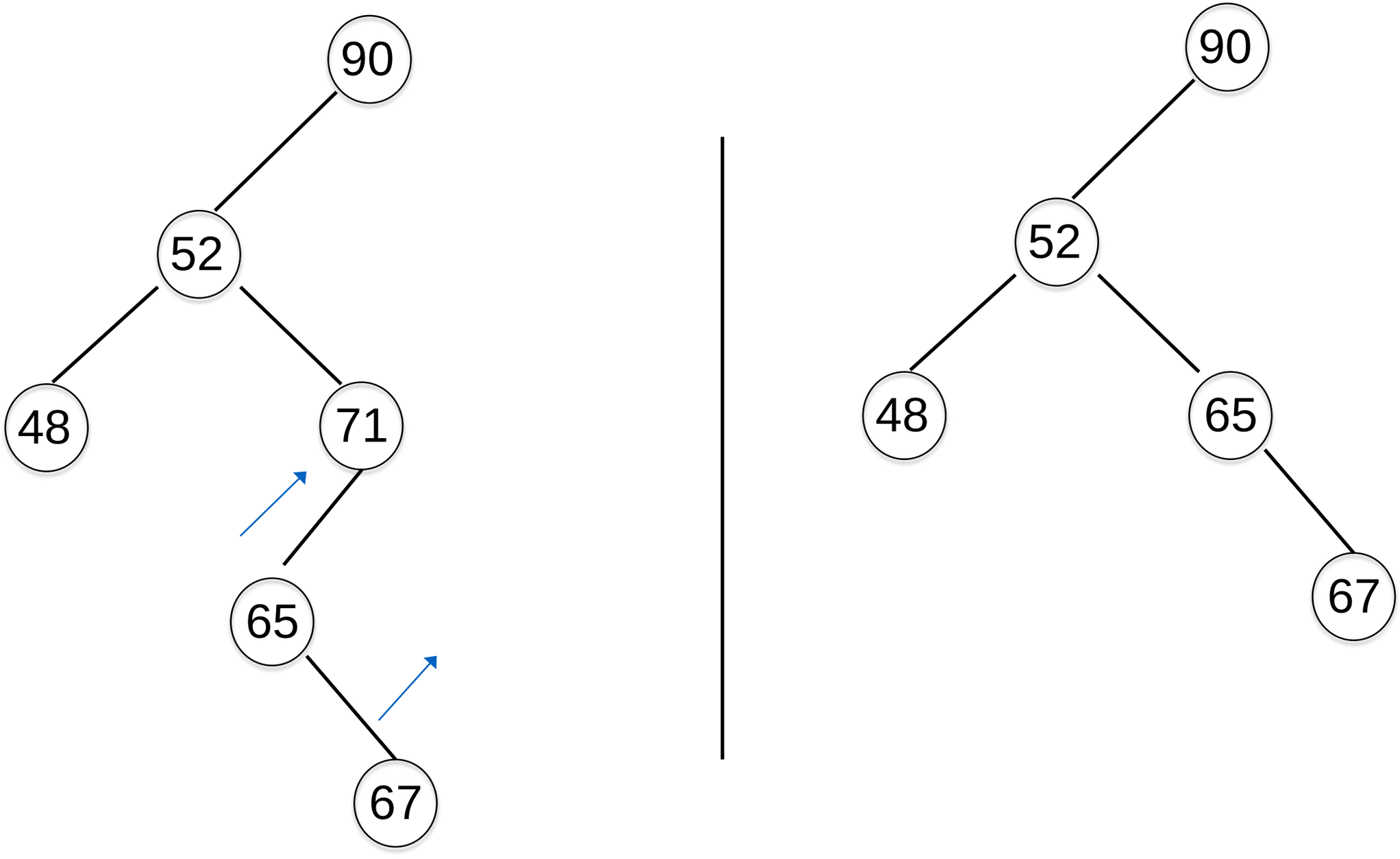
Deletions in a BST

Case 2: the node has only a left child



Deletions in a BST

Case 2: the node has only a left child

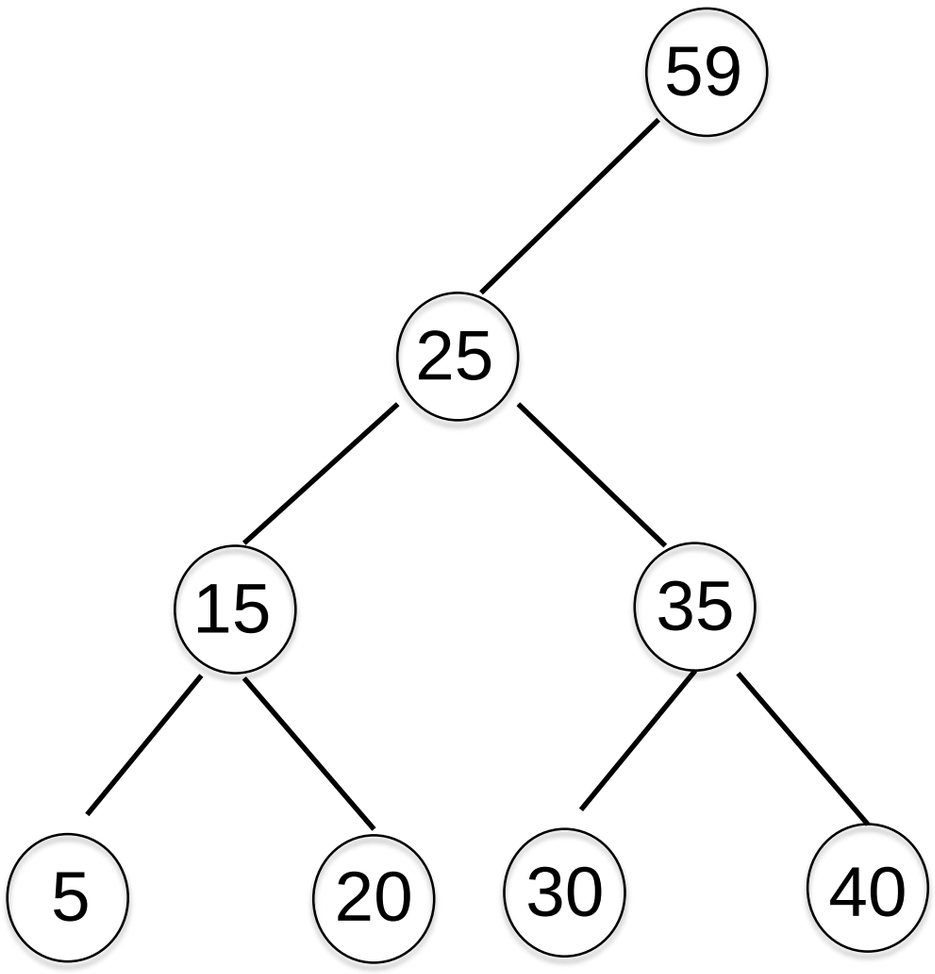


Deletions in a BST

// The most difficult case, the node has two children
// deleting this node will leave two children in trouble

Case 4: if the node has a two children

Deletions in a BST



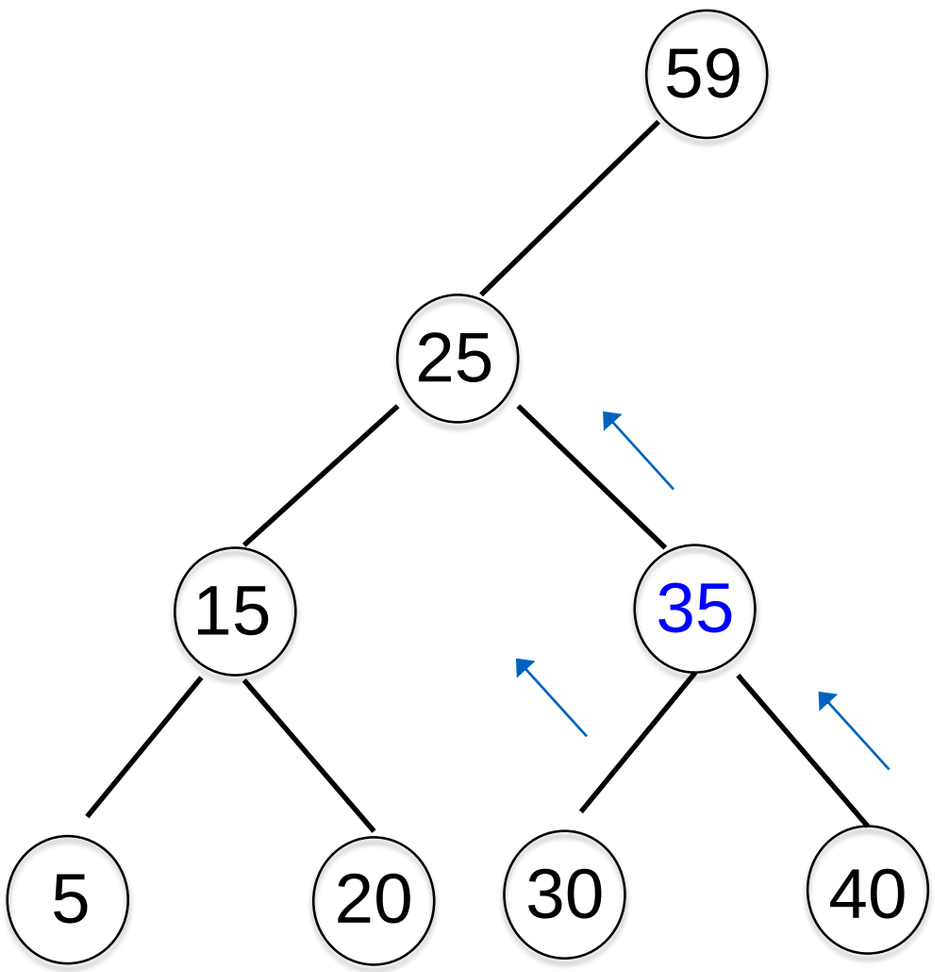
// Why is this a difficult case

Case 4: if the node has a two children

Let's delete 25 – Two choices

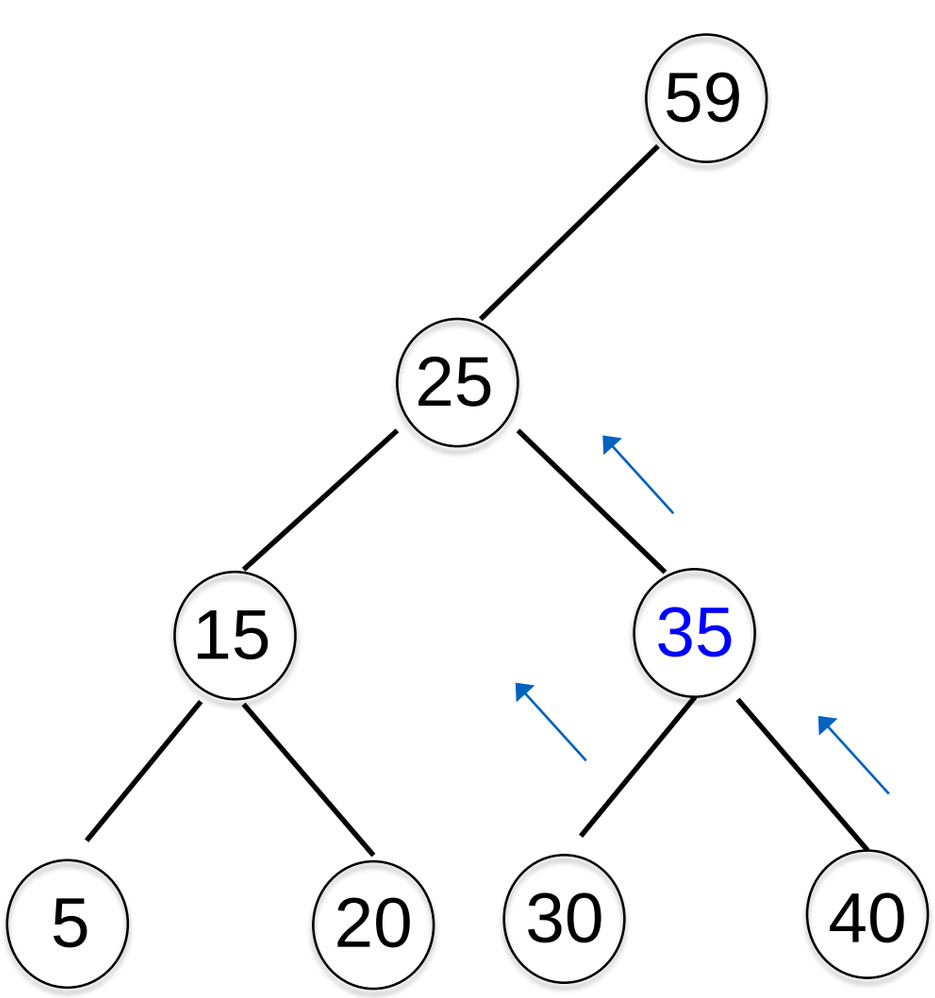
- Replace with root of left sub-tree, OR
- Replace with the root of right sub-tree

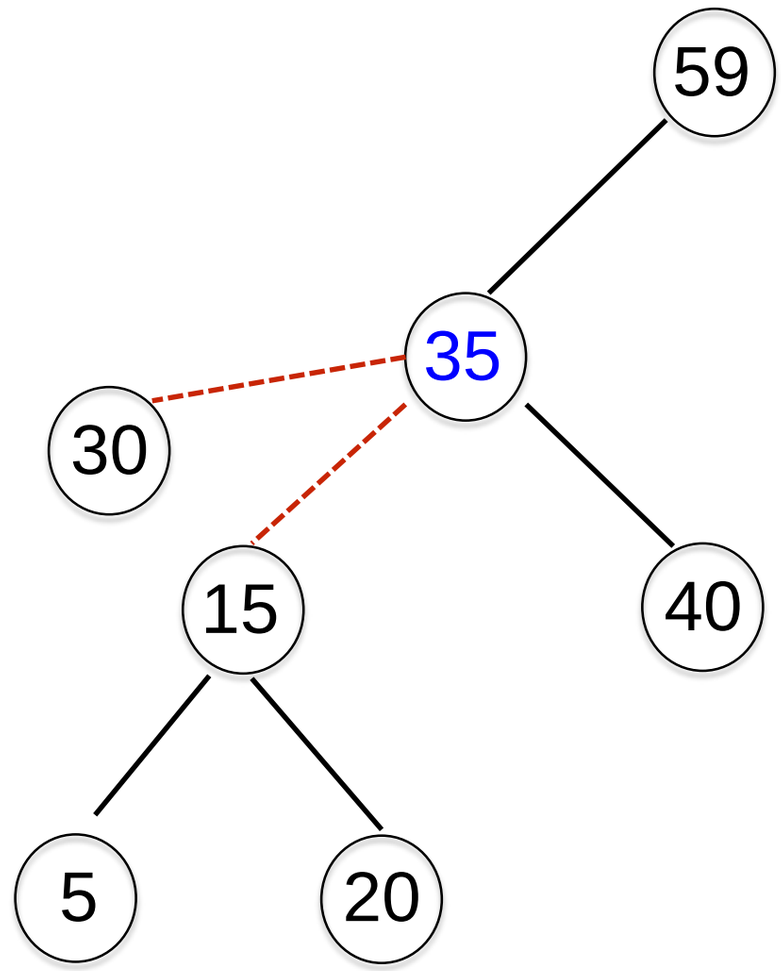
Deletions in a BST



- Replace with the root of right sub-tree

Deletions in a BST





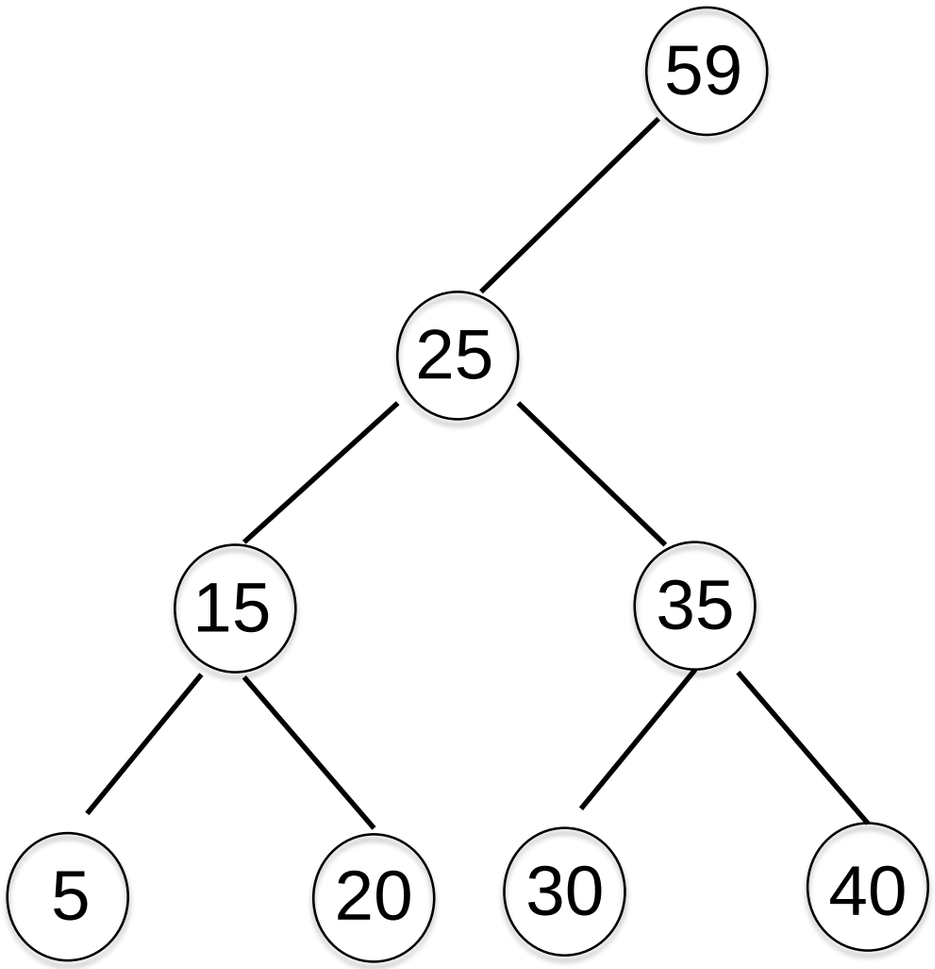
Deletions in a BST

// Thus a trick is needed

Case 4: if the node has a two children

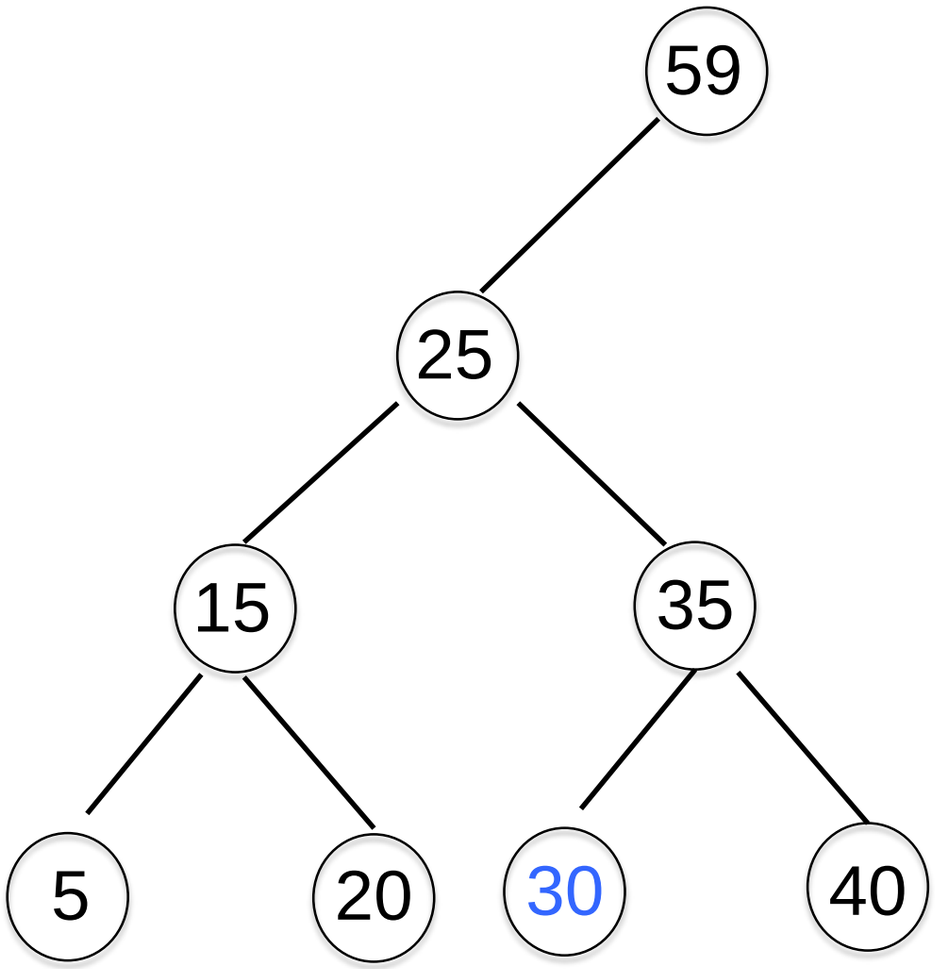
Replace the node with its **predecessor** or **successor** from **the inorder traversal** of the tree, and delete that node instead.

Inorder Successor



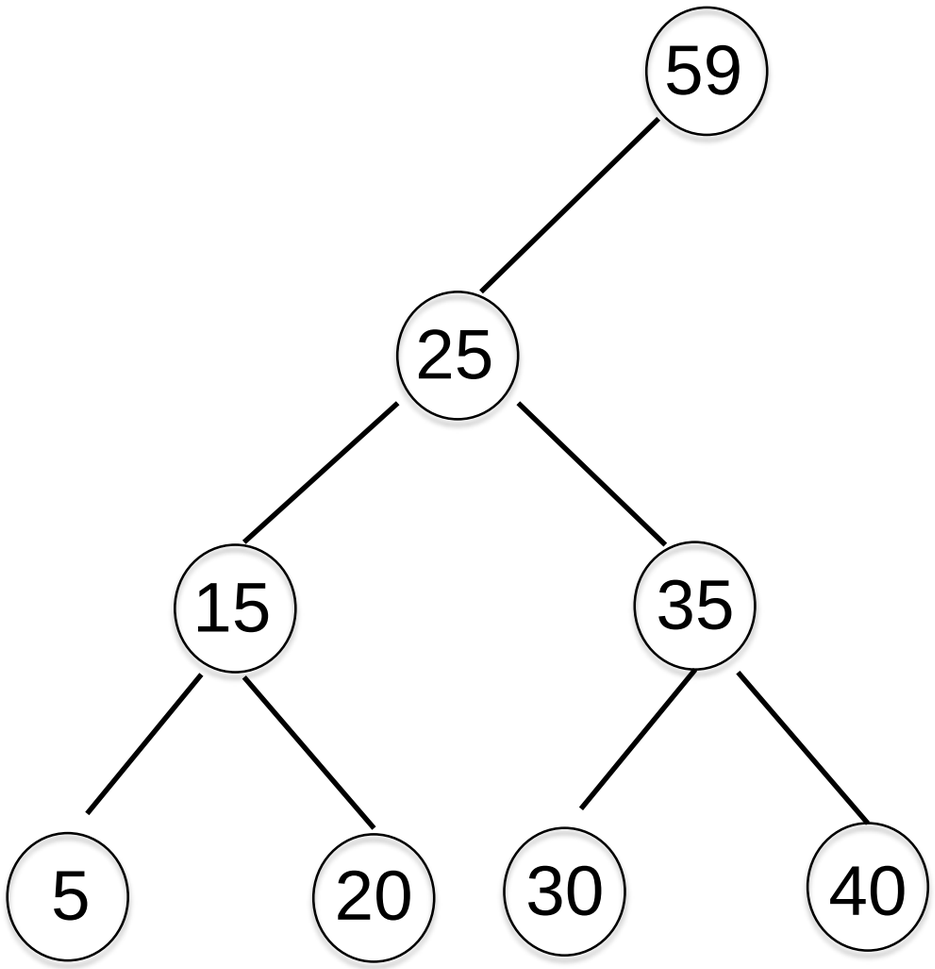
What is the in-order successor of node 25?

Inorder Successor



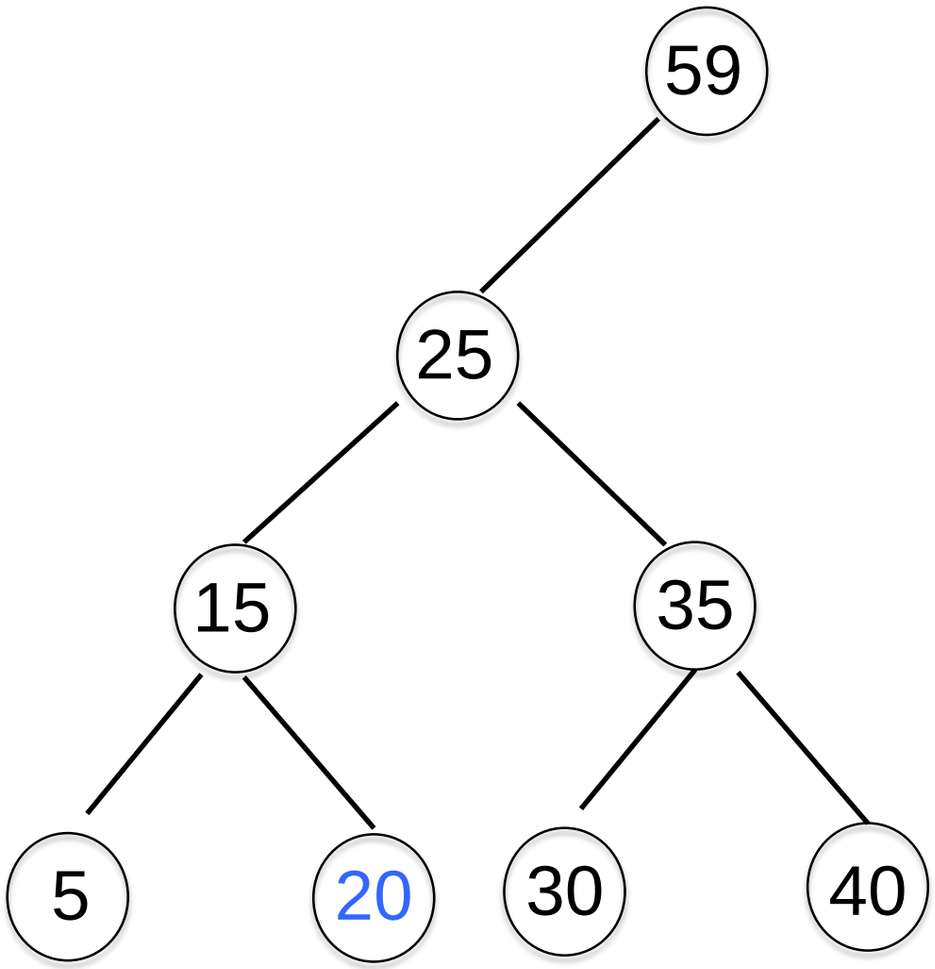
How to find it?

Inorder Predecessor



What is the in-order predecessor of node 25?

Inorder Predecessor



How to find it?

Deletions in a BST

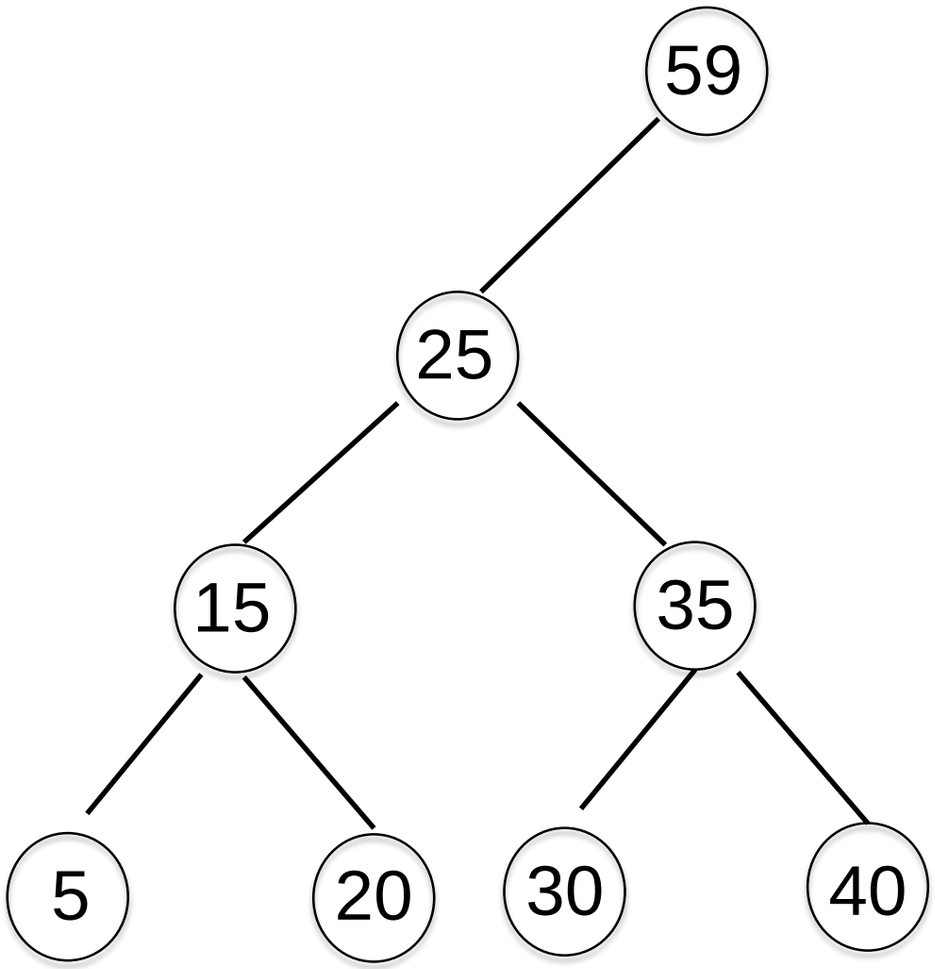
// Thus a trick is needed

Case 4: if the node has a two children

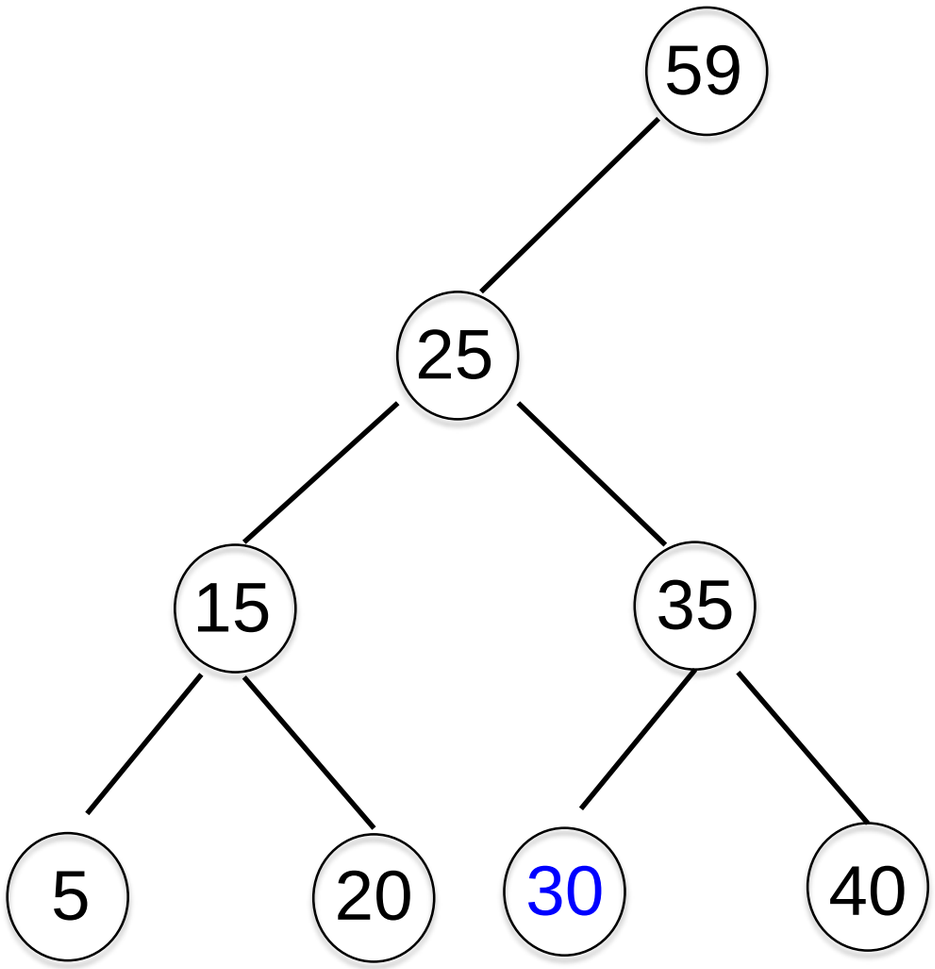
Replace the node with its **predecessor or successor from the inorder traversal** of the tree. and delete that node instead.

Deletions in a BST

Let’s delete 25



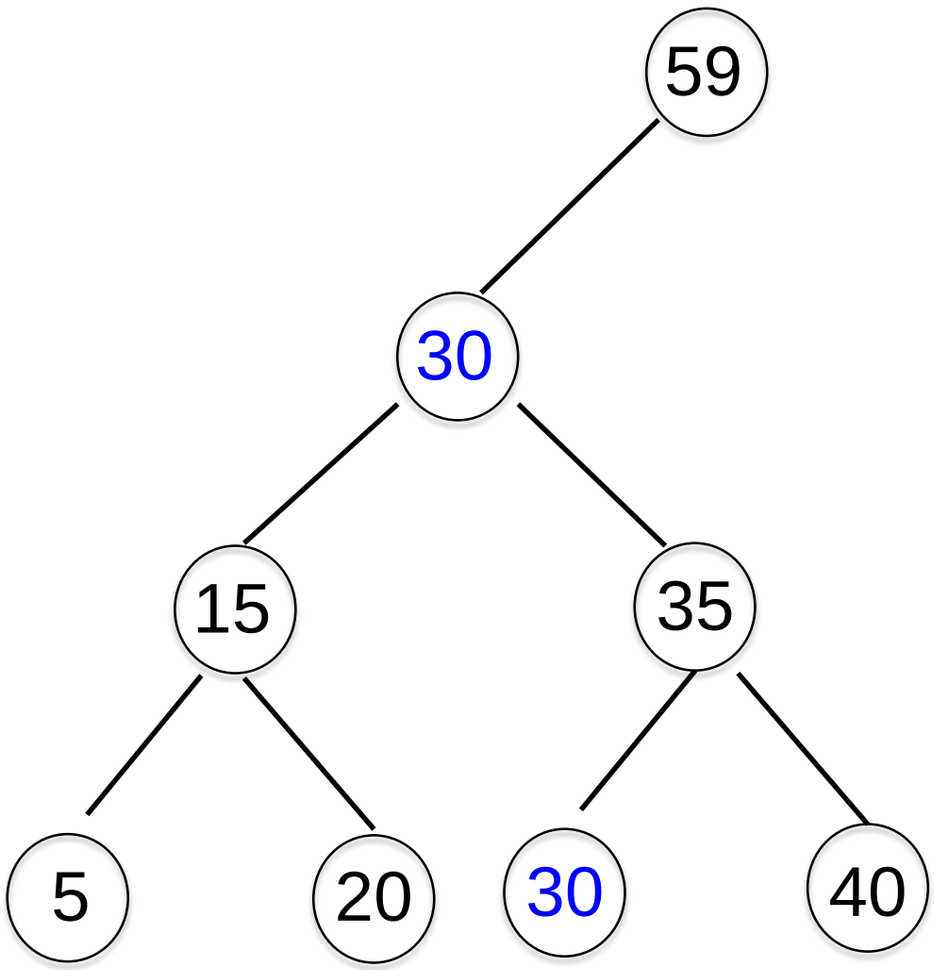
Deletions in a BST



Let’s delete 25

- Identify its successor

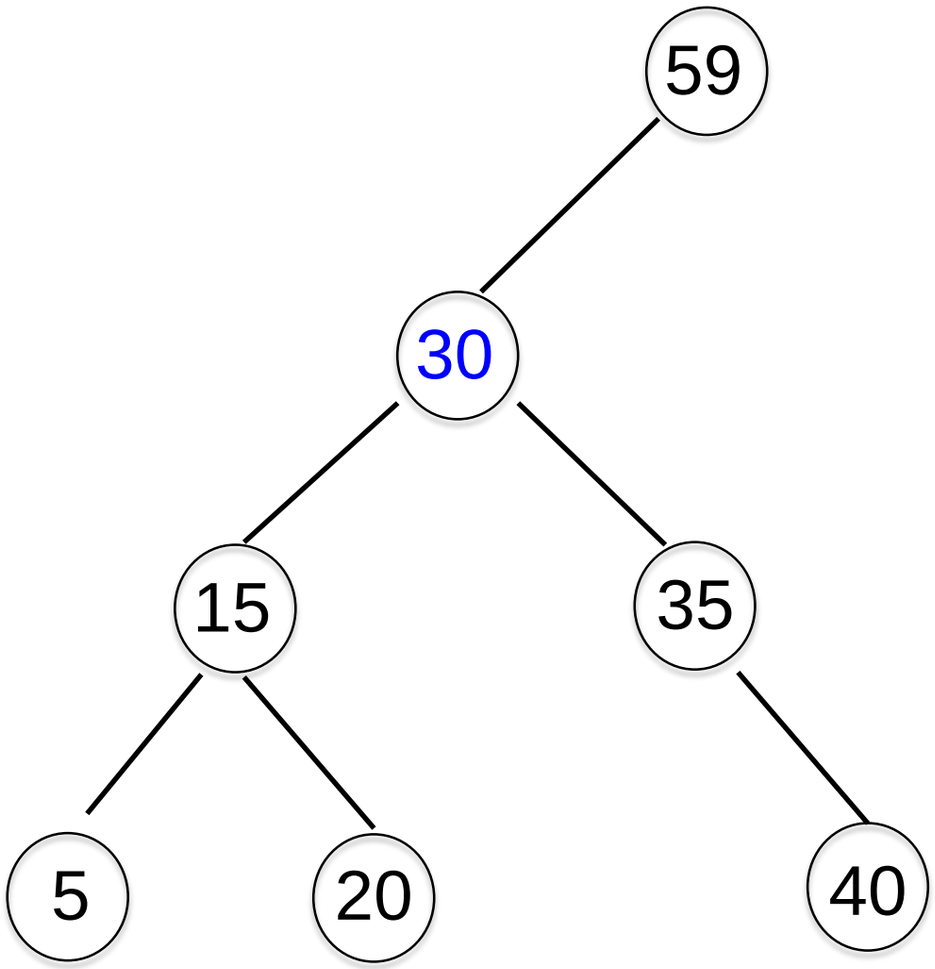
Deletions in a BST



Let’s delete 25

- Replace it with its successor

Deletions in a BST



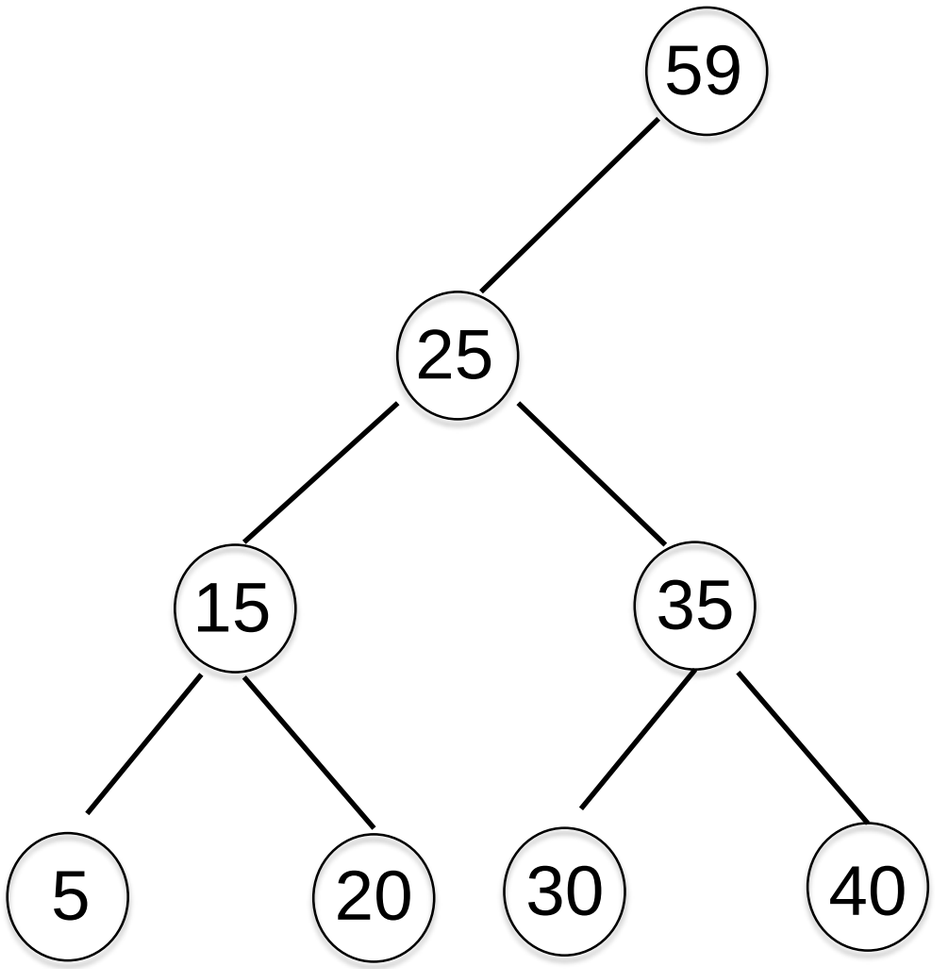
Let’s delete 25

- Delete the successor

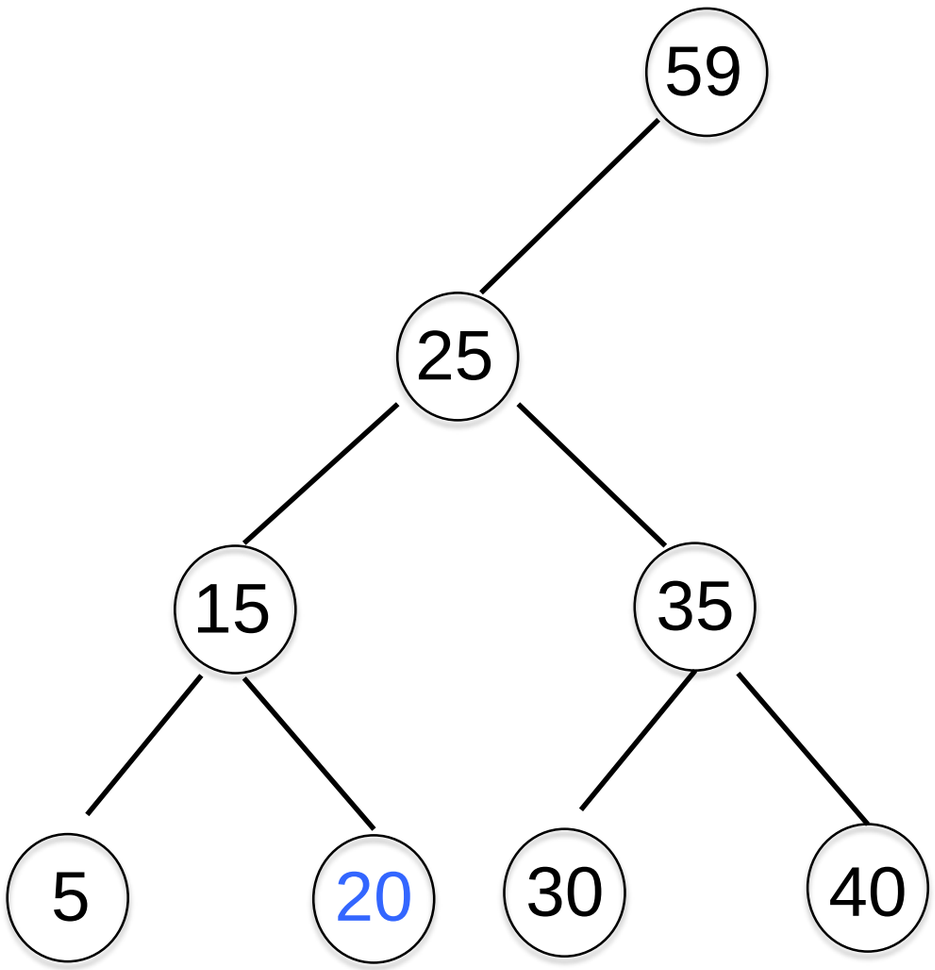
OR

Deletions in a BST

Let’s delete 25



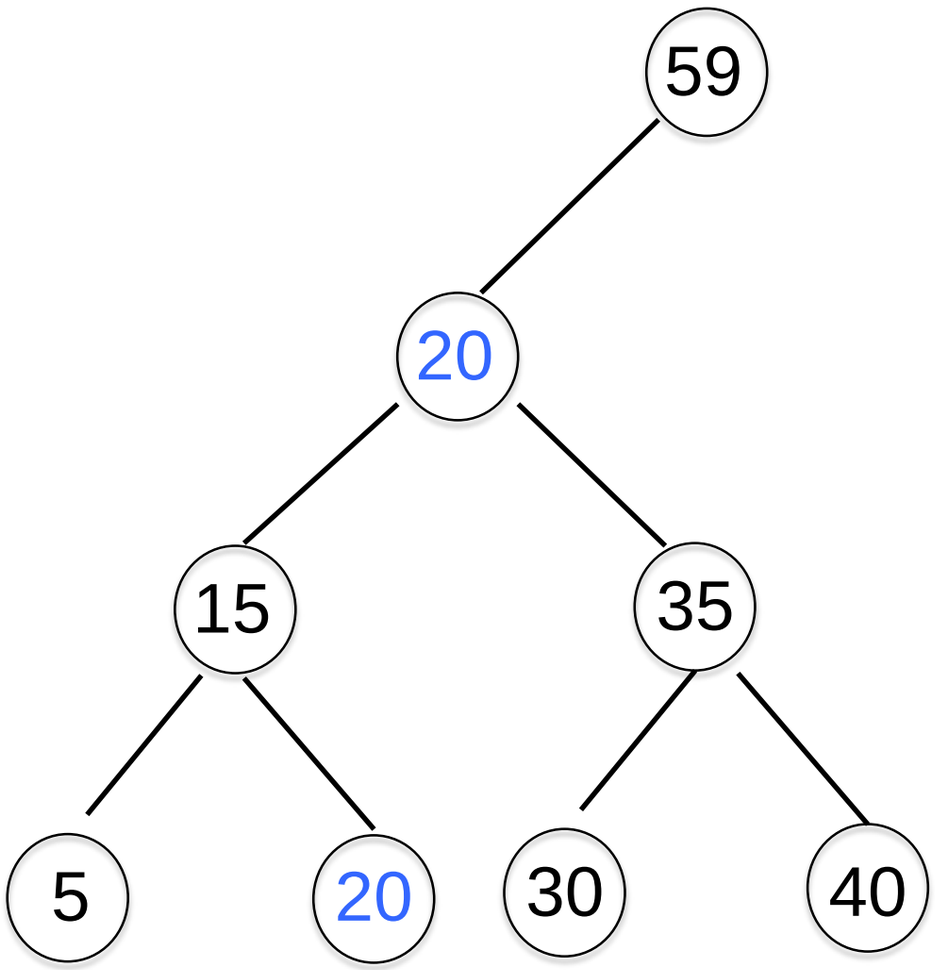
Deletions in a BST



Let’s delete 25

- Identify its predecessor

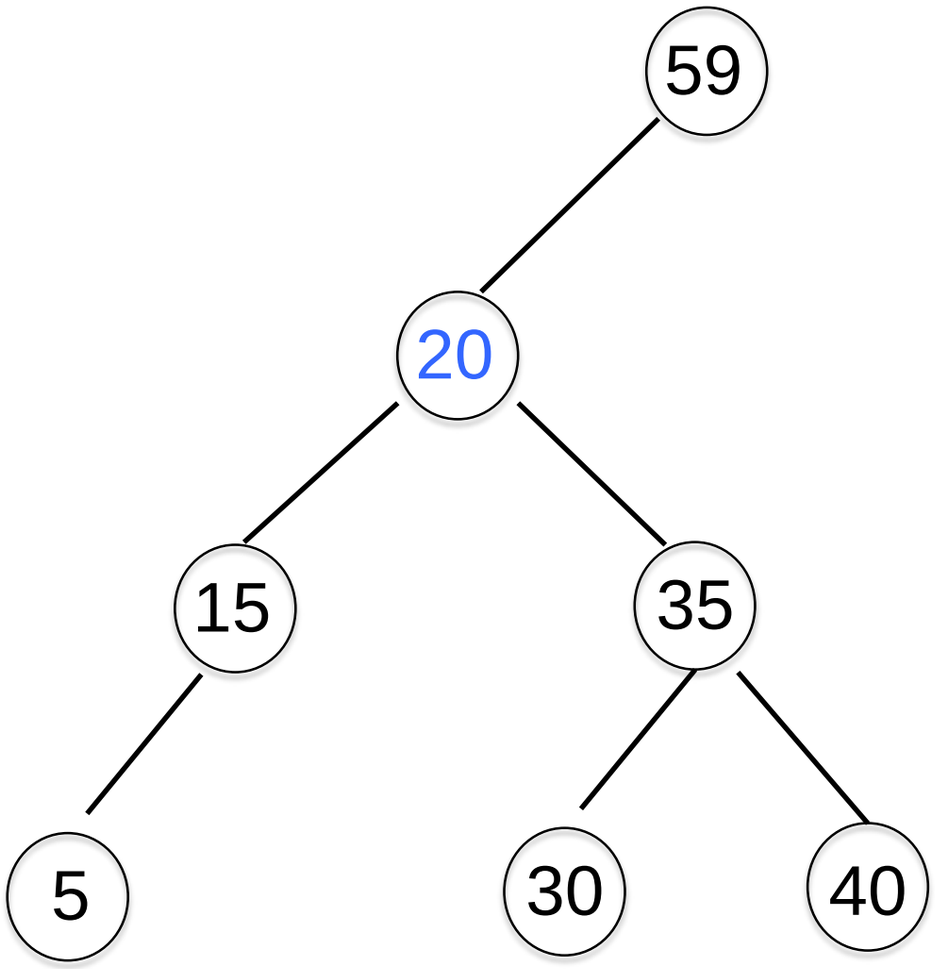
Deletions in a BST



Let’s delete 25

- Replace the node with predecessor

Deletions in a BST



Let’s delete 25

- Delete the predecessor

Deletions in a BST

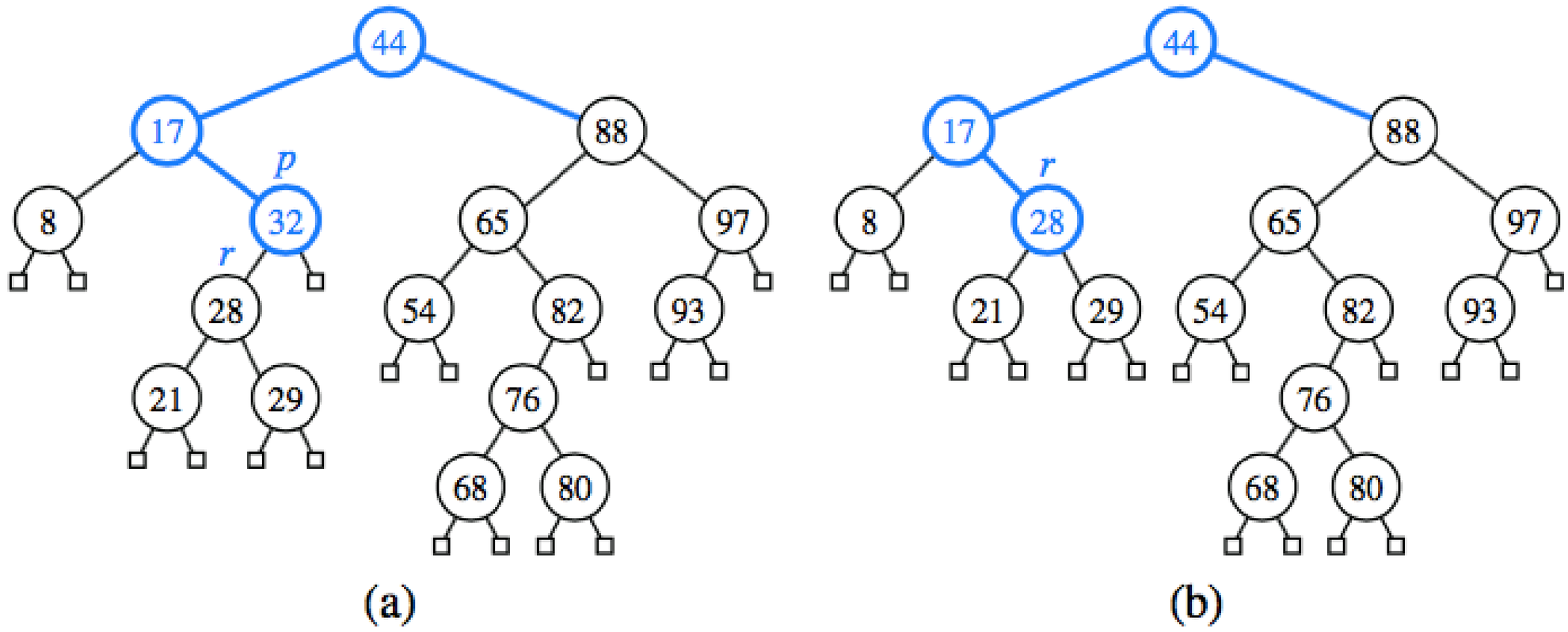


Figure 11.5: Deletion from the binary search tree of Figure 11.4b, where the entry to delete (with key 32) is stored at a position p with one child r: (a) before the deletion; (b) after the deletion.

Deletions in a BST

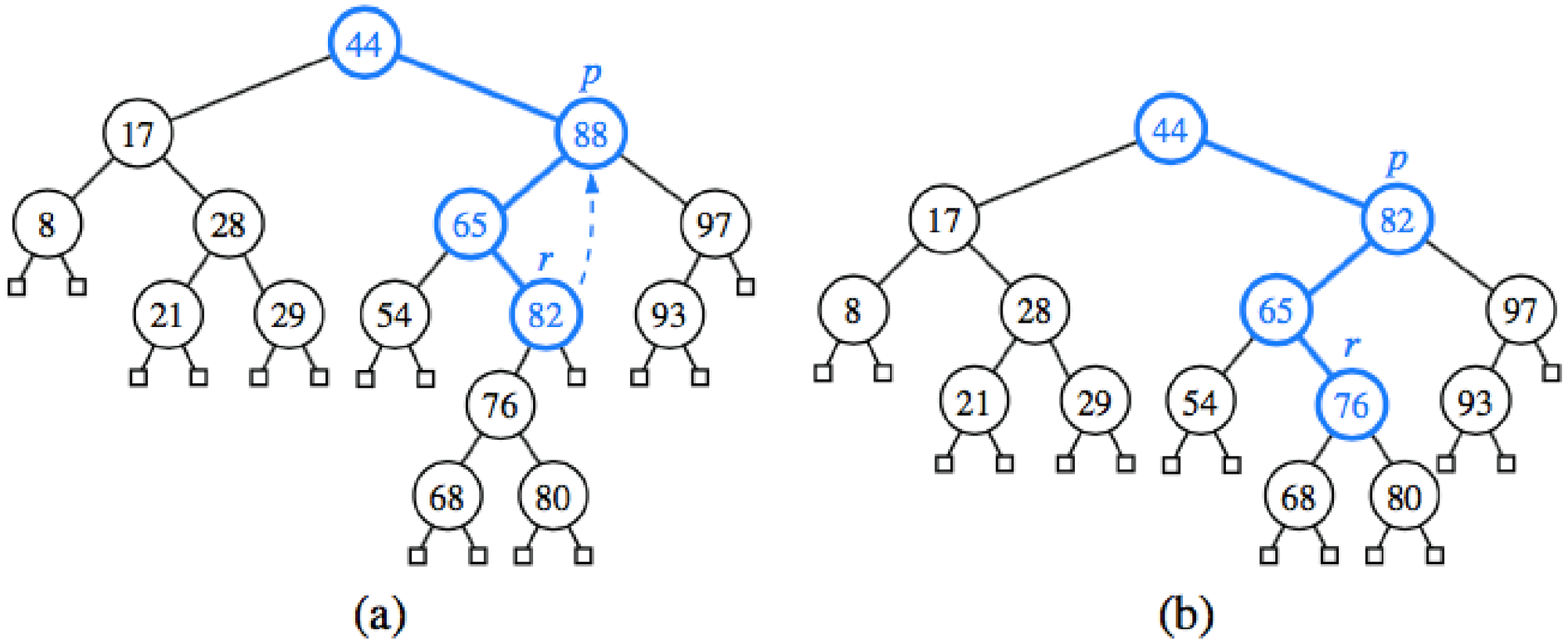


Figure 11.6: Deletion from the binary search tree of Figure 11.5b, where the entry to delete (with key 88) is stored at a position p with two children, and replaced by its predecessor r: (a) before the deletion; (b) after the deletion.

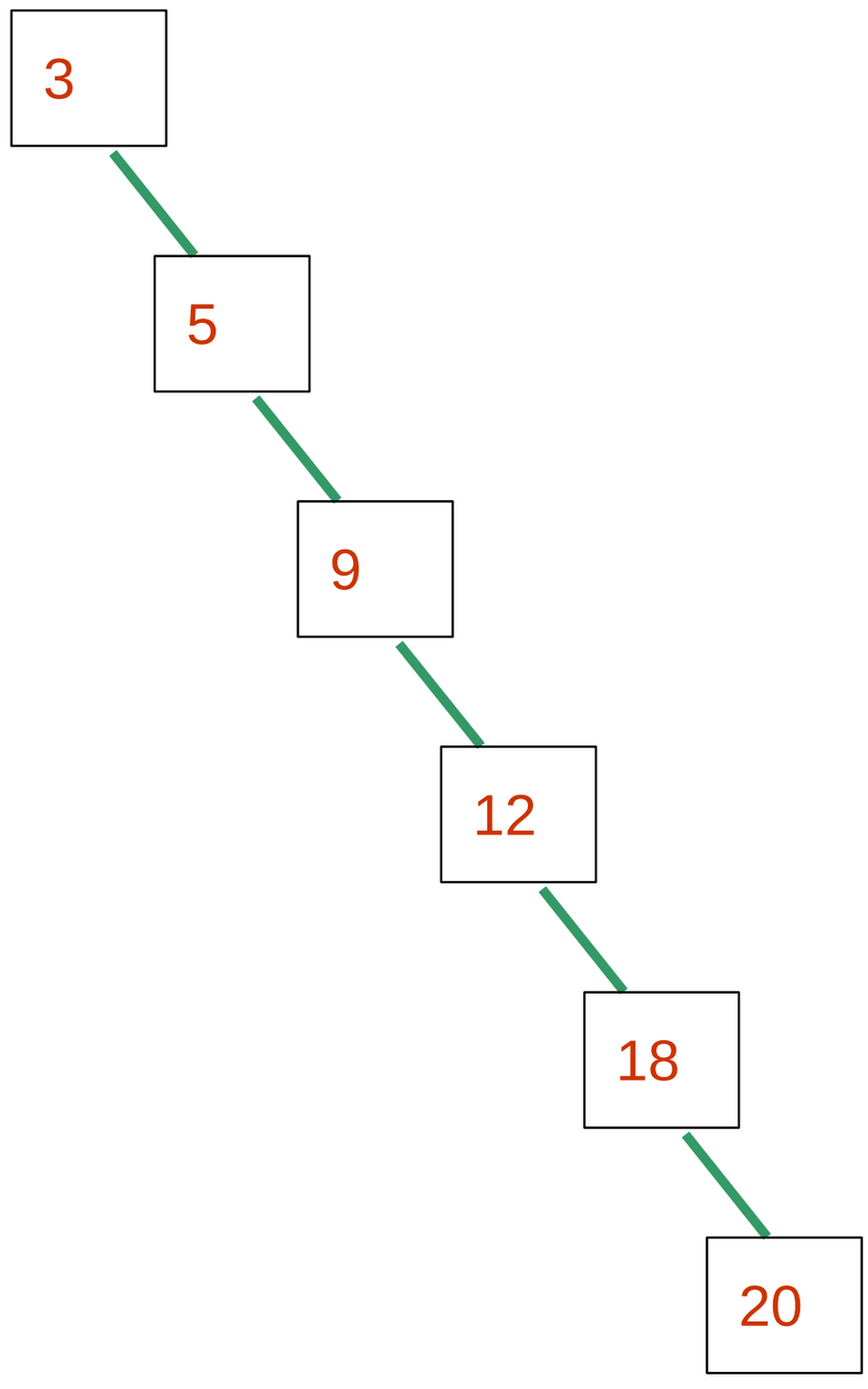
BST Analysis

- 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 $O(\lg n)$

Random BST

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

  - You can see the complete proof in "Introduction to Algorithms," by Cormen, Chapter 12, Theorem 12.4

  - But to make it easy for you to understand the proof, I will explain the PROOF OUTLINE on the board.

Questions as Homework

- How to proof through “Proof via Induction” that the number of external nodes in a binary tree having n internal nodes is $n + 1$?
- Why is the expected height of a randomly built BST of n nodes 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