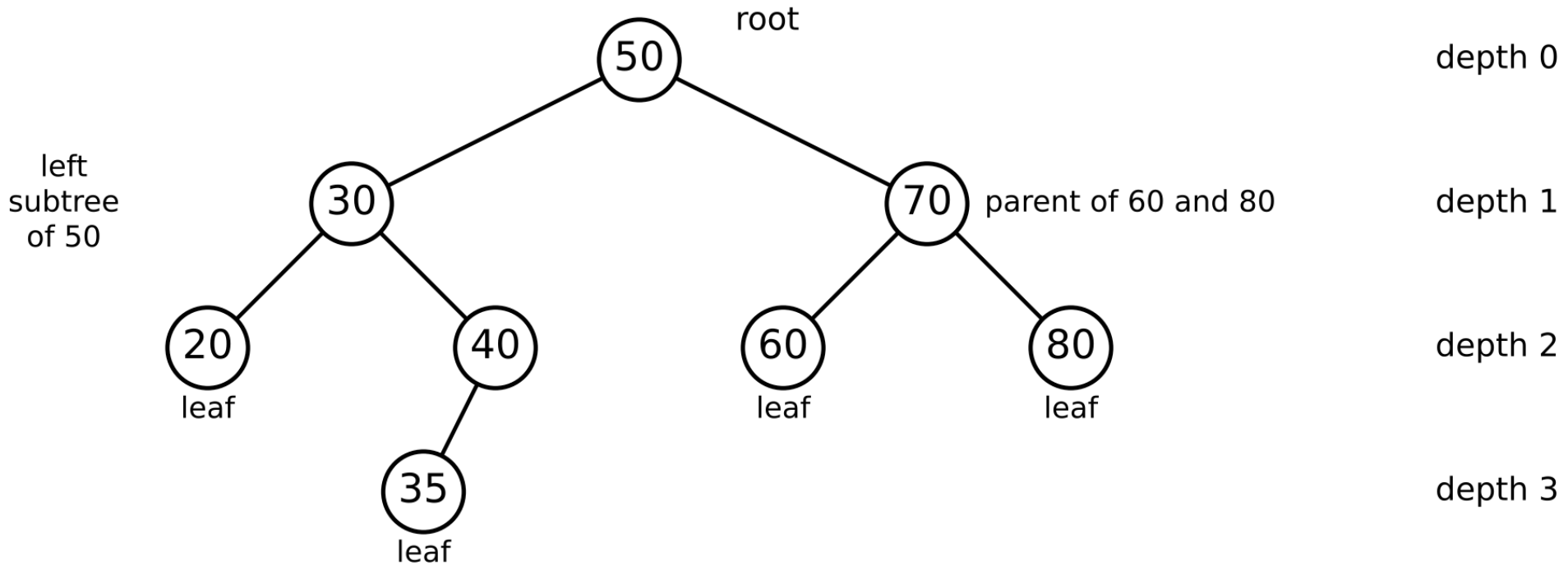


Course Algorithms and Data Structures · Lecture 9

Binary Trees and Binary Search Trees

Textbook: A. Levitin, Sections 1.4, 4.5, 5.3

Trees: the vocabulary



A **binary tree**: every node has at most two children, a left and a right one.

The main terms

Term	Meaning
Root	the top node; it has no parent
Parent, child	a node and the nodes directly below it
Leaf	a node without children
Subtree	a child together with everything below it
Depth of a node	number of edges from the root
Height of a tree	the largest depth; empty tree: −1

How a binary tree is stored

Nodes and links

each node: key, left, right (links to children or null). The tree is reached through its root

In an array

root in cell 0; children of cell i in 2i + 1 and 2i + 2; parent in ⌊(i − 1)/2⌋. Good for trees without gaps

A binary tree is recursive: a root with two smaller binary trees below it.

Three traversals

- **Preorder**: root, left subtree, right subtree
- **Inorder**: left subtree, root, right subtree
- **Postorder**: left subtree, right subtree, root

```
ALGORITHM Inorder(T)
    if T ≠ null
        Inorder(T.left)
        visit T
        Inorder(T.right)
```

Traversals of our tree

Traversal	Order of visits
Preorder	50, 30, 20, 40, 35, 70, 60, 80
Inorder	20, 30, 35, 40, 50, 60, 70, 80
Postorder	20, 35, 40, 30, 60, 80, 70, 50

Inorder gives the keys **sorted**. Every traversal visits each node once: **O(n)**.

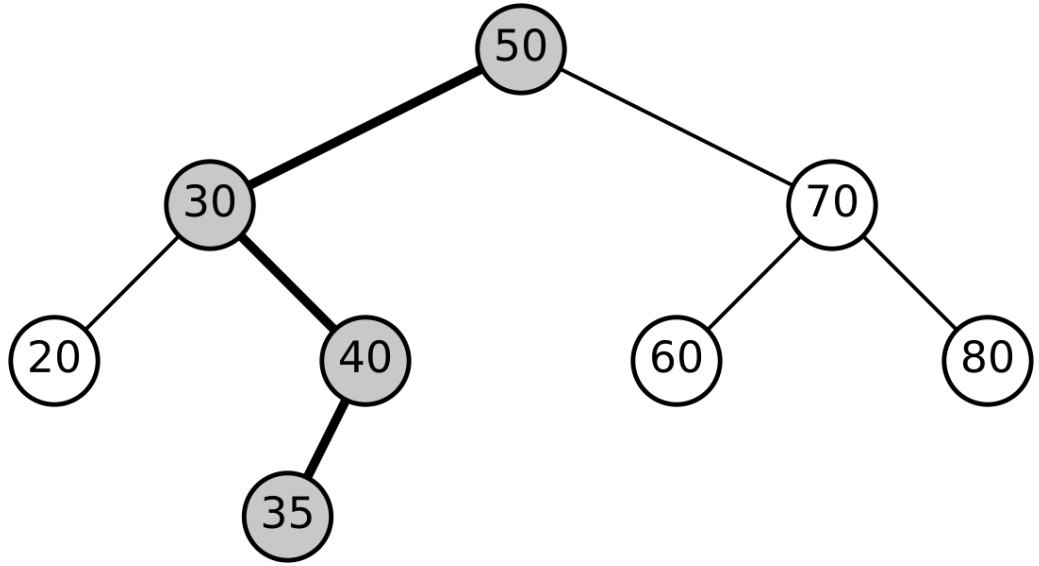
Binary search tree (BST)

> For **every** node: all keys in its **left** subtree are **smaller**, all keys in its **right** subtree are **larger**.

That is why inorder lists the keys in sorted order: smaller keys first, then the node, then larger keys.

Search in a BST

```
ALGORITHM BSTSearch(T, K)
    p ← T
    while p ≠ null and p.key ≠ K do
        if K < p.key
            p ← p.left
        else
            p ← p.right
    return p
```



Search for 35: 50 → 30 → 40 → 35. Search for 65: 50 → 70 → 60 → null, not found.

Insertion

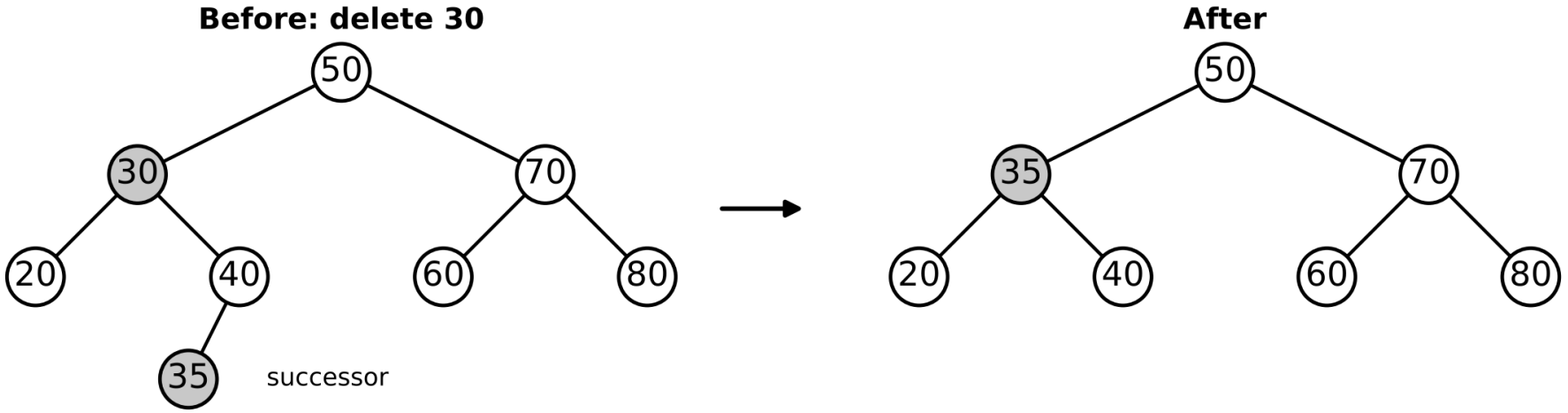
Search for the key; the search ends at a null link, and **that is where the new key goes**. A new key always becomes a leaf.

> Insert 65: follow 50 → 70 → 60 and attach 65 as the right child of 60.

Our tree was built by inserting 50, 30, 70, 20, 40, 60, 80, 35 into an empty tree.

Deletion: three cases

1. **A leaf** (20): just remove it
2. **One child** (40): connect the parent to that child
3. **Two children** (30): replace the key by its **inorder successor**, the smallest key in the right subtree (35), then delete that node



How fast is a BST?

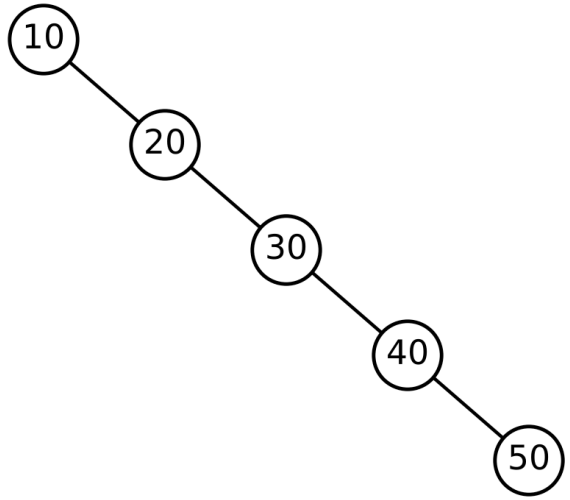
Search, insertion and deletion follow one path from the root: **O(h)**, where h is the height.

Balanced

h ≈ $\log_2$ n: O(log n)

Degenerate

sorted input: h = n − 1, O(n)



BST compared with arrays and lists

Operation	Sorted array	Linked list	BST, balanced	BST, degenerate
Search	O(log n)	O(n)	O(log n)	O(n)
Insert	O(n)	O(1) at front	O(log n)	O(n)
Delete	O(n)	O(n) to find	O(log n)	O(n)
Sorted order	O(n)	needs sorting	O(n), inorder	O(n), inorder

Self-balancing trees (AVL, red-black) guarantee O(log n): later courses, Levitin 6.3.

Summary

- A binary tree is a root with a left and a right subtree, stored with nodes and two links.
- Preorder, inorder and postorder visit all nodes in O(n); inorder of a BST is sorted.
- In a BST smaller keys go left and larger keys go right; operations take O(h).
- $h \approx \log_2 n$ when balanced, but $n - 1$ for sorted input.

Review questions

1. What is a root, a leaf and the height of a tree? Find them in the tree from the lecture.
2. Write the preorder, inorder and postorder traversals of the BST built by inserting 8, 3, 10, 1, 6.
3. What happens to the shape of a BST if the keys are inserted in increasing order? Why is this bad?

Reading: Levitin, Sections 1.4, 4.5, 5.3; Cormen et al., Chapter 12.