Chapter 12. Binary Search Tree

Joon Soo Yoo

May 8, 2025

Assignment

- Read §12.1, 12.2
- Problems
  - §12.1 - 1, 3
  - §12.2 - 1, 3, 5

Chapter 12: Binary Search Trees

- **Chapter 12.1: What is a Binary Search Tree?**
- Chapter 12.2: Querying a Binary Search Tree
- Chapter 12.3: Insertion and Deletion

Binary Search Trees: Overview

- **BSTs support fundamental dynamic-set operations**:
  - SEARCH, MINIMUM, MAXIMUM, PREDECESSOR, SUCCESSOR, INSERT, DELETE
- **Operation running time depends on tree height**:
  - $\Theta(\lg n)$ in the best case (balanced BST)
  - $\Theta(n)$ in the worst case (degenerate / linear BST)

BST as Dictionary and Priority Queue

- **BST as a Dictionary:**
  - Supports dynamic set operations:
    - SEARCH
    - INSERT
    - DELETE
  - Store and retrieve key-value pairs efficiently (when balanced).
- **BST as a Priority Queue:**
  - Supports priority queue operations:
    - MINIMUM / MAXIMUM (find smallest or largest key)
    - DELETE-MIN / DELETE-MAX (delete smallest or largest key)
  - Can maintain ordered keys dynamically.

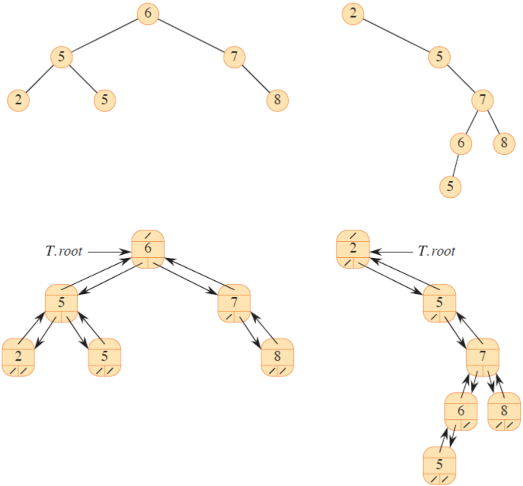
Binary Search Tree: Structure

- **A Binary Search Tree (BST)** is organized as a binary tree.
- Each node contains:
  - **Key and satellite data**
  - **Pointers:** left child, right child, and parent (`left`, `right`, `p`)
  - Missing child or parent $\rightarrow$ stored as `NIL`
- **Root pointer (`T.root`)**:
  - Points to the root of the tree
  - `T.root.p` is always `NIL`

Binary-Search-Tree Property

- **Ordering rule:**
  - For any node x:
    - All keys in the left subtree of $x \leq x$.key
    - All keys in the right subtree of $x \geq x$.key
- **Implication:**
  - BST naturally supports sorted order traversal.
  - Different BSTs can represent the same set of keys $\rightarrow$ shape and height may vary.

BST Diagram



(a) (b)

BST Supports Sorted Order Traversal

- **Inorder Traversal:**
  - Visits nodes in order: **Left → Node → Right**
  - For BSTs, this produces keys in ascending (sorted) order.
- **Example:**
  - BST structure:
    - Left subtree → smaller keys
    - Right subtree → larger keys
  - Inorder traversal naturally visits:

$$\text{smallest} \rightarrow \ldots \rightarrow \text{largest}$$

- **Thus, BST + Inorder traversal = Sorted order output.**

Inorder Tree Walk (Algorithm)

```
INORDER-TREE-WALK(x)
if x ≠ NIL
        INORDER-TREE-WALK(x.left)
        print x.key
        INORDER-TREE-WALK(x.right)
```

- **Example:** Inorder traversal of Figure 12.1 prints keys in order:

$$2,\ 5,\ 5,\ 6,\ 7,\ 8$$

Inorder Tree Walk takes $\Theta(n)$ Time

- **Theorem 12.1**

  If x is the root of an n-node subtree, then the call:

  $$\text{INORDER-TREE-WALK}(x)$$

  takes $\Theta(n)$ time.

- **Intuition:**
  - Every node is visited exactly once $\rightarrow$ at least $\Omega(n)$
  - Each visit takes constant work $\rightarrow$ total time is $O(n)$

Proof of Theorem (Part 1)

- **Base Case:**
  - Empty subtree $\rightarrow n = 0$:

    $$T(0) = c$$

    for some constant $c > 0$.

- **Recursive Case ($n > 0$):**
  - Left subtree $\rightarrow k$ nodes $\rightarrow T(k)$
  - Right subtree $\rightarrow n - k - 1$ nodes $\rightarrow T(n - k - 1)$
  - Current node $\rightarrow$ constant work d

$$T(n) \leq T(k) + T(n - k - 1) + d$$

- **Goal:** Show $T(n) = O(n)$

Proof of Theorem (Part 2)

- **Guess (substitution method):**

$$T(m) \le (c+d)\cdot m + c$$

- **Substitute into recurrence:**

$$T(n) \le (c+d)k + c + (c+d)(n-k-1) + c + d$$

- **Simplify:**

$$= (c+d)n + c - (c+d) + c + d$$

$$= (c+d)n + c$$

$\rightarrow$ Matches the guess $\rightarrow$ proof complete.

- **Final conclusion:**

$$T(n) = \Theta(n)$$

Chapter 12: Binary Search Trees

- Chapter 12.1: What is a Binary Search Tree?
- **Chapter 12.2: Querying a Binary Search Tree**
- Chapter 12.3: Insertion and Deletion

Queries Supported by Binary Search Trees

- Binary Search Trees can support:
  - **SEARCH**
  - **MINIMUM**
  - **MAXIMUM**
  - **PREDECESSOR**
  - **SUCCESSOR**

- **Running time:**
  - Each operation takes $O(h)$ time, where h is the height of the tree.

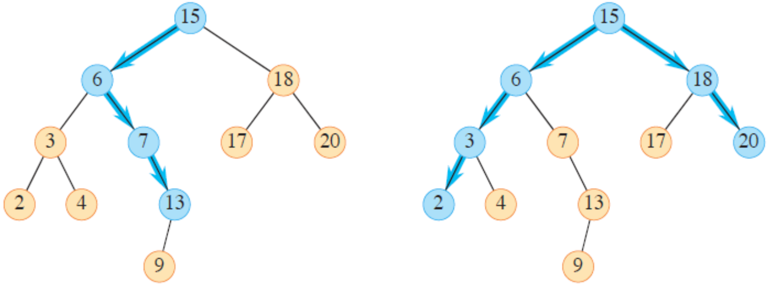
Searching in BST (Alg)

```
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)

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
```

- $k < x.key \rightarrow$ Go left
- $k > x.key \rightarrow$ Go right

Searching in BST (Diagram)



(a) (b)

Finding Minimum and Maximum in BST

Minimum and Maximum Queries:

- Follow child pointers from the root:
  - **Minimum:** follow left child pointers.
  - **Maximum:** follow right child pointers.

Finding Minimum/Maximum in BST

```
TREE-MINIMUM(x)
while x.left ≠ NIL
    x = x.left
return x
```

```
TREE-MAXIMUM(x)
while x.right ≠ NIL
    x = x.right
return x
```

Why TREE-MINIMUM is Correct

- **Binary-Search-Tree Property:**
  - Left subtree keys $\leq$ Current node
  - Right subtree keys $\geq$ Current node
- **Two Cases:**
  - No left child $\rightarrow$ current node is minimum.
  - Has left child $\rightarrow$ smaller key exists $\rightarrow$ minimum is in left subtree $\rightarrow$ follow left.
- **TREE-MAXIMUM is symmetric $\rightarrow$ follow right for maximum.**

Running Time of Minimum and Maximum

- **Follow one simple path downward:**
  - Minimum $\rightarrow$ only follow left children.
  - Maximum $\rightarrow$ only follow right children.
- **Running time is proportional to the height of the tree:**

$$O(h) \text{ time}$$

- **Tree height:**
  - Balanced BST $\rightarrow O(\log n)$
  - Worst-case unbalanced BST $\rightarrow O(n)$

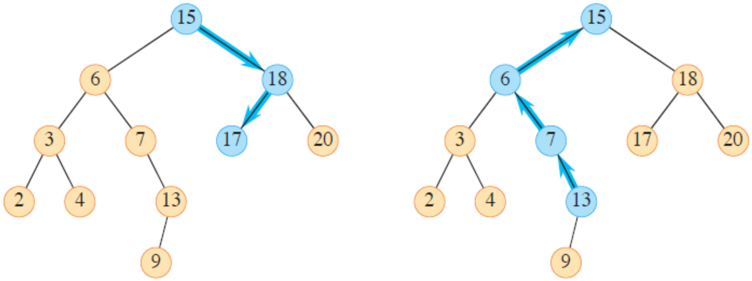
Importance of Balancing in BST

- **BST operation time depends on tree height h:**

$$O(h)$$

- **Balanced vs Unbalanced BST:**
  - **Balanced tree:** $h = O(\log n) \rightarrow$ Fast operations.
  - **Unbalanced tree (degenerate):** $h = O(n) \rightarrow$ Slow operations.
- **Solution: Self-balancing trees (next chapters).**
  - Red-Black Trees (Chapter 13)
  - AVL Trees, B-Trees (other types)
- **Summary:**
  - **Balancing is crucial** $\rightarrow$ ensures $O(\log n)$ height $\rightarrow$ efficient search, insert, and delete.

Finding Successor in BST (Diagram)



(c) (d)

Finding Successor in BST (Algorithm)

```
TREE-SUCCESSOR(x)
if x.right ≠ NIL
    return TREE-MINIMUM(x.right)   // leftmost node in right subtree
else // find the lowest ancestor of x whose left child is an ancestor of x
    y = x.p
    while y ≠ NIL and x == y.right
        x = y
        y = y.p
    return y
```

Finding Successor in BST (Case 1)

- **Definition:** Successor $\rightarrow$ next node in inorder traversal (smallest key greater than current node).
- **Case 1: If right subtree exists**
  - Successor is the **minimum** node in the right subtree.
  - Use TREE-MINIMUM to find it.

Example:

- Successor of 15 $\rightarrow$ right subtree $\rightarrow$ smallest $\rightarrow$ 17.

Finding Successor in BST (Case 2)

- **Case 2: No right subtree**
  - Go up the tree until coming from a left child.
  - The parent at this point is the successor.
- **Example:**
  - Successor of 13 $\rightarrow$ no right subtree $\rightarrow$ go up $\rightarrow$ first from left $\rightarrow$ 15.
- **Running time:**

$$O(h) \text{ (path down or up)}$$

Summary of BST Query Operations

- **Binary Search Tree supports:**
  - **SEARCH**
  - **MINIMUM**
  - **MAXIMUM**
  - **SUCCESSOR**
  - **PREDECESSOR**

Theorem 12.2

The dynamic-set operations SEARCH, MINIMUM, MAXIMUM, SUCCESSOR, and PREDECESSOR can be implemented so that each one runs in $O(h)$ time on a binary search tree of height h.

Question?