Data Structure - Spring 2022
15. Search Tree (Lab)

Walid Abdullah Al
Computer and Electronic Systems Engineering
Hankuk University of Foreign Studies

TA: **Seong Joo Kim**

Based on:
Goodrich, Chapter 9,11
Karumanchi, Chapter 6-7
Slides by Prof. Yung Yi, KAIST
Slides by Prof. Chansu Shin, HUFS

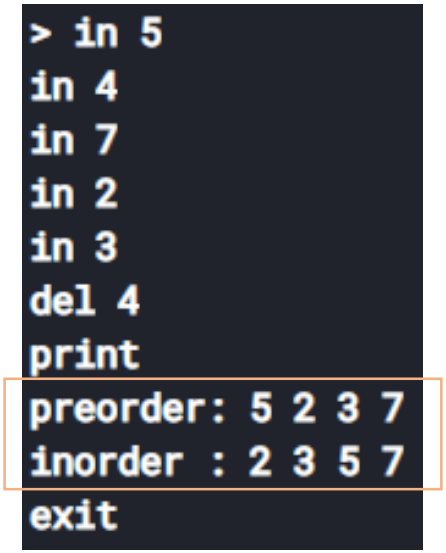
Computer Vision Lab
Hankuk University of Foreign Studies

Today's Task

- **Complete BinarySearchTree class**
  - search: (already coded)
  - **search_p**
  - insert: (already coded)
  - preorder, postorder, inorder : (already coded)
  - **find_min_p:** (already coded)
  - **replace**
  - delete: (already coded)

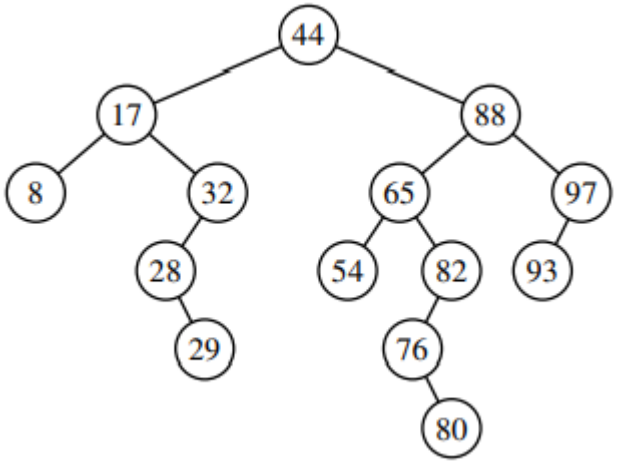
Command Interface

- **Commands:**
  - **in 5:** inserts key:5 into the tree
  - **del 5:** deletes key:5 from the tree
  - **find 5:** searches for key:5
  - **print:** prints the keys in preorder and inorder traversal
  - **exit:** exits the program



Binary Search Tree (BST)

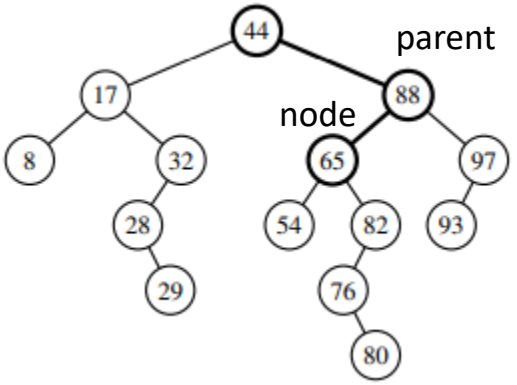
- a binary tree with each node storing a key-value pair (***k***, v) such that:
  - Keys of its left subtree are < ***k***
  - Keys of its right subtree are > ***k***



A binary search tree with integer keys.

search_p: Search with parent

- **search_p(root, k):** finds the **node** with key **k** and its **parent**
- Similar process but use **loop** instead of recursion
- **Initialize:**
  - node=parent=root
- **Loop:**
  - **If** node is None: **Break**
  - **If** k==node.key: **Break**
  - parent=node
  - **If** k<node.key **and** left subtree exists: node=node.left
  - **If** k>node.key **and** right subtree exists: node=node.right
- **Return** node, parent



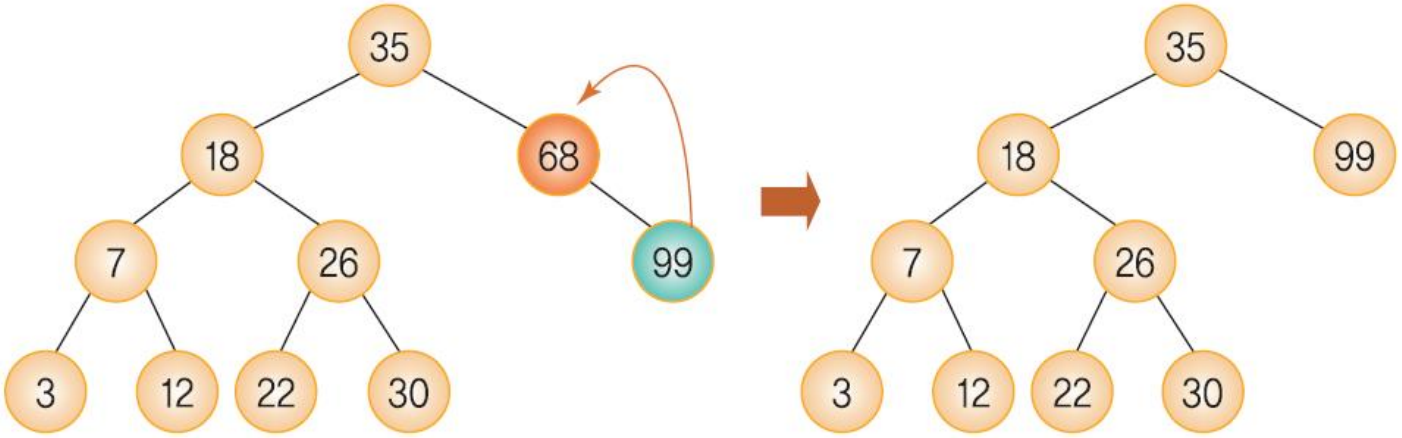
Search(root, 65)

BST: Deletion

- **delete(root, k):** removes the node storing key **k**
- Search the **node** and its parent **p** using **search_p**
- Deleting **node** is not as simple as insertion
  - Insertion: always done as leaf nodes
  - Deletion: can be for any nodes
- Two scenario:
  - **node** has atmost ($\leq$) one child
  - **node** has two children

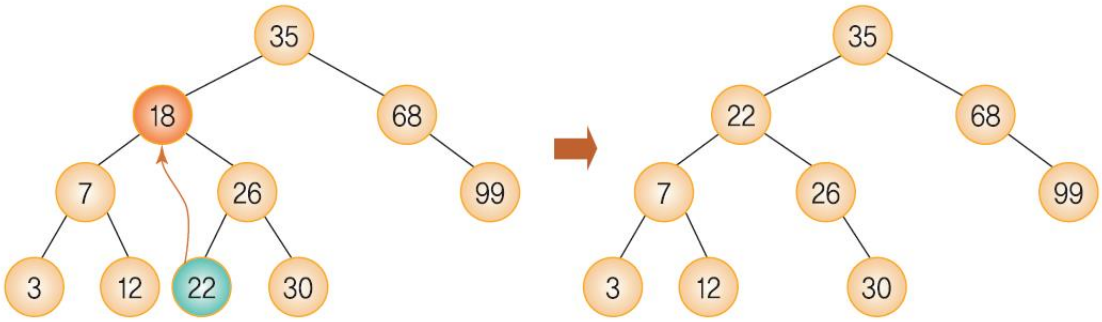
BST: Deletion

- *Case-I:* **node** has atmost (≤) one child
  - Delete **node** and link its child to the **parent** (i.e, bring the child in **node**'s position)
- Note: this case also generalizes the no-child case



BST: Deletion

- *Case-II:* **node** has both left and right child
  - To delete and replace **node,**
  - bring the *largest-key-node* from the *left-subtree*
  - **Or,** the *smallest-key-node* from the *right-subtree*
  - (FOR THIS LAB, DO THE **LATTER**)

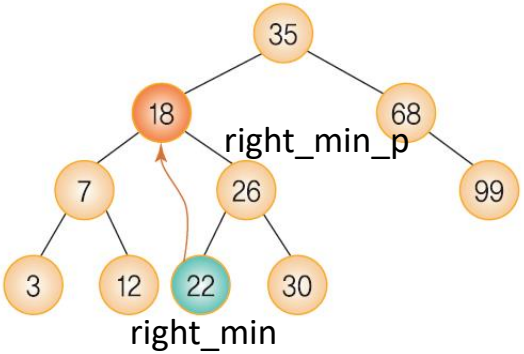


How to find **right_min?**

- **right_min:** minimum-key-node of the right subtree
- **right_min_p:** <right_min>'s parent

```
def find_min_p(root, p):
        while root.left is not None:
            p = root
            root = root.left
        return root, p

right_min, right_min_p = find_min_p(node.right, node)
```

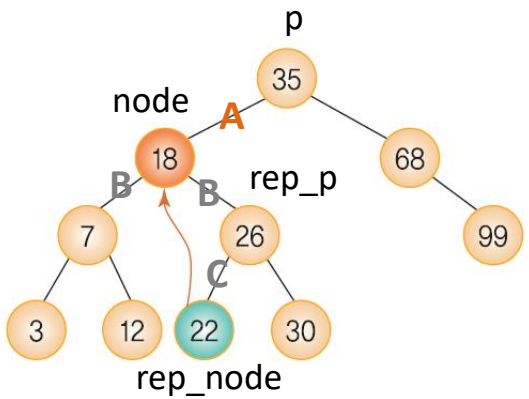


How to **replace** node during deletion?

- **replace(node, rep_node, p, rep_p):**

```
replace <node> with <rep_node> """
# p: parent of <node>
# rep_p: parent of <rep_node>
```

```
A. REPLACE p-to-node link
case-1: node is root (p is None)
case-2: node is p.left
case-3: node is p.right
```



replace (contd.)

B. REPLACE node-to-child(s) link(s) IF <rep_node> exists

case-1: <rep_node> is <node>'s left child --> Do nothing

case-2: <rep_node> is <node>'s right child

--> replace the left link IF node.left exists

case-3: <rep_node> is not <node>'s child

--> replace both left and right links

C. DELETE <rep_p> to <rep_node> link