Data Structure - Spring 2022

14. Heap & Search Tree

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

Priority Queue

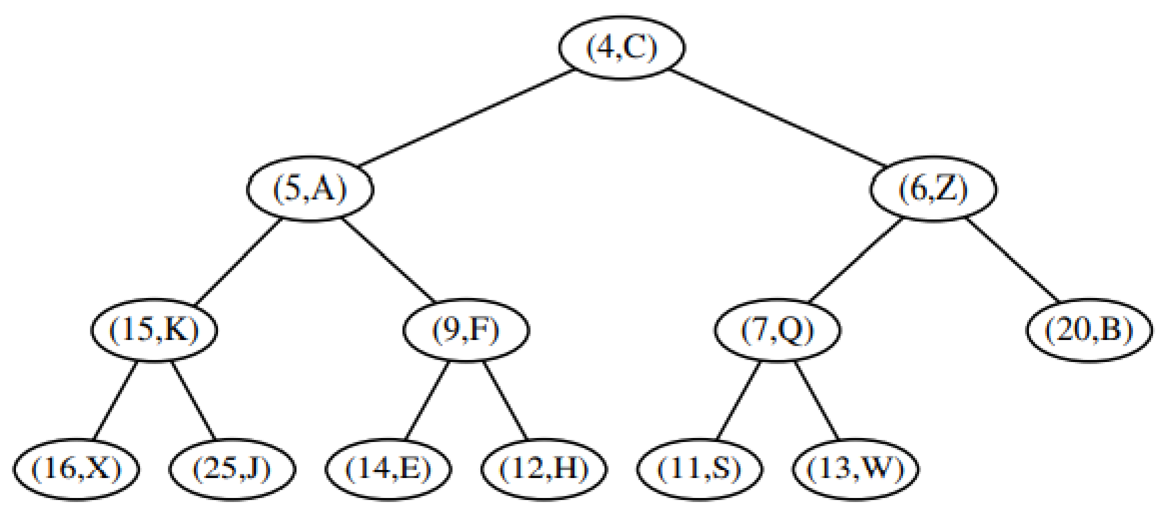
- Queue but not first-in-first-out (FIFO)
- Rather:
  - Arbitrary insertion
  - Priority-based removal
- Stores item as key-value pair (k, v)
  - Key, **k**: priority of the item
  - Example: {(5,A), (7,D), (9,C)}
- Dequeue ≡ Remove_min:
  - Remove the item with the minimum key

Priority Queue: Operation Example

Operation	Return Value	Priority Queue
P.add(5,A)		{(5,A)}
P.add(9,C)		{(5,A), (9,C)}
P.add(3,B)		{(3,B), (5,A), (9,C)}
P.add(7,D)		{(3,B), (5,A), (7,D), (9,C)}
P.min()	(3,B)	{(3,B), (5,A), (7,D), (9,C)}
P.remove_min()	(3,B)	{(5,A), (7,D), (9,C)}
P.remove_min()	(5,A)	{(7,D), (9,C)}
len(P)	2	{(7,D), (9,C)}
P.remove_min()	(7,D)	{(9,C)}
P.remove_min()	(9,C)	{ }
P.is_empty()	True	{ }
P.remove_min()	"error"	{ }

Binary Heap

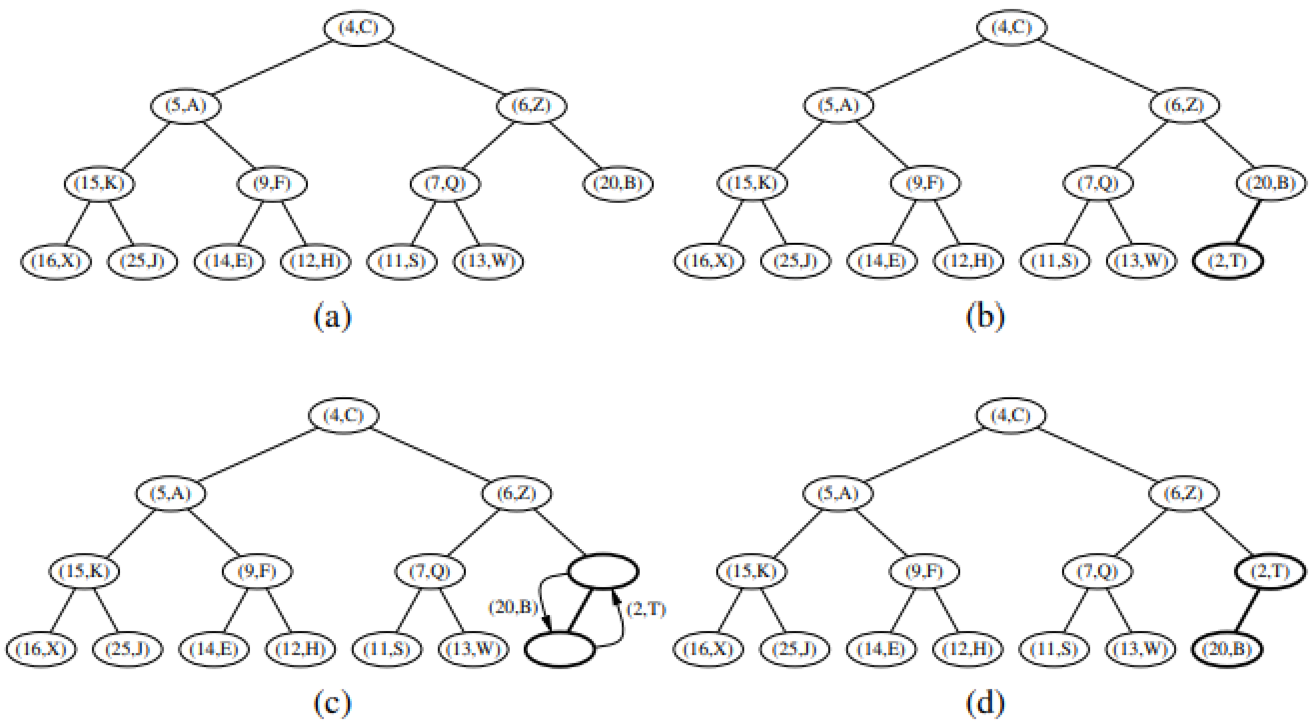
- An efficient structure to implement priority queue
- Properties:
  - **Complete binary tree** property
  - **Heap-order** property: key of any node >= key of its parent



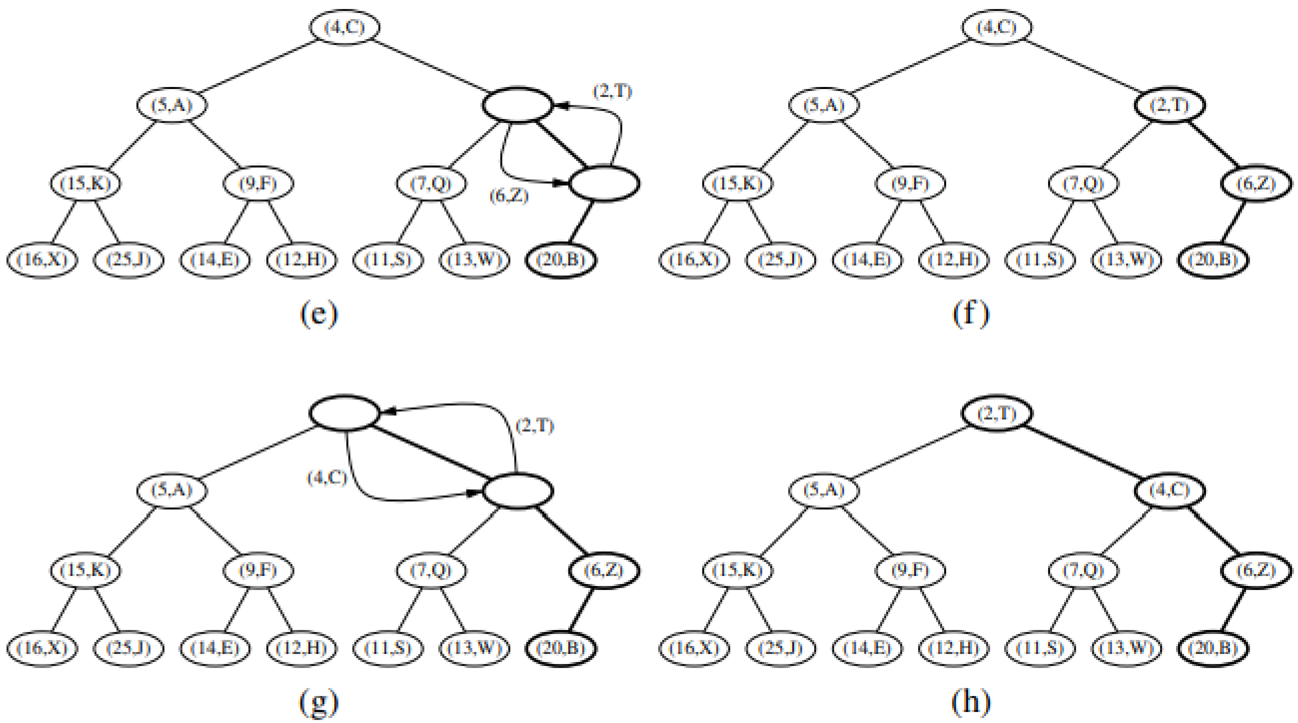
Heap: Insertion

- **add(k, v):** insert key-value pair (***k***, ***v***)
- Insert at the rightmost position ***q*** of the last level
  - to maintain the complete binary tree property
- Perform **up-heap bubbling**
  - To maintain the heap-order property
  - Continue to swap with the parent node until ***k*** >= key(parent)
- Worst-case running time: O(logn)

Heap: Insertion



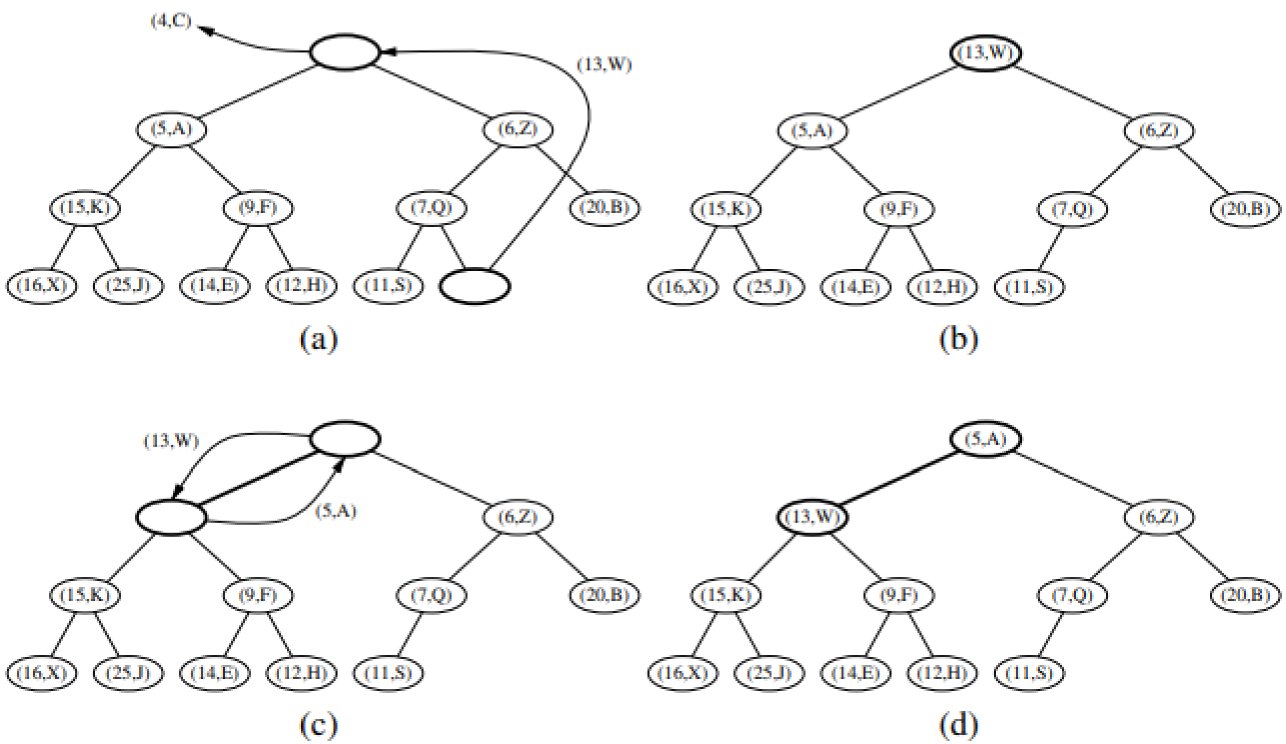
Heap: Insertion



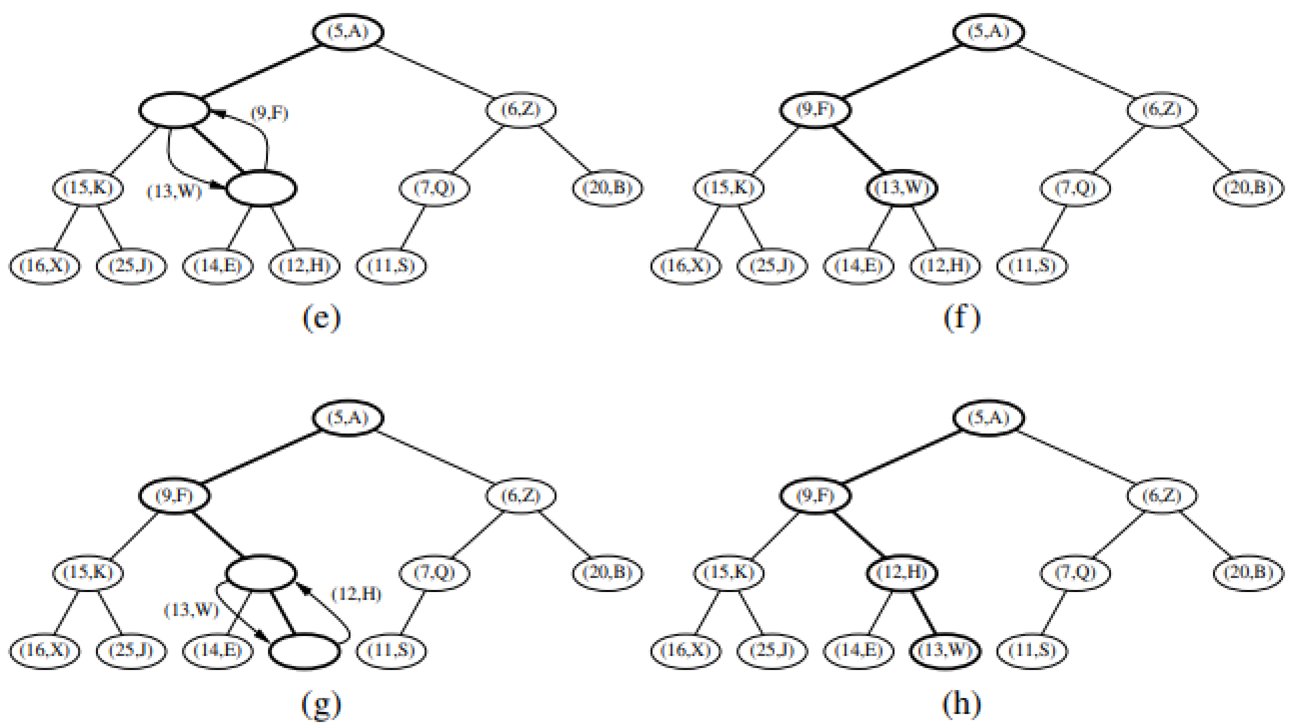
Heap: min-key deletetion

- **remove_min():** removes the root node
- To maintain the complete binary tree property:
  - Remove the node at the last position (rightmost position of the last level)
    and copy it to the root (replacing the original root-item)
  - Say: the new root is: (***k***, v)
- Perform **down-heap bubbling**
  - to maintain the heap-order property
  - Continue to swap with the minimal-key-child
    until ***k*** <= key(minimal-key-child)
- Worst-case running-time: O(logn)

Heap: remove_min

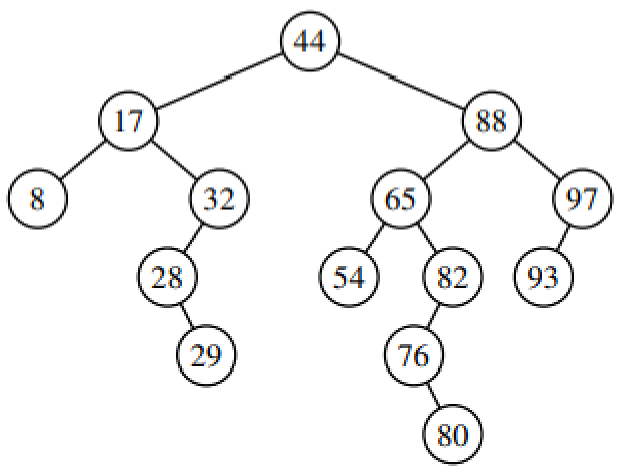


Heap: remove_min



Binary Search Tree (BST)

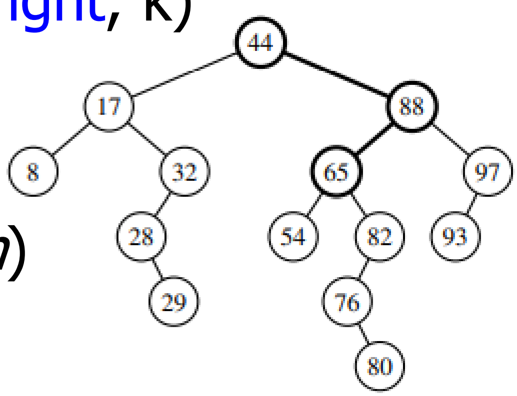
- a binary tree with each node storing a key-value pair (***k***, v) such that:
  - Keys of its left subtree are < ***k***
  - Keys of its right subtree are > ***k***



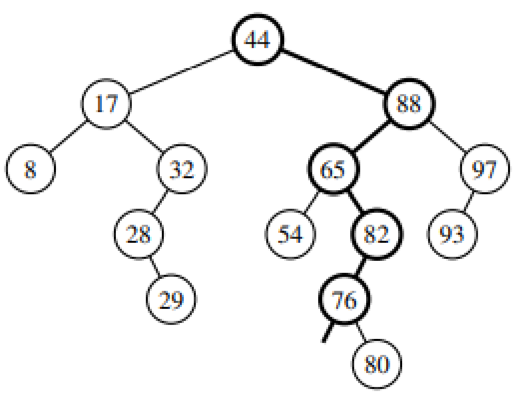
A binary search tree with integer keys.

BST: Search

- **search(root, k):** searches the node storing key ***k***.
- **if** k == root.key:
  - **return** root
- **elif** k < root.key **and** left subtree exists
  - **return** search(root.left, k)
- **elif** k > root.key **and** right subtree exists
  - **return** search(root.right, k)
- **Else:**
  - **return** None
  - (*unsuccessful search*)



Search(root, 65)



Search(root, 68)

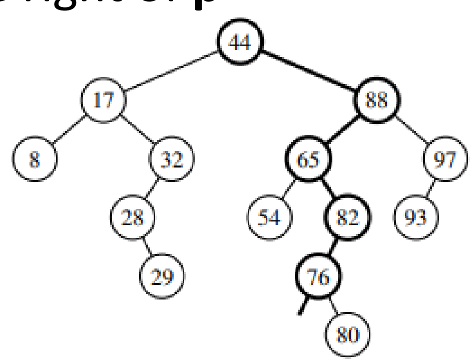
BST: Search parent also

- **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

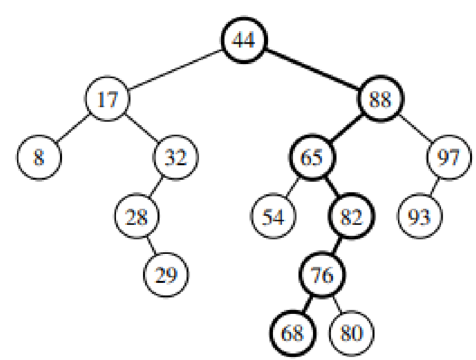
BST: Insertion

- **insert(root, k, v):** inserts the key-value pair (k,v)
- Insertion process:
  - node, p = **search_p(root, k, v)**
  - **if** k==node.key: # key already exists
    - Update **node**.value to **v**
  - **if** k<p.key:
    - Insert (k,v) to the left of **p**
  - **if** k>p.key:
    - Insert (k,v) to the right of **p**

Insert key: 68



Finding the parent



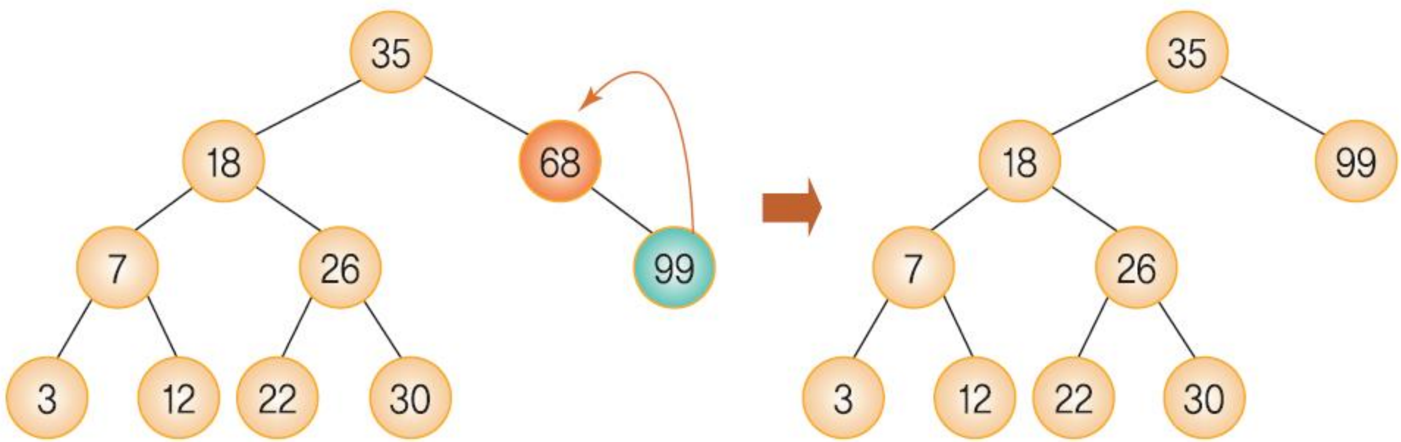
Adding the new key

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

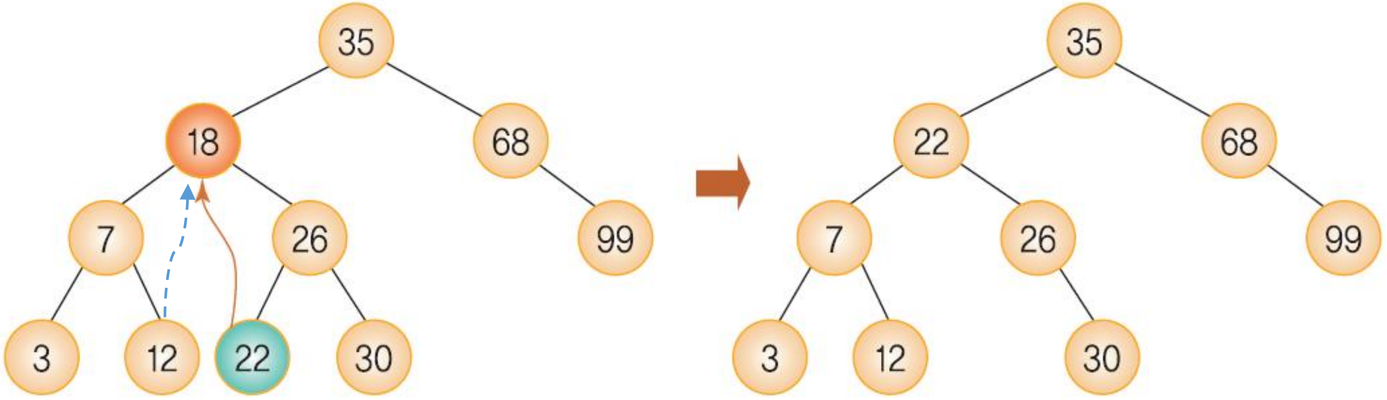
BST: Deletion

- *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*



Running Time

- Depends on tree height
  - Best-case height: log(n)
  - Worst-case height: n