The Australian National University Semester 2, 2024
School of Computing Tutorial 7

Convenor(s): Yun Kuen Cheung (Marco) & Ahad N. Zehmakan

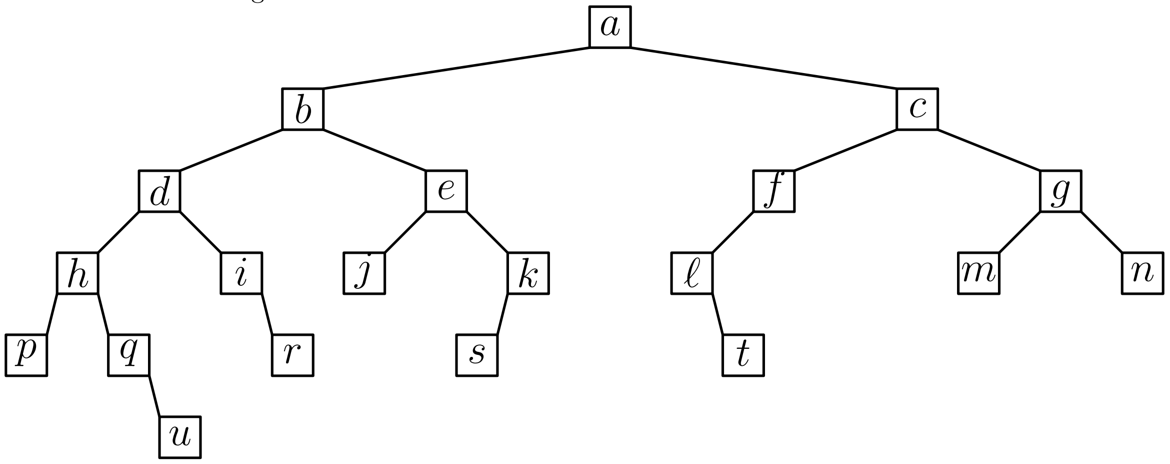
Concrete version: Thursday 12th September, 2024, 09:38 (Compilation date and time)

Algorithms (COMP3600/6466)

Problems marked with (∗) are challenge exercises. They will not be discussed in tutorials, and solutions will not be released. Once confident in your solution for a challenge exercise, you are welcome to discuss it with your tutor during the consultation period in the tutorial session, or schedule a time with Marco to present it.

Exercise 1 ***Binary Search Trees***

1. Consider the following BST.



   (a) What are the predecessors of nodes a, d and ℓ respectively?

   (b) What are the successors of nodes a, p and n respectively?

   (c) Write down the pre-order, in-order and post-order traversals of the above BST.

Solution

(a) The predecessor of node a is node k. The predecessor of node d is node u. The predecessor of node ℓ is node a.

(b) The successor of node a is node ℓ. The successor of node p is node h. The successor of node n does not exist, since n contains the largest key.

(c) Pre-order: a, b, d, h, p, q, u, i, r, e, j, k, s, c, f, ℓ, t, g, m, n
In-order: p, h, q, u, d, i, r, b, j, e, s, k, a, ℓ, t, f, c, m, g, n
Post-order: p, u, q, h, r, i, d, j, s, k, e, b, t, ℓ, f, m, n, g, c, a

2. The post-order traversal of a BST outputs the following sequence of keys:

$$1, 3, 4, 5, 2, 8, 9, 7, 6, 16, 19, 18, 15, 21, 24, 20, 10.$$

   (a) Draw the BST.

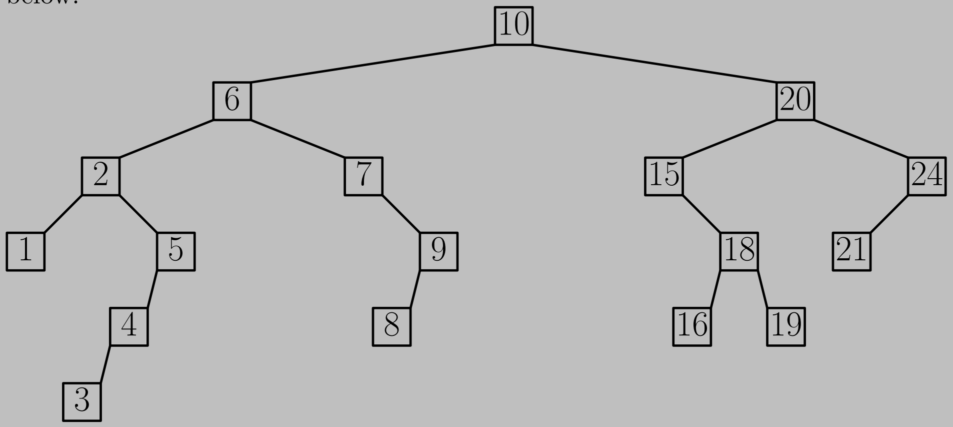
   (b) What is the pre-order traversal of the BST?

   (c) (*) Given any post-order traversal sequence of a BST saved in an array as input, present an algorithm which outputs the pre-order traversal sequence of the same BST. What is the runtime of your algorithm?

Solution

(a) Recall that in post-order traversal, the root is the last node to traverse. This means in our case the root has key 10. Then we know the nodes with keys $1, 3, 4, 5, 2, 8, 9, 7, 6$ are in the left subtree of the root, and $1, 3, 4, 5, 2, 8, 9, 7, 6$ is the post-order traversal of the left subtree. Analogously, $16, 19, 18, 15, 21, 24, 20$ is the post-order traversal of the right subtree of the root. One step further, the post-order of the left subtree of node 6 is $1, 3, 4, 5, 2$, while the post-order of the right subtree of node 6 is $8, 9, 7$.

By recursive construction following the above logic, the BST can be re-constructed, given below:



(b) The pre-order traversal is $10, 6, 2, 1, 5, 4, 3, 7, 9, 8, 20, 15, 18, 16, 19, 24, 21$.

3. Write down code (or pseudo-code) that implements the following operations on binary search trees. You may use operations we have discussed in the lecture as subroutines.

   (a) successor(target): given a target, return the node which has the smallest key among all nodes with keys larger than the target. You can assume that the target exists in the BST.

   (b) find-min(root): given the root of a BST, return the node which has the smallest key in the BST.

   (c) find-max(root): given the root of a BST, return the node which has the largest key in the BST.

   (d) delete-min(root): given the root of a BST, remove the node which has the smallest key in the BST.

   (e) delete-max(root): given the root of a BST, remove the node which has the largest key in the BST.

Solution

(a)
```
def successor(target):
    node = search(target)
    if node.right is not None:
        node = node.right
        while node.left is not None:
            node = node.left
        return node
    else:
        while node.parent is not None:
            if target < node.parent.key:
                return node.parent
            node = node.parent
        return None
```

(b)
```
def find_min(root):
    node = root
    while node.left is not None:
        node = node.left
    return node
```

(c)
```
def find_max(root):
    node = root
    while node.right is not None:
        node = node.right
    return node
```

(d)
```
def delete_min(root):
    node = find_min(root)
    delete(root, node.key)
```

(e)
```
def delete_max(root):
    node = find_max(root)
    delete(root, node.key)
```

4. Given a non-empty BST with n nodes as input.

    (a) Present a divide-and-conquer style algorithm which computes, for each node v in the BST, the height of the subtree rooted at v. The number is saved in the member variable v.height of each node object. Your algorithm must run in $\mathcal{O}(n)$ time.

    (b) Present an algorithm which determines whether the BST is height-balanced or not. Your algorithm must run in $\mathcal{O}(n)$ time.

Solution

(a) The key observation is that for a node v which is not **None**, the height of the subtree rooted at v is equal to one plus the maximum of the height of the left subtree of v, and the height of the right subtree of v.

A commonly used trick to simplify the implementation is to assume that a **None** node has height -1.

```
def compute_heights(node):
    if node is None:
        return -1
    node.height = max(compute_heights(node.left), compute_heights(node.right), 0) + 1
    return node.height
```

Once after the above procedure defined, call compute_heights(root)

(b) The key idea is that in part (a), when computing node.height, we can also check if node is balanced or not. Python allows a procedure returning a tuple, which makes the code below simpler. There are two values to be returned, the first is the height of the node, and second is whether the node is balanced or not.

If you use C++, Java or some other PL, the code implementation may be slightly more complex, for which we do not go into. (Algorithm course focuses on algorithmic idea, not code implementation.)

```
def check_height_balance(node):
    if node is None:
        return -1, True
    lheight, lbalance = check_height_balance(node.left)
    rheight, rbalance = check_height_balance(node.right)
    balance = (abs(lheight - rheight) <= 1)
    node.height = max(lheight, rheight) + 1
    return node.height, (lbalance and rbalance and balance)
```

Once after the above procedure defined, call check_height_balance(root)[1]

5. Given a non-empty BST with n nodes as input.

   (a) Present an algorithm which computes, for each node v in the BST, the number of nodes in the subtree rooted at v. The number is saved in the member variable v.number_nodes of each node object. Your algorithm must run in $\mathcal{O}(n)$ time.

   (b) Using part (a) as a subroutine, present an algorithm which takes k as input, where $1 \leq k \leq n$, and outputs the k-th smallest key in the BST. Your algorithm must run in $\mathcal{O}(n)$ time.

   (c) (*) Present another algorithm for the problem in part (b), but with runtime $\mathcal{O}(h + k)$, where h is the height of the BST.

Solution

(a) Given that you have already done 4(a), this can be similarly done.

```
def compute_number_nodes(node):
    if node is None:
        return 0
    node.size = compute_number_nodes(node.left) + compute_number_nodes(node.right) + 1
    return node.size
```

Once after the above procedure defined, call compute_number_nodes(root)

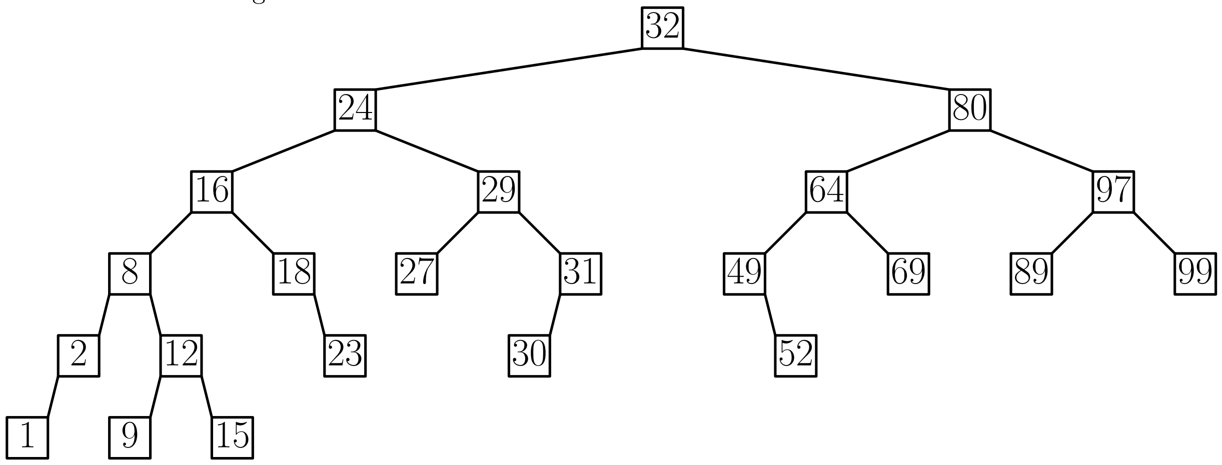
(b) First, call compute_number_nodes(root) which computes node.size for each node in the BST. By using these sizes, at any given node, we can determine which subtree to go downward to find the k-th key of the BST.

```
def find_kth_key(node, k):
    if node.left is None:
        if $k$ == 1:
            return node.key
        return find_kth_key(node.right, k-1)
    if $k$ == node.left.size + 1:
        return node.key
    elif $k$ <= node.left.size:
        return find_kth_key(node.left, k)
    else:
        return find_kth_key(node.right, k-node.left.size-1)
```

Once after the above procedure defined, call find_kth_key(root, k)

Exercise 2 ***Balanced Binary Search Trees***

6. Consider the following AVL tree.

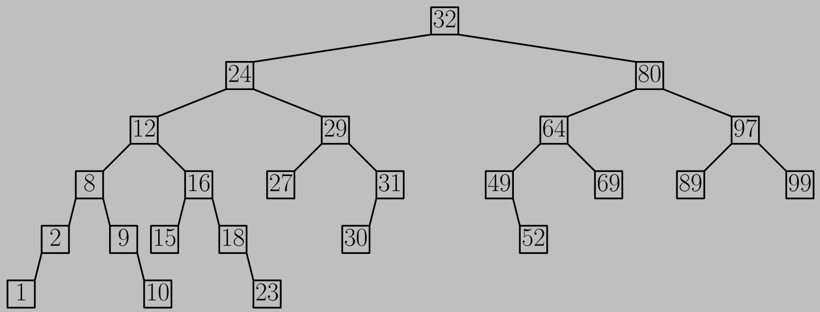


Perform the following operations step-by-step, and draw the new AVL tree after each step.

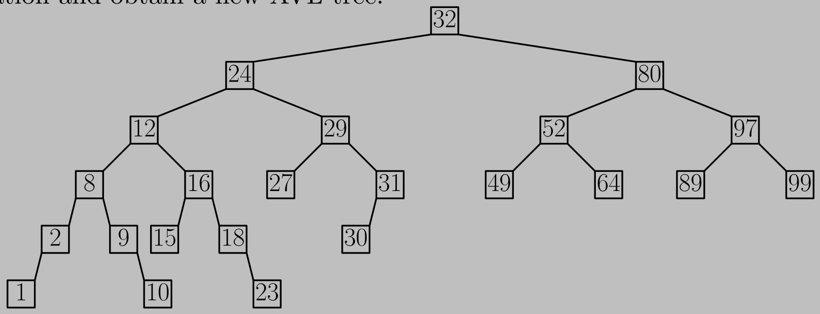
(a) Insert a new key 10.
(b) Then delete the node with key 69.
(c) Then delete the node with key 24.

Solution

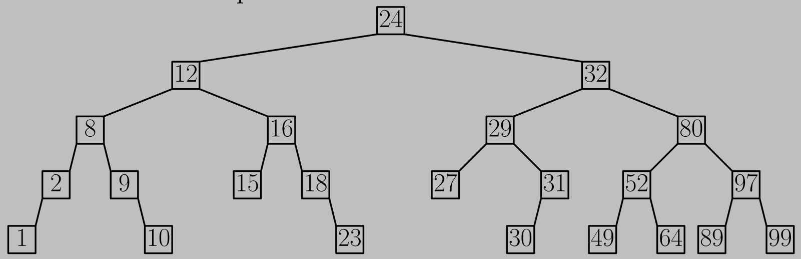
(a) After inserting a new node 10 as a leaf under the node 9, we check whether the ancestors of 9 are balanced or not. Nodes $9, 12, 8$ are balanced, but node 16 is not. We should perform an LR rotation where $x = 16$, $y = 8$ and $z = 12$. (x, y, z are notations used in the lecture slide about LR rotation.) After this rotation, the tree becomes height-balanced again. The new AVL tree is



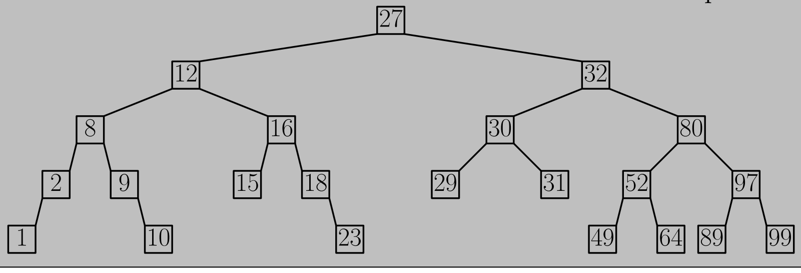
(b) After deleting the node 69 which is a leaf, node 64 becomes unbalanced. We perform LR rotation and obtain a new AVL tree:



Then node 80 is balanced, but node 32 is not. An LL rotation is performed to rebalance. The final AVL tree of this step is:



(c) When deleting node 24, we find its successor (node 27) and "relocate" the successor to where node 24 used to be. Then we check from the original parent of node 27, i.e., node 29 to see if it is balanced or not. It is not, and an RL rotation is performed. Then we check that both node 32 and node 27 are balanced. The final AVL tree of this step is:



7. (*) Given a non-empty BST with n nodes, which can be unbalanced. Present an $\mathcal{O}(n)$-runtime algorithm that constructs a new height-balanced BST with the same set of keys as the input BST.