



Data Structures

Chapter 5 Tree

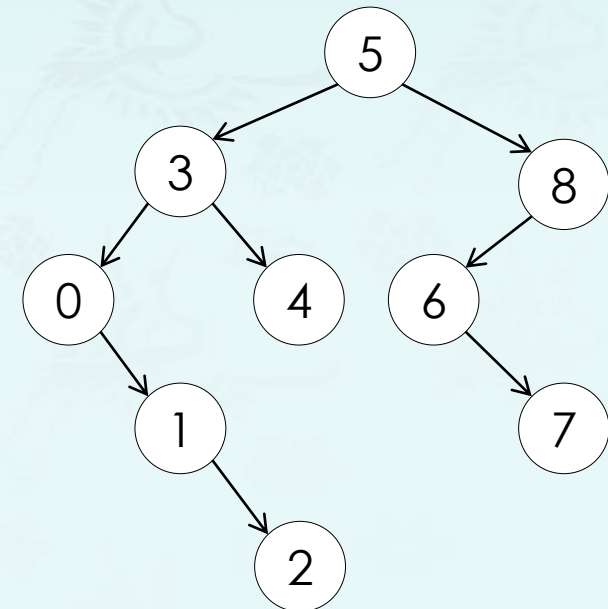
1. Introduction
2. Binary Tree
- 3. Binary Search Tree**
 - Introduction
 - Operations
 - **Demo & Coding**
4. Balancing Tree

Minimum, Maximum:

- `Minimum()` and `maximum()` returns the node with min or max key.
 - Note that the entire tree does not need to be searched.
 - The minimum key is always located at the left most node, the maximum at the right most node.
 - Complexity of algorithm to find the maximum or minimum will be $O(\log N)$ in almost balanced binary tree. If tree is skewed, then we have worst case complexity of $O(N)$.

```
tree minimum(tree node) { // returns left-most node key
    if (node->left == nullptr) return node;
    return minimum(node->left);
}
```

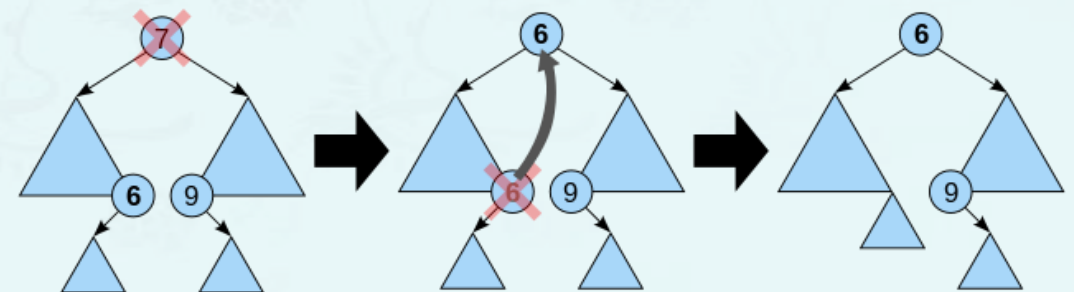
```
tree maximum(tree node) { // returns right-most node key
    if (node->right == nullptr) return node;
    return maximum(node->right);
}
```



pred(), succ() – predecessor, successor:

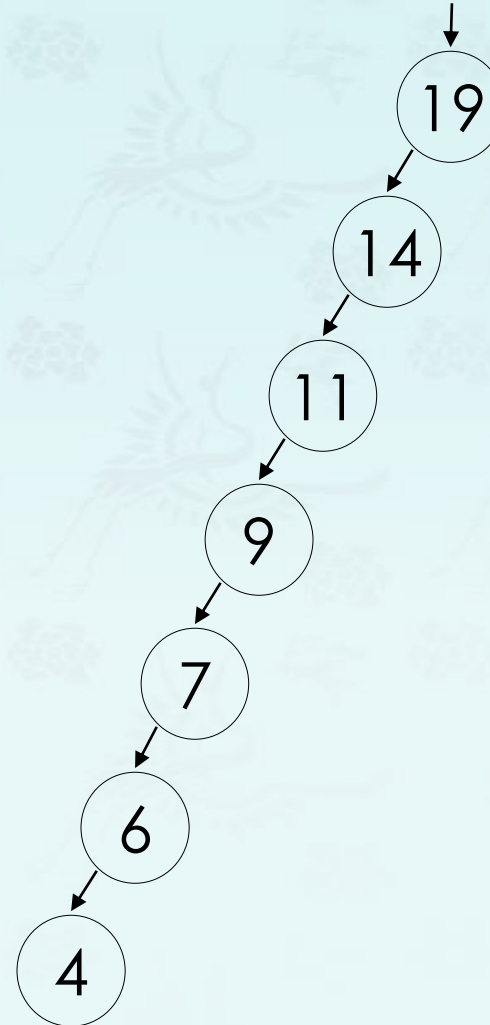
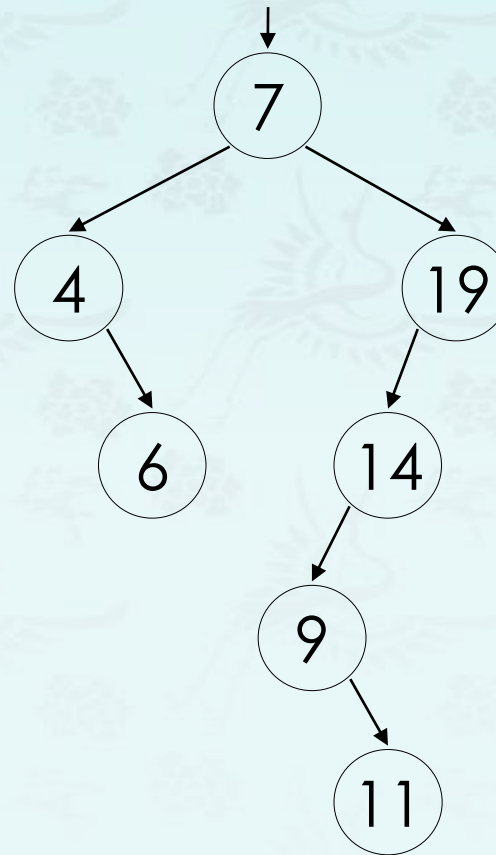
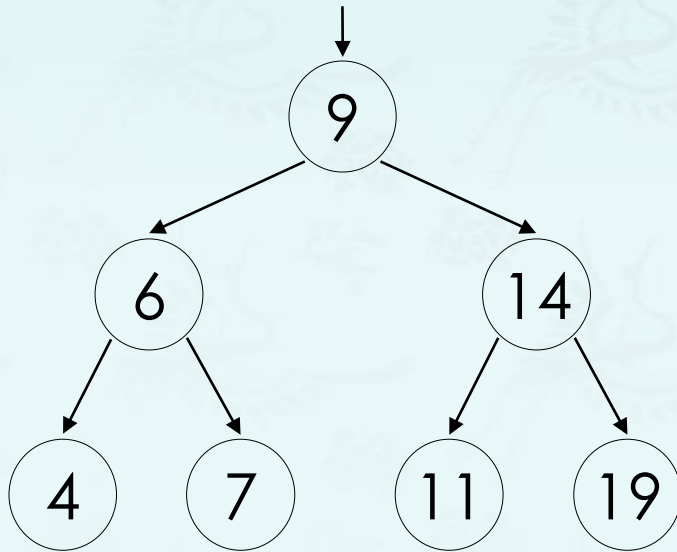
- Successor
 - If the given node has a right subtree then by the BST property the next larger key must be in the right subtree. Since all keys in a right subtree are larger than the key of the given node, the successor must be the smallest of all those keys in the right subtree.
- Predecessor
 - If the given node has a left subtree then by the BST property the next smaller key must be in the left subtree. Since all keys in a left subtree are smaller than the key of the given node, the predecessor must be the largest of all those keys in the left subtree.
- Complexity of algorithm
 - $O(\log N)$ in almost balanced binary tree. If tree is skewed, then we have worst case complexity of $O(N)$.

```
tree successor(tree node) {  
    if (node != nullptr && node->right != nullptr)  
        return minimum(node->right);  
    return nullptr;  
}
```



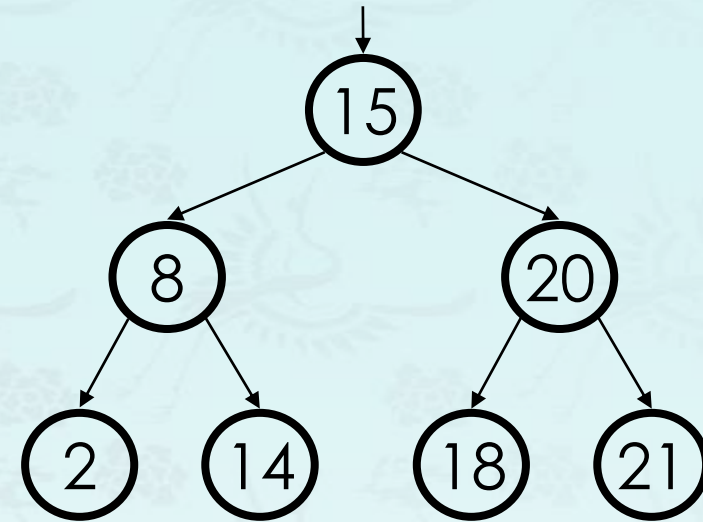
Binary Search Trees: Observations

- What do you see in the following BSTs?
 - A **balanced** tree of N nodes has a height of $\sim \log_2 N$.
 - A very **unbalanced** tree can have a height close to N .



Binary Search Trees: Observations

- For binary tree of height h :
 - max # of leaves: 2^h
 - max # of nodes: $2^{h+1} - 1$
 - min # of leaves: 1
 - min # of nodes: $h + 1$
- The shallower the BST the better.
 - Average case height is $O(\log N)$
 - Worst case height is $O(N)$
 - Simple cases such as adding $(1, 2, 3, \dots, N)$, or the opposite order, lead to the worst case scenario: height $O(N)$.



Binary Search Trees: Observations

- Q: If you have a sorted sequence, and we want to design a data structure for it. Which one are you going to use an array or BST? and why?

Time Complexity	
BST	$O(h)$
Array	$O(\log n)$

- Q: When searching, we're traversing a path (since we're always moving to one of the children); since the length of the longest path is the height h of the binary search tree, then finding an element takes $O(h)$.
 - Since $h = \log n$ (where n is the number of elements), then it's good! – right?
 - No, of course, it is wrong! Why?

A: The nodes could be arranged in linear sequence in BST, so the *height* h could be n . In worst case, it is $O(n)$ instead of $O(h)$.

Operations: growN() & trimN() for testing

- It performs a user specified number of insertion(or grow) or deletion(or trim) of nodes in the tree.
- The function **growN()** inserts a user specified number N of nodes in the tree.
 - If it is an empty tree, the value of keys to add ranges from 0 to N-1.
 - If there are some existing nodes in the tree, the value of keys to add ranges from $\text{max} + 1$ to $\text{max} + 1 + N$, where max is the maximum value of keys in the tree.
- This function growN() is provided for your reference^^.

Operations: growN() & trimN() for testing

- The function **trimN()** deletes N number of nodes in the tree.
 - The nodes to trim are **randomly** selected from the tree.
 - If N is less than the tree size (which is not N), you just trim N nodes.
 - If the N is larger than the tree size, set it to the tree size.
 - At any case, you should trim all nodes one by one, but randomly.
 - With an AVL tree, reconstruct it **after** trimming N nodes from BST.

Step 1: Get a list (vector) of all keys from the tree first.

Get the size of the tree using the size().

Use assert to check two sizes;

Step 2: Shuffle the vector with keys. – shuffle()

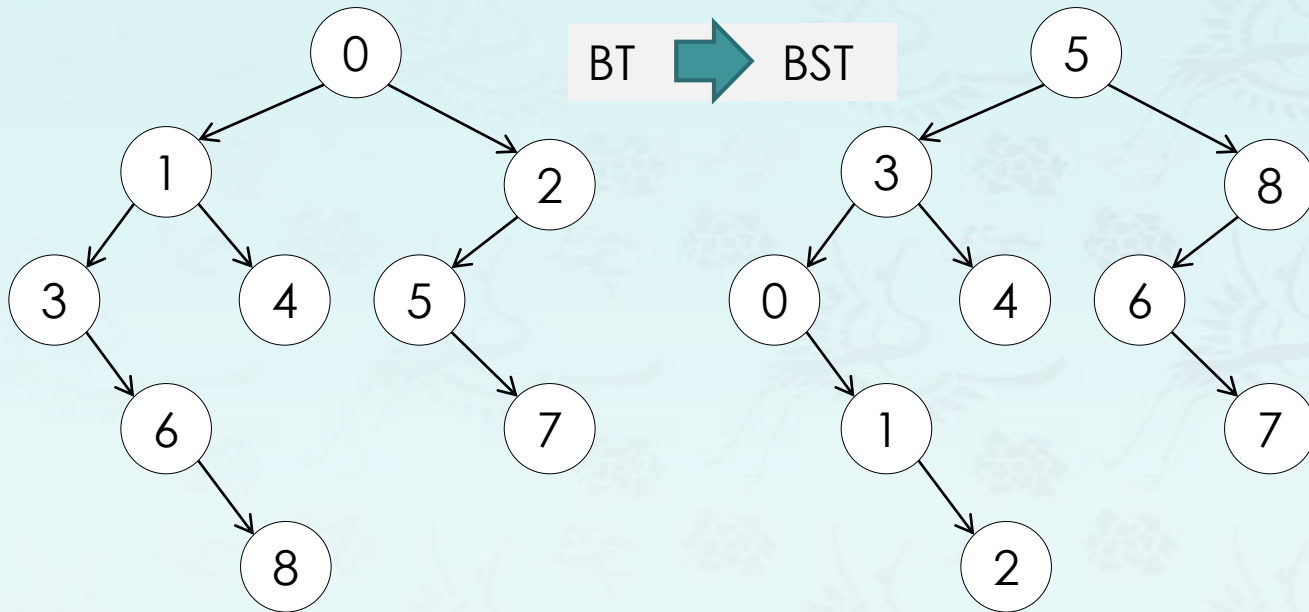
Step 3: Invoke trim() N times with a key from the vector in sequence.

Inside a for loop, trim() may return a new root of the tree.

Step 4: The function is called with AVLtree = true, then reconstruct the tree.

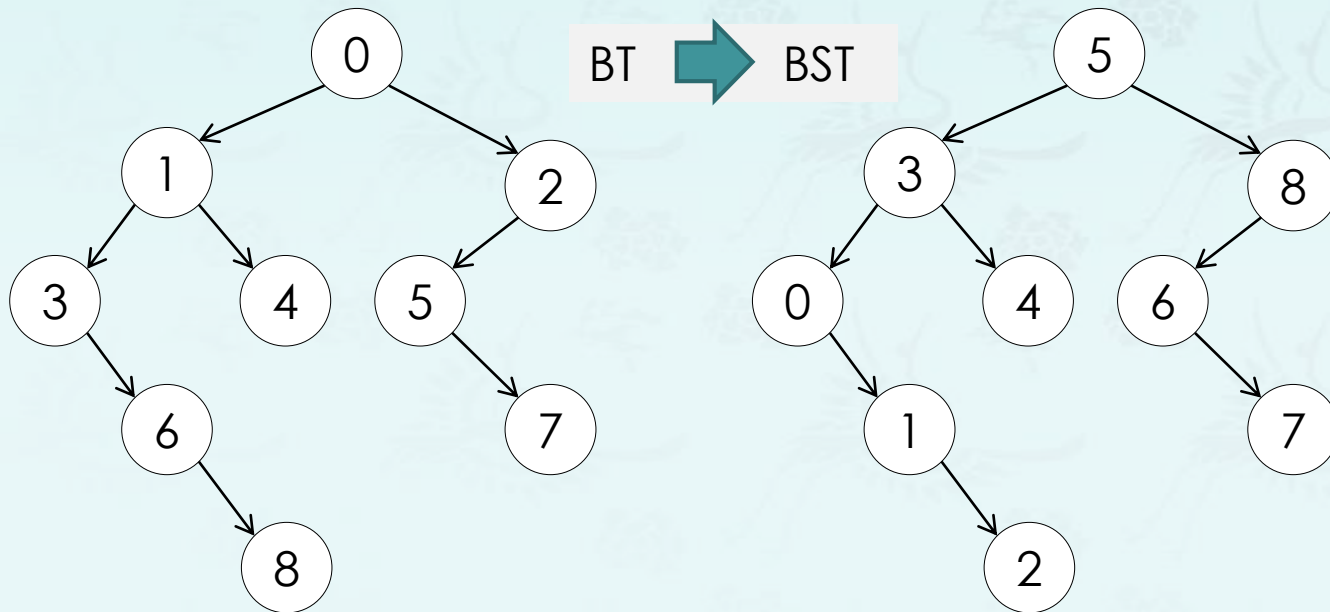
Convert BT to BST in-place

- Convert a binary tree to a binary search tree while keeping its tree structure as it is.
- For example:



Convert BT to BST in-place

- Convert a binary tree to a binary search tree while keeping its tree structure as it is.
- Algorithm:
 - Step 1** – store keys of a binary tree into a container like **vector** or **set**. (**Do not use an array.**)
 - Step 2** – sort the keys in vector. Skip this step if set is used since it is already sorted.
 - Step 3** – Now, do the **inorder** traversal of the tree and copy back the elements of the container into the nodes of the tree one by one.



- (1) Retrieve the keys from BT:
3 6 8 1 4 0 5 7 2 // if in-order used
- (2) Sort keys in the container:
0 1 2 3 4 5 6 7 8
- (3) Replace keys in BT with sorted keys while in-order traversal.

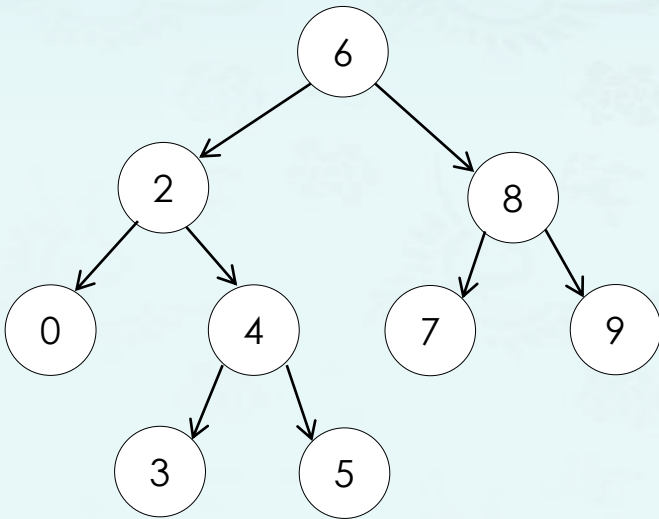
```
void inorder(tree root) {
    if (root == nullptr) return;

    inorder(root->left);
    cout << root->key;
    inorder(root->right);
}
```

L
V
R

Operations: LCA in BST

- Find the lowest common ancestor(LCA) of two given nodes, given in BST.
 - The LCA is defined between two nodes p and q as the lowest node in T that has both p and q as descendants (where we allow a node to be a descendant of itself)."
 - In BST, all of the nodes' values will be unique.
Two nodes given, p and q, are different and both values will exist in the BST.



For example:

2, 8 -> 6

2, 5 -> 2

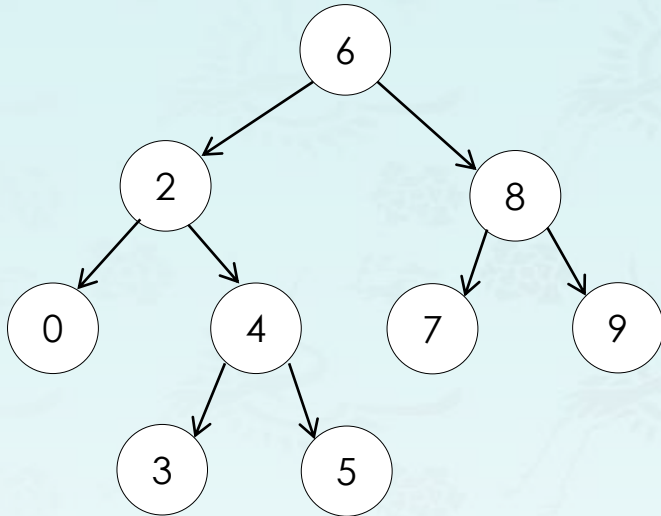
9, 5 -> 6

8, 7 -> 8

0, 5 -> 2

Operations: LCA(**iteration**) in BST

- **Intuition (Iteration):** Traverse down the tree iteratively to **find the split point**. The point from where p and q won't be part of the same subtree or when one is the parent of the other.



For example:

2, 5 -> 2

9, 7 -> 8

0, 4 -> 2

0, 5 -> 2

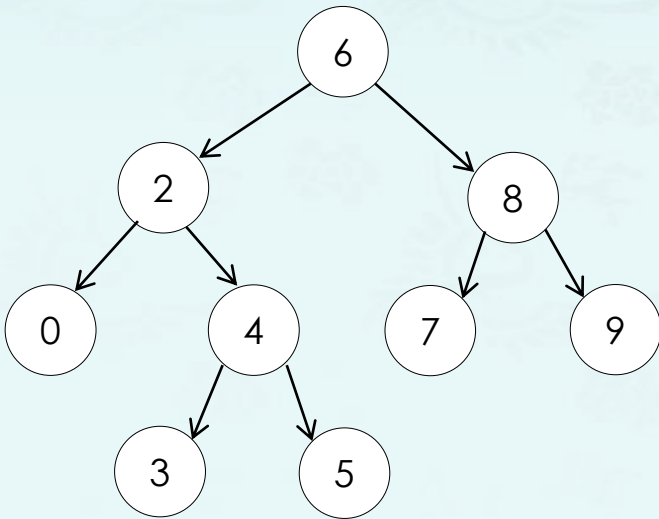
2, 7 -> 6

```
int LCAiteration(tree node, tree p, tree q) {
    while (node != nullptr) {
        if (both p & q > node)
            node move to right to search
        else if (both p & q < node)
            node moves to left to search
        else
            return node->key    // found
    }
    return 0;    // not found
} // iteration solution
```

Operations: LCA(**recursion**) in BST

■ Algorithm: (Recursion)

1. Start traversing the tree from the root node.
2. If both the nodes p and q are in the right subtree, then continue the search with right subtree starting step 1.
3. If both the nodes p and q are in the left subtree, then continue the search with left subtree starting step 1.
4. If both step 2 and step 3 are **not true**, this means we have **found** the node which is common to node p's and q's subtrees. Hence we return this common node as the LCA.



```
tree LCA(tree root, tree p, tree q) {  
  
    // your code here  
  
} // recursive solution
```

Operations: LCA in BST

- Recursion Algorithm
 - Time Complexity: $O(N)$, where N is the number of nodes in the BST. In the worst case we might be visiting all the nodes of the BST.
 - Space Complexity: $O(N)$. This is because the maximum amount of space utilized by the recursion stack would be N since the height of a skewed BST could be N .
- Iteration Algorithm
 - Time Complexity : $O(N)$, where N is the number of nodes in the BST. In the worst case we might be visiting all the nodes of the BST.
 - Space Complexity : $O(1)$.



Data Structures

Chapter 5 Tree

1. Introduction
2. Binary Tree
- 3. Binary Search Tree**
 - Definition
 - Operations
 - Demo & Coding
4. Balancing Tree