Tutorial 7: Binary Search Tree

CSC3100 Data Structures

Chen Shi

224040349@link.cuhk.edu.cn

Spring 2025

Table of Contents

What is a Binary Search Tree?

What is a Binary Tree?

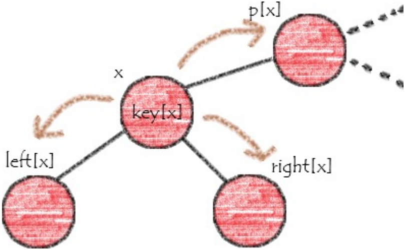
A tree in which each node has at most two children.

What is a Binary **Search** Tree?

A tree in which each node has at most two children **and** is organized such that nodes with smaller values are on the left and larger ones on the right.

What is a Binary **Search** Tree?

A tree in which each node has at most two children **and** is organized such that nodes with smaller values are on the left and larger ones on the right.

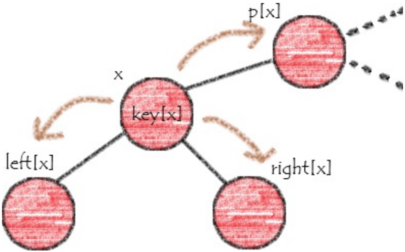


Each node x has the following attributes:

- A node key, denoted as key[x]
- A parent pointer, denoted as p[x]
- A left child pointer, denoted as left[x]
- A right child pointer, denoted as right[x]

What is a Binary **Search** Tree?

A tree in which each node has at most two children **and** is organized such that nodes with smaller values are on the left and larger ones on the right.

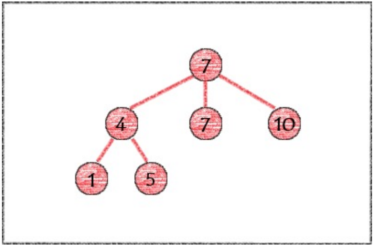


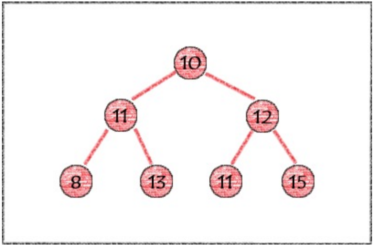
For each node x, the following holds:

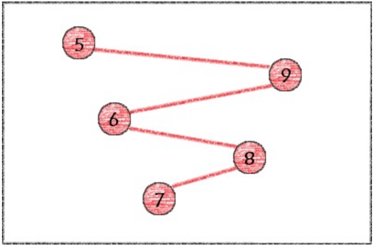
- key[left[x]] ≤ key[x] if left[x] exists
- key[right[x]] ≥ key[x] if right[x] exists

Exercise 1

Which of the following is/are binary search trees?

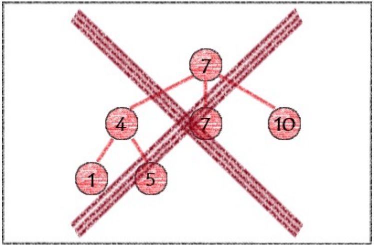


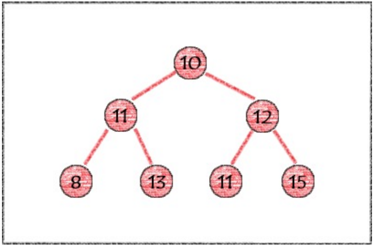


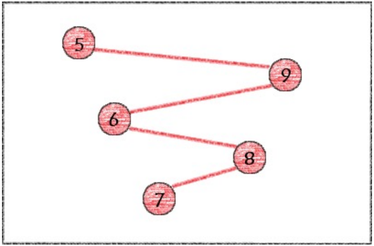


Exercise 1

Which of the following is/are binary search trees?

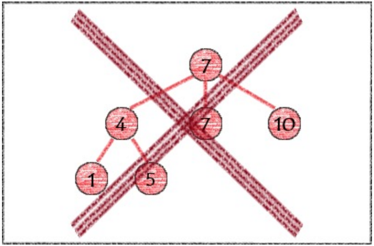


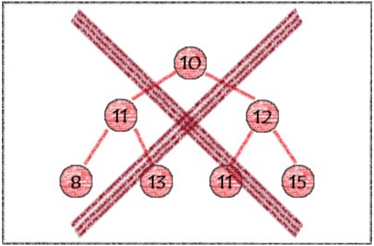


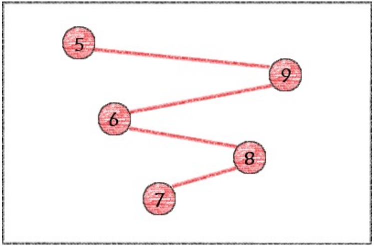


Exercise 1

Which of the following is/are binary search trees?

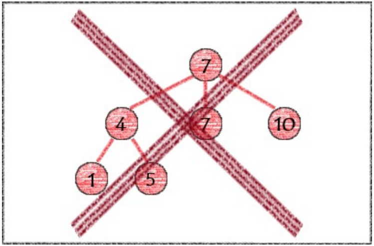


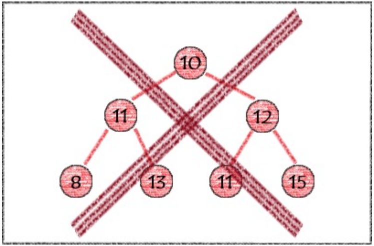




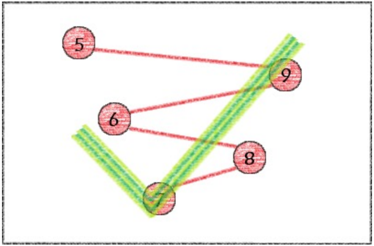
Exercise 1

Which of the following is/are binary search trees?





Inorder Traversal

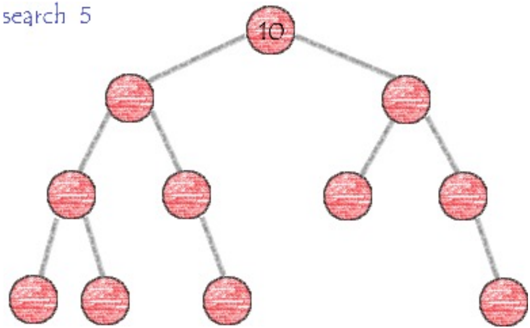


Querying a Binary Search Tree

- Searching
  Given a pointer to the root of the tree and a key k, return a pointer to a node with key k if one exists; otherwise, return NIL.
- Minimum (Maximum)
  Return an element whose key is a minimum (maximum).
- Successor (Predecessor)
  Given a pointer to a node of the tree, return its successor (predecessor).
  - The successor of a node x is the node with the smallest key greater than key[x].
  - The predecessor of a node x is the node with the largest key smaller than key[x].

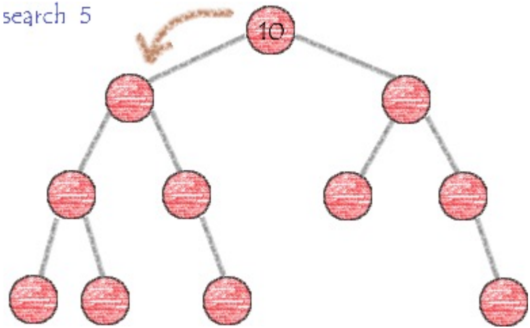
Searching in a BST

Given a pointer to the root of the tree and a key k, return a pointer to a node with key k if one exists; otherwise, return NIL.



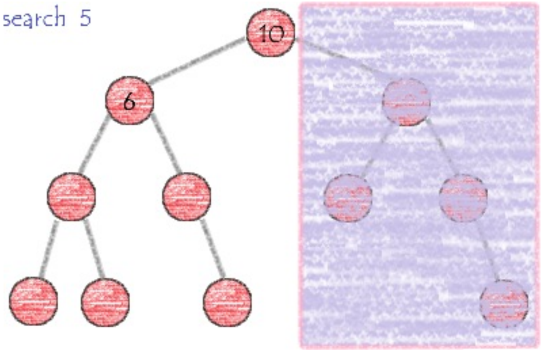
Searching in a BST

Given a pointer to the root of the tree and a key k, return a pointer to a node with key k if one exists; otherwise, return NIL.



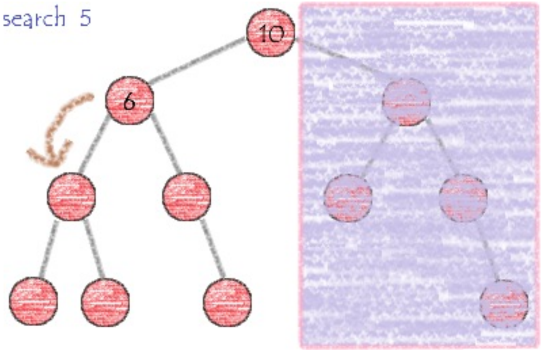
Searching in a BST

Given a pointer to the root of the tree and a key k, return a pointer to a node with key k if one exists; otherwise, return NIL.



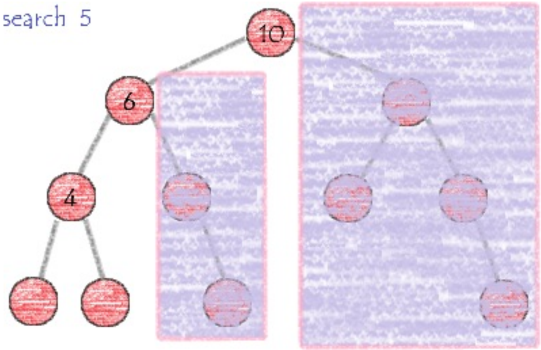
Searching in a BST

Given a pointer to the root of the tree and a key k, return a pointer to a node with key k if one exists; otherwise, return NIL.



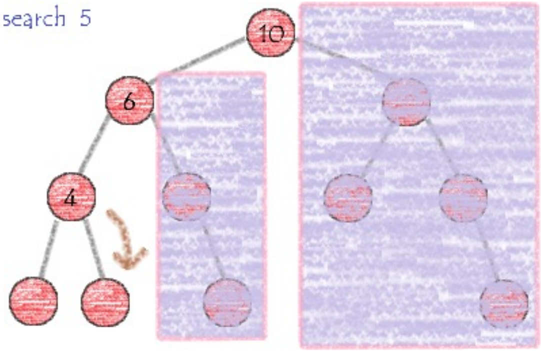
Searching in a BST

Given a pointer to the root of the tree and a key k, return a pointer to a node with key k if one exists; otherwise, return NIL.



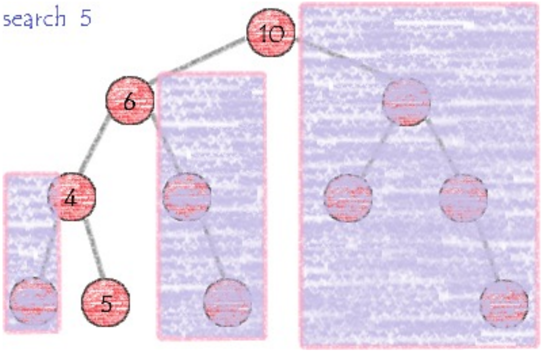
Searching in a BST

Given a pointer to the root of the tree and a key k, return a pointer to a node with key k if one exists; otherwise, return NIL.



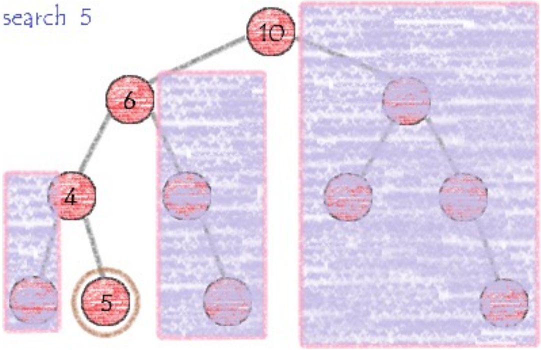
Searching in a BST

Given a pointer to the root of the tree and a key k, return a pointer to a node with key k if one exists; otherwise, return NIL.



Searching in a BST

Given a pointer to the root of the tree and a key k, return a pointer to a node with key k if one exists; otherwise, return NIL.



Searching in a BST

Given a pointer to the root of the tree and a key k, return a pointer to a node with key k if one exists; otherwise, return NIL.

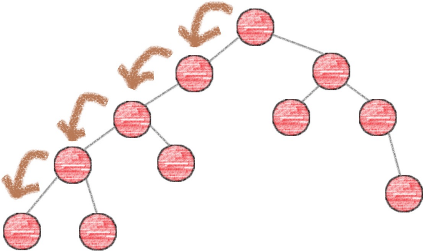
```
Function TREE-SEARCH(Node x, Key k)
    if x == NIL or k == x.key
        return x
    if k < x.key
        return TREE-SEARCH(x.left, k)
    else
        return TREE-SEARCH(x.right, k)
```

Finding Minimum in a BST

Return an element whose key is a minimum.

Finding Minimum in a BST

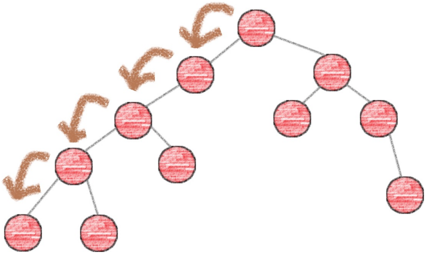
Return an element whose key is a minimum.



Finding an element in a binary search tree whose key is a minimum is done by keep following left child pointers from the root.

Finding Minimum in a BST

Return an element whose key is a minimum.

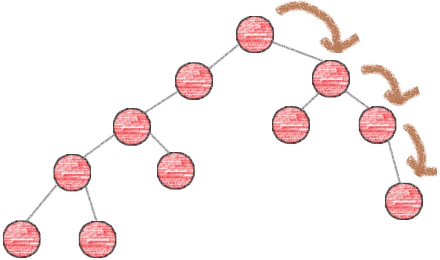


Finding an element in a binary search tree whose key is a minimum is done by keep following left child pointers from the root.

```
Function TREE-MINIMUM(Node x)
  while (x.left != NIL)
    x = x.left
  return x
```

Finding Maximum in a BST

Return an element whose key is a maximum.

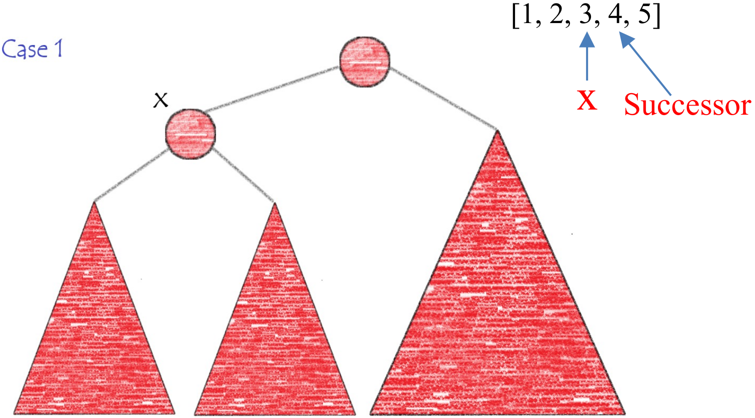


Finding an element in a binary search tree whose key is a maximum is done by keep following right child pointers from the root.

```
Function TREE-MAXIMUM(Node x)
    while (x.right != NIL)
        x = x.right
    return x
```

Finding Successor in a BST

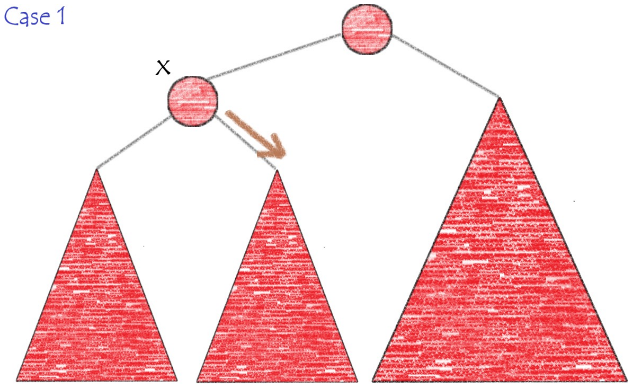
Given a pointer to a node of the tree, return its successor, the node with the smallest key greater than key[x].



Case 1: If the right subtree of node x is nonempty, then the successor of x is just the leftmost node in x's right subtree.

Finding Successor in a BST

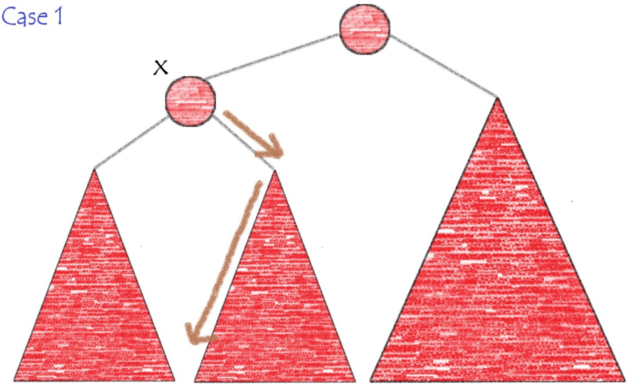
Given a pointer to a node of the tree, return its successor, the node with the smallest key greater than key[x].



Case 1: If the right subtree of node x is nonempty, then the successor of x is just the leftmost node in x's right subtree.

Finding Successor in a BST

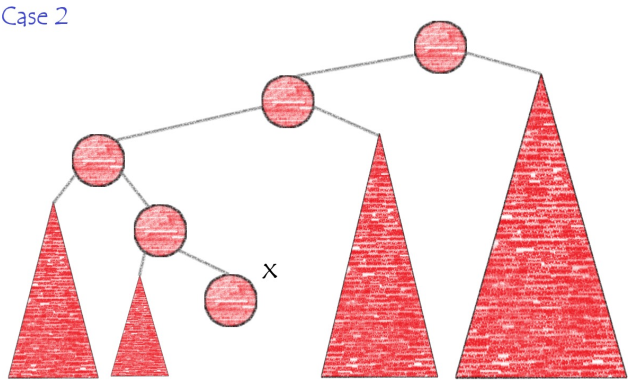
Given a pointer to a node of the tree, return its successor, the node with the smallest key greater than key[x].



Case 1: If the right subtree of node x is nonempty, then the successor of x is just the leftmost node in x's right subtree.

Finding Successor in a BST

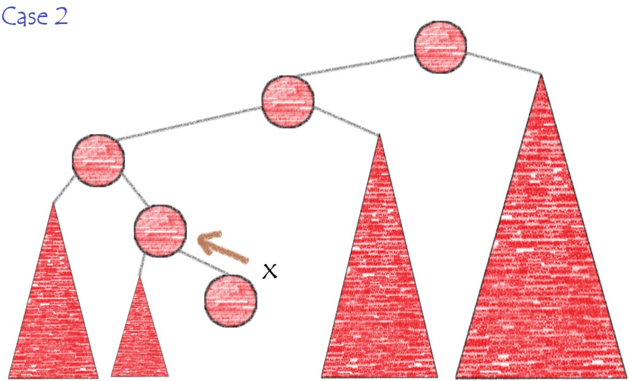
Given a pointer to a node of the tree, return its successor, the node with the smallest key greater than key[x].



Case 2: If the right subtree of node x is empty and x has a successor y, then y is the lowest ancestor of x whose left child is also an ancestor of x.

Finding Successor in a BST

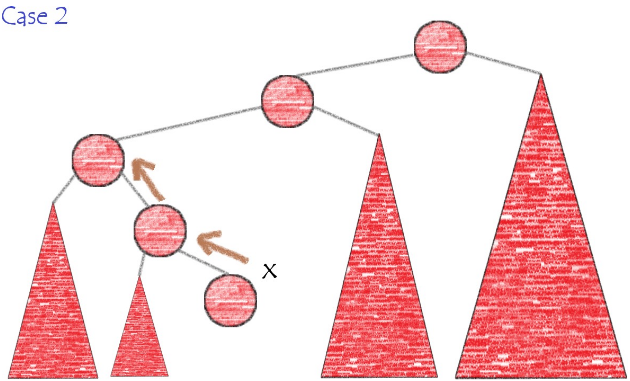
Given a pointer to a node of the tree, return its successor, the node with the smallest key greater than key[x].



Case 2: If the right subtree of node x is empty and x has a successor y, then y is the lowest ancestor of x whose left child is also an ancestor of x.

Finding Successor in a BST

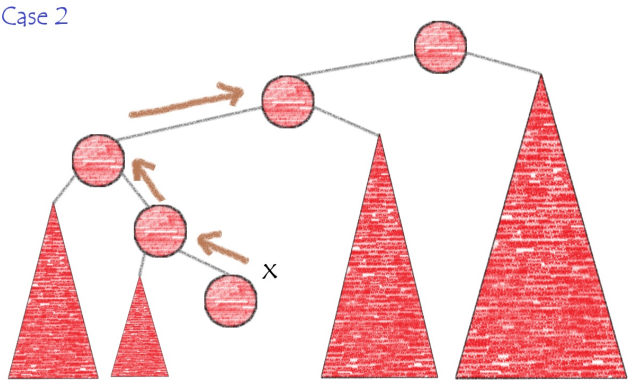
Given a pointer to a node of the tree, return its successor, the node with the smallest key greater than key[x].



Case 2: If the right subtree of node x is empty and x has a successor y, then y is the lowest ancestor of x whose left child is also an ancestor of x.

Finding Successor in a BST

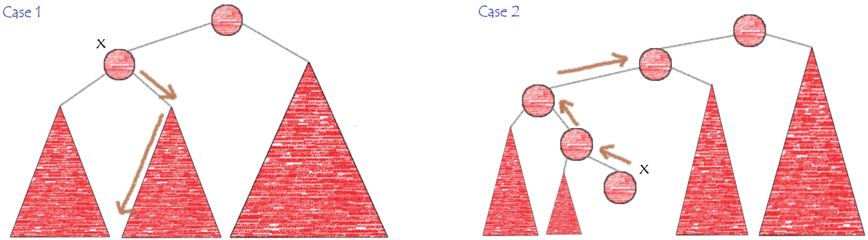
Given a pointer to a node of the tree, return its successor, the node with the smallest key greater than key[x].



Case 2: If the right subtree of node x is empty and x has a successor y, then y is the lowest ancestor of x whose left child is also an ancestor of x.

Finding Successor in a BST

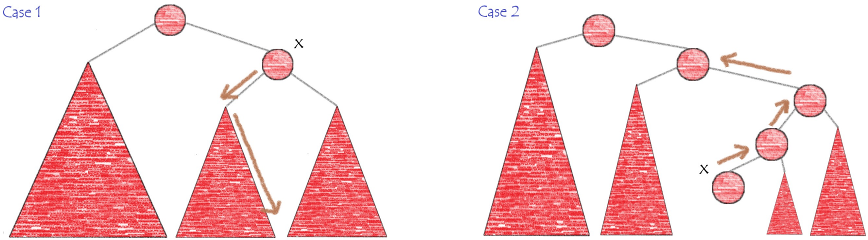
Given a pointer to a node of the tree, return its successor, the node with the smallest key greater than key[x].




```
Function TREE-SUCCESSOR(Node x)
  if (x.right != NIL)
    return TREE-MINIMUM(x.right)
  Node y = x.p
  while (y != NIL) and (x == y.right)
    x = y
    y = y.p
  return y
```

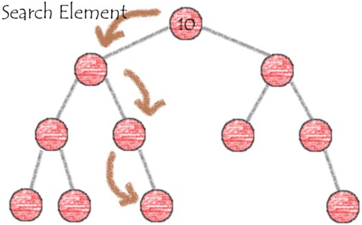
Finding Predecessor in a BST

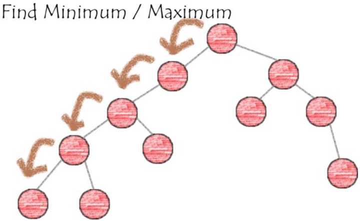
Given a pointer to a node of the tree, return its predecessor, the node with the largest key smaller than key[*x*].

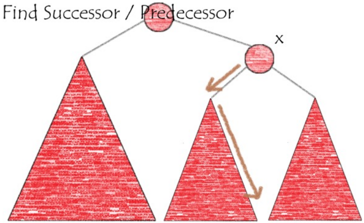



```
Function TREE-PREDECESSOR(Node x)
  if (x.left != NIL)
    return TREE-MAXIMUM(x.left)
  Node y = x.p
  while (y != NIL) and (x == y.left)
    x = y
    y = y.p
  return y
```

Complexity Analysis



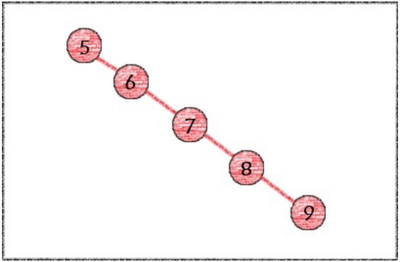


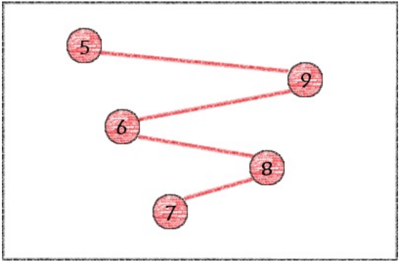


Each query operation runs in $O(h)$ performance, where h is the tree's height.

Complexity Analysis

In the worst-case scenario, the tree's height is exactly n, where n is the number of nodes in the tree.



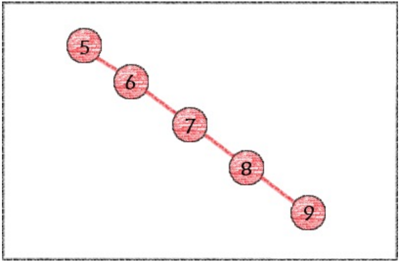


Each query operation runs in $O(n)$ performance.

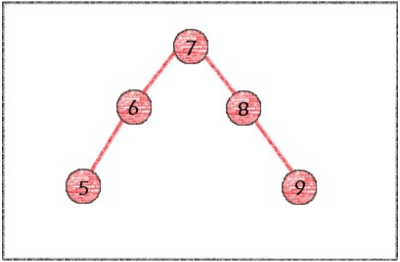
What is a Balanced Binary Search Tree?

What is a Balanced Binary Search Tree?

A binary search tree that maintains a balanced height, that is, to ensure the difference in heights between the left and right subtrees of any node is at most one.



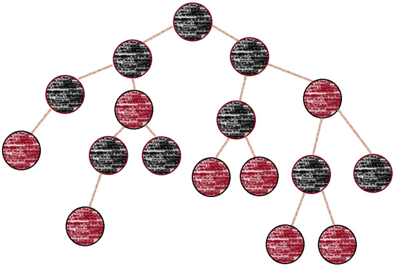
Not a Balanced BST



A Balanced BST

Red-Black Tree (will not be covered in our exams)

Red-Black Tree is a Binary Search Tree with each node colored either red or black.

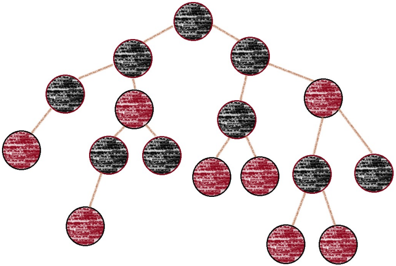


1. The root node must be black
2. All leaf nodes (NIL) are black
3. Both children of a red node must be black
4. Every path from root to any leaf has the same number of black nodes

By constraining the node colors on any simple path from the root to a leaf, red-black trees ensure that no such path is more than twice as long as any other so that the tree is approximately balanced.

Red-Black Tree (will not be covered in our exams)

Red-Black Tree is a Binary Search Tree with each node colored either red or black.



Performance:

- Search runs in $O(\log n)$.
- Insertion runs in $O(\log n)$.
- Deletion operation runs in $O(\log n)$.

AVL Tree (will be covered in Tutorial 8)

AVL Tree, named after its inventors Adelson-Velsky and Landis, is a self-balancing Binary Search Tree. By performing single or double rotations after insertions or deletions, AVL maintains strict height balance: for each node x, the heights of the left and right subtrees of x differ by at most one.

Performance:

- Search runs in $O(\log n)$.
- Insertion runs in $O(\log n)$.
- Deletion operation runs in $O(\log n)$.

LeetCode Exercise

Exercise 2 - Sorted Array to Binary Search Tree

Abridged Statement:

Given an array nums where the elements are sorted in ascending order, convert it into a height-balanced binary search tree.

Sample Input 1:

nums = $[-10, -3, 0, 5, 9]$

Sample Output 1:

$[0, -3, 9, -10, \textit{null}, 5]$
$[0, -10, 5, \textit{null}, -3, \textit{null}, 9]$ is also accepted

Sample Input 2:

nums = $[1, 3]$

Sample Output 2:

$[1, \textit{null}, 3]$
$[3, 1]$ is also accepted

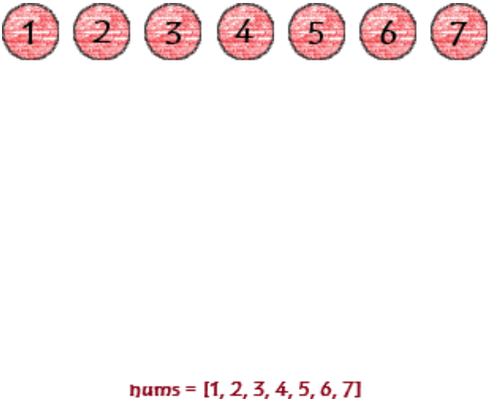
Problem is taken from [leetcode.com/problems/convert-sorted-array-to-binary-search-tree.](https://leetcode.com/problems/convert-sorted-array-to-binary-search-tree)

Sorted Array to Binary Search Tree

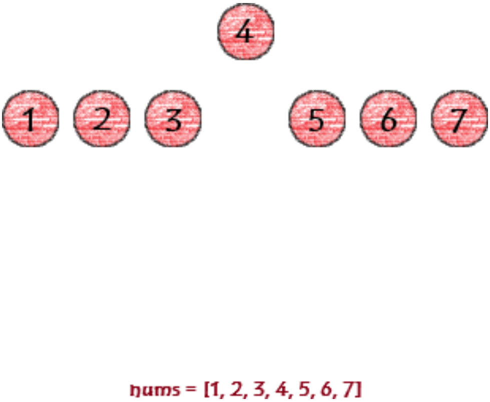
Key observations and implementations:

- Keeping the left and right subtree sizes equal for each node ensures the tree's height is balanced.
- The middle element(s) of the current segment must be the candidate of the root.
- Recursively apply the process to the left and right halves.

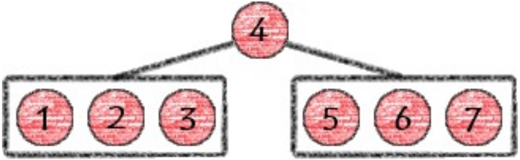
Sorted Array to Binary Search Tree

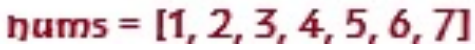


Sorted Array to Binary Search Tree

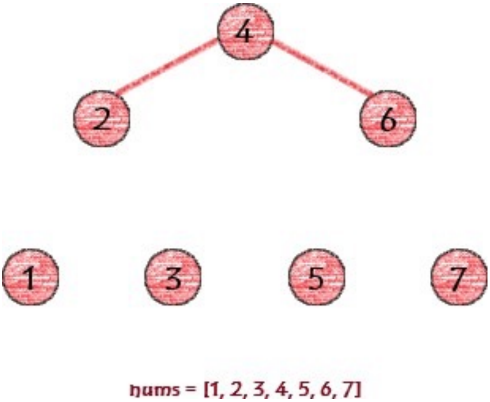


Sorted Array to Binary Search Tree

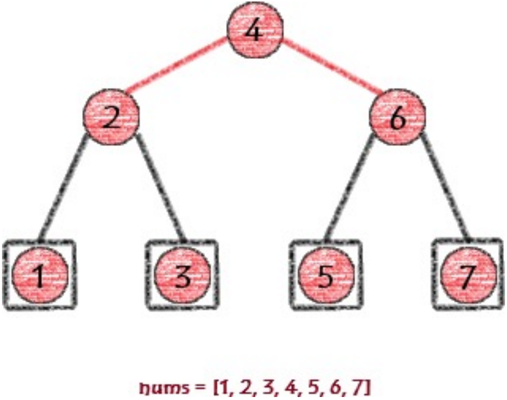




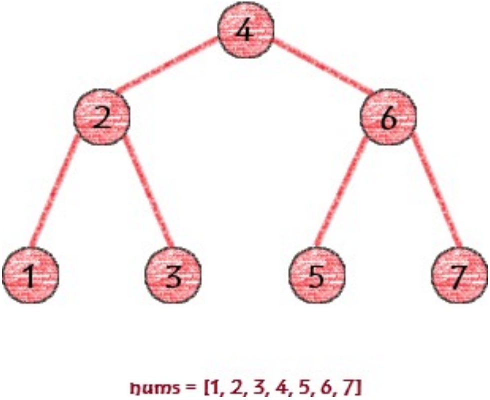
Sorted Array to Binary Search Tree



Sorted Array to Binary Search Tree



Sorted Array to Binary Search Tree



Sorted Array to Binary Search Tree for Python

```
class Solution:
    def create(self, nums, l, r):
        mid = (l + r) // 2
        root = TreeNode(nums[mid])
        if l <= mid - 1:
            root.left = self.create(nums, l, mid - 1)
        if mid + 1 <= r:
            root.right = self.create(nums, mid + 1, r)
        return root

    def sortedArrayToBST(self, nums: List[int]) -> Optional[
    TreeNode]:
        return self.create(nums, 0, len(nums) - 1)
```

Sorted Array to Binary Search Tree for C++

```
class Solution {
public:
  TreeNode *create(vector<int> &nums, int l, int r) {
    int mid = (l + r) / 2;

    TreeNode *root = new TreeNode(nums[mid]);
    if (l <= mid - 1)
      root->left = create(nums, l, mid - 1);
    if (mid + 1 <= r)
      root->right = create(nums, mid + 1, r);

    return root;
  }

  TreeNode *sortedArrayToBST(vector<int> &nums) {
    return create(nums, 0, (int)nums.size() - 1);
  }
};
```

Sorted Array to Binary Search Tree for Java

```java
class Solution {
  public TreeNode create(int[] nums, int l, int r) {
    if (l > r) return null;

    int mid = l + (r - l) / 2;
    TreeNode root = new TreeNode(nums[mid]);
    root.left = create(nums, l, mid - 1);
    root.right = create(nums, mid + 1, r);
    return root;
  }

  public TreeNode sortedArrayToBST(int[] nums) {
    return create(nums, 0, nums.length - 1);
  }
}
```

Exercise 3 - *K*-th Smallest Element in a BST

Abridged Statement:

> Given a root of a binary search tree and an integer k, return k-th smallest value of all values of the nodes in the tree.

Sample Input 1:

```
root = [3, 1, 4, null, 2]
k = 1
```

Sample Output 1:

```
1
```

Sample Input 2:

```
root = [5, 3, 6, 2, 4, null, null, 1]
k = 3
```

Sample Output 2:

```
3
```

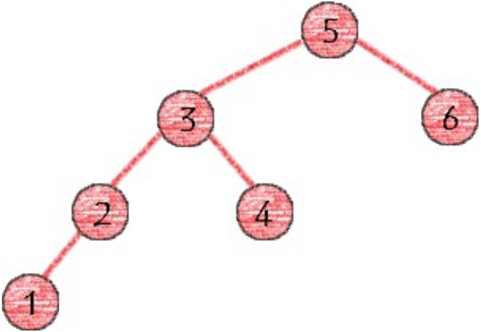
Problem is taken from [leetcode.com/problems/kth-smallest-element-in-a-bst.](https://leetcode.com/problems/kth-smallest-element-in-a-bst)

K-th Smallest Element in a BST

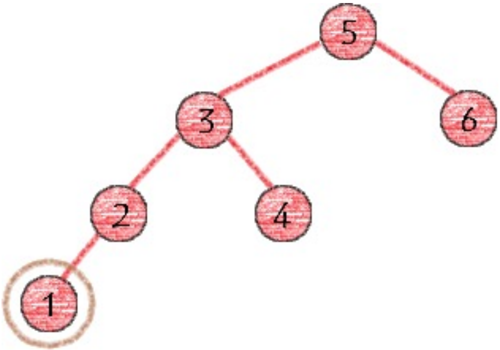
Key observations and implementations:

- Performing in-order traversal on a BST results in a sorted array.
- k-th smallest element in the BST is the k-th element of the sorted array.

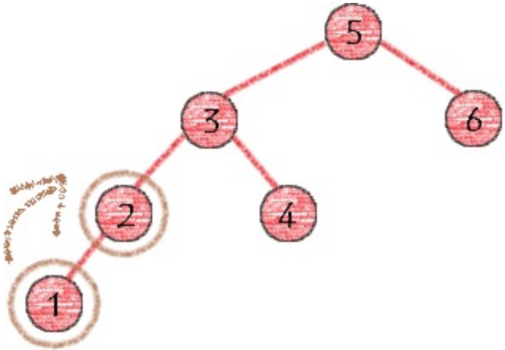
K-th Smallest Element in a BST



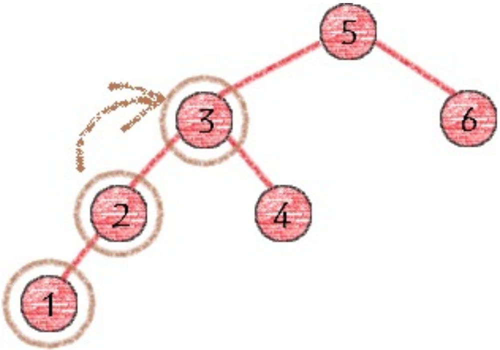
K-th Smallest Element in a BST



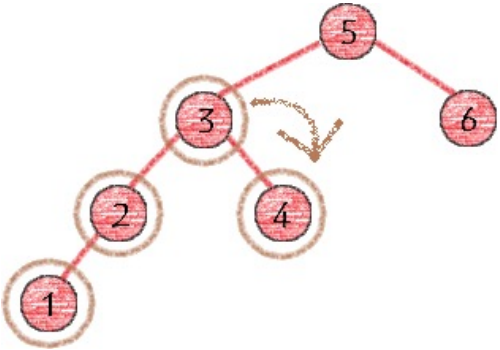
K-th Smallest Element in a BST



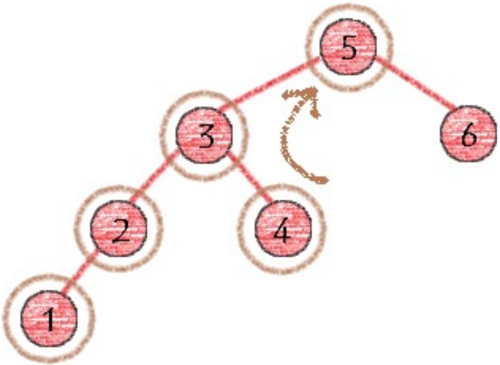
K-th Smallest Element in a BST



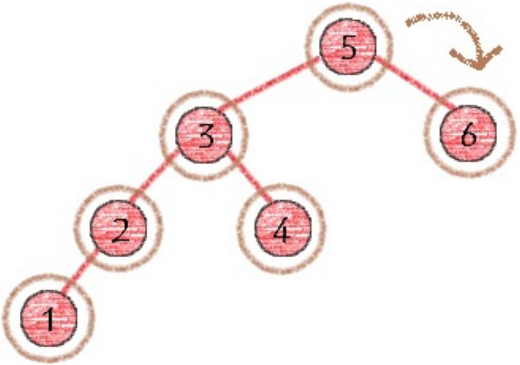
K-th Smallest Element in a BST



K-th Smallest Element in a BST



K-th Smallest Element in a BST



K-th Smallest Element in a BST for Python

```
class Solution:
    def __init__(self):
        self.ans = []

    def inorder(self, root):
        if root is None:
            return
        self.inorder(root.left)
        self.ans.append(root.val)
        self.inorder(root.right)

    def kthSmallest(self, root: Optional[TreeNode], k: int) ->
    int:
        self.inorder(root)
        return self.ans[k - 1]
```

K-th Smallest Element in a BST for C++

```
class Solution {
public:
  vector<int> ans;

  void inorder(TreeNode *root) {
    if (root == NULL)
      return;
    inorder(root->left);
    ans.push_back(root->val);
    inorder(root->right);
  }

  int kthSmallest(TreeNode *root, int k) {
    inorder(root);
    return ans[k - 1];
  }
};
```

K-th Smallest Element in a BST for Java

```java
class Solution {
  List < Integer > ans = new ArrayList < > ();

  void inorder(TreeNode root) {
    if (root == null)
      return;
    inorder(root.left);
    ans.add(root.val);
    inorder(root.right);
  }

  public int kthSmallest(TreeNode root, int k) {
    inorder(root);
    return ans.get(k - 1);
  }
}
```

Alternative Solution *K*-th Smallest Element in a BST

```
class Solution:
    def __init__(self):
        self.ans = []
    def kthSmallest(self, root: Optional[TreeNode], k: int) ->
    int:
        stack= []
        curr, count = root, 0
        while curr is not None or len(stack) > 0:
            while curr is not None:
                stack.append(curr)
                curr = curr.left
            curr = stack.pop()
            count += 1
            if count == k:
                return curr.val
            curr = curr.right
        return None
```

Alternative Solution K-th Smallest Element in a BST

Using DFS to determine the size of each subtree.

Perform a search by considering the following cases:

1. If size[left-subtree] + 1 = k, the root is the k-th smallest element.
2. Else if $k \leq$ size[left-subtree], the k-th smallest element is in the left subtree.
3. Otherwise, the k-th smallest element is in the right subtree, adjusting k to $k -$ size[left-subtree] $- 1$.

End of Session