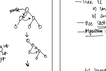
# Binary Search Tree 22

Monday, 2 October 2023 7:24 PM

1) Inorder BST to Sorted List

Idea: Execute the BST Inorder Traversal.
$O(n)$ time, $O(h)$ extra space.

```
Inorder: O(n) time
  ans = []
  def inorder(node):
      global ans
      if node:
          inorder(node.left)
          ans.append(node.data)
          inorder(node.right)
  inorder(root)
  return ans
```



2) Find largest BST subtree in a Binary Tree $O(n)$ time

→ Use a recursive function that returns a tuple (size, min, max) of the largest BST subtree rooted at the given node. If the subtree is not a BST, return (-1, -1, -1).

```
def largest(node):
    if node is None:
        return (0, +∞, -∞)
    left_size, left_min, left_max = largest(node.left)
    right_size, right_min, right_max = largest(node.right)
    if left_size == -1 or right_size == -1:
        return (-1, -1, -1)
    if node.data > left_max and node.data < right_min:
        return (left_size + right_size + 1, left_min, right_max)
    else:
        return (-1, -1, -1)
largest(root)
```

3) Check whether BST contains dead end

→ BST contains no integer keys from 0 onwards.
Basically check if there is a leaf node who when it asks that $x+1$ and $x-1$ exist in BST. (except for $x=0$, there we check just $x+1$)

Method 1

Traverse BST and check all nodes in a DFS.
Mark them, insert in separate set to avoid backtracking.
Check for every leaf node $x$, if $x-1$ and $x+1$ are present in mark set.

Improved Approach

Use one hash map.
Map the hash map of $x$ as value and check if $x-1$ and $x+1$ are present.
If one of $x$ and $x$ is absent $x+1$ and $x-1$ are present, return True.

Code

```
Check for every node, if one exists, return True.
def isDeadEnd(root):
    def check(node, mark, mini, maxi):
        if node is None:
            return False
        if node.data == 1 and node.data == 2:
            return True
        if node.data == maxi and node.data == mini:
            return True
        if node.data == 1 or node.data == 2:
            mark[node.data] = True
        left = check(node.left, mark, mini, node.data - 1)
        right = check(node.right, mark, node.data + 1, maxi)
        return left or right
    mark = {1: False, 2: False}
    return check(root, mark, 1, float('inf'))
```

4) Check if given array can represent Preorder Traversal of BST

Example: Check that given array can represent Preorder Traversal of BST

(1) 40 30 35 80 100 → Yes, this can be a valid preorder traversal of a BST.
(2) 40 30 35 80 90 → No, this cannot be a valid preorder traversal of a BST.

Method 1: To use stack (Refer to next Inorder problem above)

1) Create Inorder Array.
2) Initialize root as $-\infty$
3) Iterate through the array element $pre[i]$
   (1) If $pre[i] < root$ then return false, otherwise
   (2) Keep removing elements from stack while $pre[i] > stack[top]$
   Then check top
   Make root equal to the last popped.
   (3) Push $pre[i]$ to stack (All elements in stack are descending order)

```
def canRepresentBST(pre):
    inorder = sorted(set(pre))
    root = -float('inf')
    stack = []
    for value in pre:
        if value < root:
            return False
        while stack and stack[-1] < value:
            root = stack.pop()
        stack.append(value)
    return True
```

Time: $O(n)$

Method 2: Recursive Approach

1) Create Inorder Array.
2) Initialize root as $-\infty$
3) Iterate through the array element $pre[i]$
   (1) If $pre[i] < root$ then return false, otherwise
   (2) Keep removing elements from stack while $pre[i] > stack[top]$
   Then check top
   Make root equal to the last popped.
   (3) Push $pre[i]$ to stack (All elements in stack are descending order)

```
def canRepresentBST(pre):
    inorder = sorted(set(pre))
    root = -float('inf')
    for value in pre:
        if value < root:
            return False
        while stack and stack[-1] < value:
            root = stack.pop()
        stack.append(value)
    return True
```

5) Array vs. Subsequence: Find all possible subsequences

→ An subsequence is considered if it contains every array of binary subsequence in array.

Example: Given array [1, 2, 3] contains every array of binary subsequence in array.

$[1,2,3] \to [1,2,3] \to [1,2] \to [1,3] \to [2,3] \to [1] \to [2] \to [3]$

$2^n \to (2^3 = 8)$
$\to \{\emptyset, \{1\}, \{2\}, \{3\}, \{1,2\}, \{1,3\}, \{2,3\}, \{1,2,3\}\}$

Time: $O(2^n)$

Method: To use stack (Refer to next Inorder problem above)

1) Create Inorder Array.
2) Initialize root as $-\infty$
3) Iterate through the array element $pre[i]$
   (1) If $pre[i] < root$ then return false, otherwise
   (2) Keep removing elements from stack while $pre[i] > stack[top]$
   Then check top
   Make root equal to the last popped.
   (3) Push $pre[i]$ to stack (All elements in stack are descending order)

```
def canRepresentBST(pre):
    inorder = sorted(set(pre))
    root = -float('inf')
    for value in pre:
        if value < root:
            return False
        while stack and stack[-1] < value:
            root = stack.pop()
        stack.append(value)
    return True
```

6) Inorder Successor of a Node in BST

Idea: Find the position of BST in $O(h)$ time and $O(1)$ space.

→ If we find node's parent, we can find the inorder successor.
→ If node has right child, then inorder successor is the left most child of the right child.
→ If node has no right child, then inorder successor is the first ancestor whose left child is also an ancestor of the given node.

Steps: Compare the given node with root.
→ If node is less than root, move to the left subtree.
→ If node is greater than root, move to the right subtree.
→ If node is equal to root, stop traversing.
→ Once we find the node, check its right child.
→ If it exists, the inorder successor is the leftmost child in the right subtree.
→ If it doesn't exist, the inorder successor is the first ancestor whose left child is also an ancestor of the given node.

```
def inorderSuccessor(root, node):
    if node.right:
        return leftmost(node.right)
    else:
        ancestor = None
        while root:
            if node.data < root.data:
                ancestor = root
                root = root.left
            elif node.data > root.data:
                root = root.right
            else:
                break
        return ancestor
```

7) Check BST Back Up in a given range

→ Traverse the given binary tree from root.
For every node, check if its value lies in range.
→ If yes from root if it lies and return true that left BST validation.
→ If current node is invalid then return false, return true right child.

```
def isBST(root, left, right):
    if root is None:
        return True
    if root.data <= left or root.data >= right:
        return False
    return isBST(root.left, left, root.data) and isBST(root.right, root.data, right)
```

8) Find the median of BST in $O(n)$ time and $O(1)$ space.

→ If we find median of BST in $O(n)$ time and $O(1)$ space.
→ We can use Morris Traversal.

Steps: Create an empty array.
→ Traverse the BST using Morris Traversal, but instead of processing nodes, just count the total number of nodes.
→ Calculate the median index: If the total number of nodes is odd, the median is the middle element. If even, the median is the average of the two middle elements.

Time Complexity: $O(n)$ because we still visit each node once.
Space Complexity: $O(1)$ because we only use a few extra variables.

```
def morrisTraversal(root):
    count = 0
    current = root
    while current:
        if current.left is None:
            count += 1
            current = current.right
        else:
            pre = current.left
            while pre.right and pre.right != current:
                pre = pre.right
            if pre.right == current:
                pre.right = None
                count += 1
                current = current.right
            else:
                pre.right = current
                current = current.left
    return count
def findMedian(root):
    total = morrisTraversal(root)
    if total % 2 == 1:
        return total // 2
    else:
        return (total // 2 + (total // 2 - 1)) / 2
```

9) Convert sorted array to Balanced BST $O(n)$ time, space

Idea: Find middle element of array and make it root of tree.
Then perform check, repeat for left subarray for root's left child and right subarray for root's right child.

```
def sortedArrayToBST(arr):
    if not arr:
        return None
    mid = (len(arr) - 1)//2
    root = Node(arr[mid])
    root.left = sortedArrayToBST(arr[:mid])
    root.right = sortedArrayToBST(arr[mid+1:])
    return root
```

$O(n)$ time, $O(n)$ space (for recursive call stack)

10) Sorted Linked List to Balanced BST

Example: 1) Find middle and make it root.
2) Recursively do same for left half and right half.

$O(n \log n)$ time

```
def sortedListToBST(head):
    if not head:
        return None
    if head.next is None:
        return TreeNode(head.val)
    slow = head
    fast = head.next.next
    while fast and fast.next:
        slow = slow.next
        fast = fast.next.next
    mid = slow.next
    slow.next = None
    root = TreeNode(mid.val)
    root.left = sortedListToBST(head)
    root.right = sortedListToBST(mid.next)
    return root
```

Method 2: Approach where middle element has given root to start.
In this method, we cannot give left to start.
Idea is to insert nodes in BST in same order as it appears in list so that tree is constructed in $O(n)$ time.

→ Count no. of nodes ($n$).
→ Use this to calculate and recursively construct tree.
→ After constructed left subtree, assign the root and then right subtree.
→ Finally we construct root which assigns the left and right subtree.
→ Actually constructing we also keep moving the list head pointer to move to the next node. This eliminates pointer in each recursive call.

11) Convert Sorted Array to Balanced BST

Idea to find middle element of array and make it root of tree.
Then perform check, repeat for left subarray for root's left child and right subarray for root's right child.

Approach: 1) Find middle element of array as root.
Recursively do same for left half and right half.
→ Get middle element of left half and make it left child of root.
→ Get middle element of right half and make it right child of root.

```
from typing import List, Tuple
def sortedArrayToBST(arr):
    if not arr:
        return None
    mid = (len(arr) - 1)//2
    root = TreeNode(arr[mid])
    root.left = sortedArrayToBST(arr[:mid])
    root.right = sortedArrayToBST(arr[mid+1:])
    return root
```

$O(n)$ time, $O(n)$ space

Interestingly a Binary with each nodes and leaf nodes is empty. However that nodes, from root, and find mid element of sorted array. Create root node with previously found middle node and set left child to single node left child & right child & recursively find other left right child with single node right child.
Image root node with space.
Left subtree child of designed nodes to middle element on left side of sorted array.
& designed nodes has right child except to both the sides.
Return root value.

12) Convert DLL to Balanced BST

13) Binary Tree to BST

→ Conversion in place is many hard if keeps original structure of tree.

Inorder Solution

→ Create binary array [] that stores inorder traversal. $O(n)$ time.
→ Sort the binary array. $O(n)$ Binary search tree for finding left and right child. $O(\log n)$ time for binary search tree.
→ Replace the binary traversal of tree and copy sorted array elements to tree. $O(n)$ time.

```
def binaryTreeToBST(root, root):
    def inorder(root, A):
        if root:
            inorder(root.left, A)
            A.append(root.data)
            inorder(root.right, A)
    def assignBST(root, A):
        if root is None:
            return None
        assignBST(root.left, A)
        root.data = A[0]
        A.pop(0)
        assignBST(root.right, A)
    if root is None:
        return None
    arr = []
    inorder(root, arr)
    arr.sort()
    assignBST(root, arr)
    return root
```

$O(n)$ space

14) Balance a binary search tree.

1. Traverse BST in inorder and store it in sorted array. $O(n)$ time
2. Find middle element of sorted array and make it root of balanced BST.
3. Build a BST from above created array recursively.
   This step takes $O(n)$ time, as we traverse every element exactly once and process an element takes $O(1)$ time.

$O(n)$ time (traversing nodes, inorder traversal of balanced tree)
$O(n)$ space (storing nodes in array)

```
def balanceBinarySearchTree(root, root):
    def inorder(root, A):
        if root:
            inorder(root.left, A)
            A.append(root.data)
            inorder(root.right, A)
    def BST(arr):
        if not arr:
            return None
        mid = (len(arr) - 1)//2
        root = TreeNode(arr[mid])
        root.left = BST(arr[:mid])
        root.right = BST(arr[mid+1:])
        return root
    A = []
    inorder(root, A)
    return BST(A)
```

15) Convert Linked List to Balanced BST

Example: 1) Find middle and make it root.
2) Recursively do same for left half and right half.

$O(n \log n)$ time

```
def sortedListToBST(head):
    if not head:
        return None
    if head.next is None:
        return TreeNode(head.val)
    slow = head
    fast = head.next.next
    while fast and fast.next:
        slow = slow.next
        fast = fast.next.next
    mid = slow.next
    slow.next = None
    root = TreeNode(mid.val)
    root.left = sortedListToBST(head)
    root.right = sortedListToBST(mid.next)
    return root
```

Method 2: Approach where middle element has given root to start.
In this method, we cannot give left to start.
Idea is to insert nodes in BST in same order as it appears in list so that tree is constructed in $O(n)$ time.

→ Count no. of nodes ($n$).
→ Use this to calculate and recursively construct tree.
→ After constructed left subtree, assign the root and then right subtree.
→ Finally we construct root which assigns the left and right subtree.
→ Actually constructing we also keep moving the list head pointer to move to the next node. This eliminates pointer in each recursive call.

16) Convert a Binary Tree to Doubly Linked List $O(n)$ time

→ Left and right pointers to nodes are used as a previous and next pointers respectively in the converted DLL.
→ Order of nodes in list must be same as inorder traversal of binary tree.
→ First node in inorder must be head node.

Steps:
→ Recursively convert left subtree to a DLL and connect it to inorder.
→ Make a circular linked list of nodes of the tree and make left and right point to pointers.
→ Convert the left subtree into a single node.
→ Do the same for the right subtree.
→ Connect both head to list.

```
def convertToDLL(root):
    def convert(node, left=None, right=None):
        if node is None:
            return None
        if left is None:
            left = node
        if right is None:
            right = node
        left.right = node.left = left
        right.left = node.right = right
        return left, right
    if root:
        left, _ = convert(root.left)
        _, right = convert(root.right)
        head = root
        head.left = head.right = head
        if left:
            head = head.left = left
        if right:
            head = head.right = right
        return head
    return None
```

17) Convert a Binary Tree to a Circular Doubly Linked List

→ Left and right pointers to nodes are used as a previous and next pointers respectively in the converted CDLL.
→ Order of nodes in list must be same as inorder traversal of binary tree.
→ First node in inorder must be head node.

Steps:
→ Recursively convert left subtree to a CDLL and connect it to inorder.
→ Make a circular linked list of nodes of the tree and make left and right point to pointers.
→ Convert the right subtree into a single node.
→ Do the same for the right subtree.
→ Connect both head to list.

```
def convertToCDLL(root):
    def convert(node, left=None, right=None):
        if node is None:
            return None, None
        if left is None:
            left = node
        if right is None:
            right = node
        left.right = node.left = left
        right.left = node.right = right
        return left, right
    if root:
        left, _ = convert(root.left)
        _, right = convert(root.right)
        head = root
        head.left = head.right = head
        if left:
            head = head.left = left
        if right:
            head = head.right = right
        return head
    return None
```

18) Convert a Binary Tree to Doubly Linked List $O(n)$ time

→ Left and right pointers to nodes are used as a previous and next pointers respectively in the converted DLL.
→ Order of nodes in list must be same as inorder traversal of binary tree.
→ First node in inorder must be head node.

Steps:
→ Recursively convert left subtree to a DLL and connect it to inorder.
→ Make a circular linked list of nodes of the tree and make left and right point to pointers.
→ Convert the right subtree into a single node.
→ Do the same for the right subtree.
→ Connect both head to list.

```
def convertToDLL(root):
    def convert(node, left=None, right=None):
        if node is None:
            return None, None
        if left is None:
            left = node
        if right is None:
            right = node
        left.right = node.left = left
        right.left = node.right = right
        return left, right
    if root:
        left, _ = convert(root.left)
        _, right = convert(root.right)
        head = root
        head.left = head.right = head
        if left:
            head = head.left = left
        if right:
            head = head.right = right
        return head
    return None
```

19) Convert Binary Tree to a DLL by Using Inorder Traversal

→ Idea is to do inorder traversal of binary tree.
→ Keep track of the last visited node, from prev to current node and point the prev and point next to current node.

$O(n)$ time, $O(n)$ space

```
def BTToDLL(root):
    global prev
    def inorder(root):
        global prev
        if root:
            inorder(root.left)
            if prev:
                prev.right = root
                root.left = prev
            prev = root
            inorder(root.right)
    prev = None
    inorder(root)
    return prev
```

20) Convert a Binary Tree to DLL by Using Inorder Traversal

→ Idea is to do inorder traversal of binary tree. Keep track of previously visited node in a variable prev (global variable).
→ For every visited node, make it next to prev and set prev to the current node.

$O(n)$ time, $O(n)$ space (call stack)

```
def BTToDLL(root):
    global head, prev
    def inorder(root):
        global head, prev
        if root is None:
            return None
        if root.left is None:
            head = root
        else:
            inorder(root.left)
        if prev:
            root.left = prev
            prev.right = root
        prev = root
        inorder(root.right)
    inorder(root)
    return head
```

21) Convert a Binary Tree to DLL by Using Inorder Traversal

→ Idea is to do inorder traversal of binary tree. Keep track of previously visited node in a variable prev (global variable).
→ For every visited node, make it next to prev and set prev to the current node.

$O(n)$ time, $O(n)$ space

```
def BTToDLL(root):
    global head, prev
    def inorder(root):
        global head, prev
        if root is None:
            return None
        if root.left is None:
            head = root
        else:
            inorder(root.left)
        if prev:
            root.left = prev
            prev.right = root
        prev = root
        inorder(root.right)
    inorder(root)
    return head
```