

Lab-XII

Hasan Amin

(374866)

CS-250

Data Structures and Algorithms

School of Natural Sciences

Contents

[Task 1 and 2: 3](#_Toc153470671)

[Task 3: 8](#_Toc153470672)

[Task 4: 9](#_Toc153470673)

[Task 5 13](#_Toc153470674)

# Task 1 and 2:

|  |
| --- |
| **Code** |
| class Tree:      def \_\_init\_\_(self, datum, Lchild=None, Rchild=None):          self.Lchild = Lchild          self.Rchild = Rchild          self.datum = datum      def height(self):          if self is None:              return 0          left\_height = self.Lchild.height() if self.Lchild else 0          right\_height = self.Rchild.height() if self.Rchild else 0          return 1 + max(left\_height, right\_height)      def tree\_size(self, node=True):          if node is True: node = self          if node is None:              return 0          return 1 + self.tree\_size(node.Lchild) + self.tree\_size(node.Rchild)      def add\_item(self, item):          if self is None:              self = Tree(item)          elif self.Lchild is None:              self.Lchild = Tree(item)          elif self.Rchild is None:              self.Rchild = Tree(item)          else:              right = self.Rchild.tree\_size()              left = self.Lchild.tree\_size()              if left <= right:                  self.Lchild.add\_item(item)              else:                  self.Rchild.add\_item(item)      def post\_order\_traversal(self):          result = []          if self:              if self.Lchild:                  result.extend(self.Lchild.post\_order\_traversal())                  # print(result, "l")              if self.Rchild:                  result.extend(self.Rchild.post\_order\_traversal())                  # print(result, "r")              result.append(                  self.datum              )  # left node --> right node --> root node recursion          return result      def pre\_order\_traversal(self):          result = []          if self:              result.append(                  self.datum              )  # root node --> left node --> right node recursion              if self.Lchild:                  result.extend(self.Lchild.pre\_order\_traversal())                  # print(result, "l")              if self.Rchild:                  result.extend(self.Rchild.pre\_order\_traversal())                  # print(result, "r")          return result      def in\_order\_traversal(self):          result = []          if self:              if self.Lchild:                  result.extend(self.Lchild.in\_order\_traversal())                  # print(result, "l")              result.append(                  self.datum              )  # left node --> root node --> right node recursion              if self.Rchild:                  result.extend(self.Rchild.in\_order\_traversal())                  # print(result, "r")          return result      def deleteDeepest(self, d\_node):          q = [self]          while q:              temp = q.pop(0)              if temp is d\_node:                  temp = None                  return              if temp.Rchild:                  if temp.Rchild is d\_node:                      temp.Rchild = None                      return                  else:                      q.append(temp.Rchild)              if temp.Lchild:                  if temp.Lchild is d\_node:                      temp.Lchild = None                      return                  else:                      q.append(temp.Lchild)      def deletion(self, key):          if self is None:              return None          if self.Lchild is None and self.Rchild is None:              if self.datum == key:                  return None              else:                  return self          key\_node = None          q = [self]          temp = None          while q:              temp = q.pop(0)              if temp.datum == key:                  key\_node = temp              if temp.Lchild:                  q.append(temp.Lchild)              if temp.Rchild:                  q.append(temp.Rchild)          if key\_node:              x = temp.datum              self.deleteDeepest(temp)              key\_node.datum = x          return self      def search(self, value):          if not self:              return False          if self.datum == value:              return True          found = False          if self.Lchild:              found = self.Lchild.search(value)          if not found and self.Rchild:              found = self.Rchild.search(value)          return found      def \_\_str\_\_(self):          if (self.Lchild is None) and (self.Rchild is None):              return f"{self.datum}"          return f"{self.datum} (L: {self.Lchild} & R: {self.Rchild})"      def \_\_iter\_\_(self):          self.\_stack = []          self.\_current = self          return self      def \_\_next\_\_(self):          while self.\_current or self.\_stack:              if self.\_current:                  self.\_stack.append(self.\_current)                  self.\_current = self.\_current.Lchild              else:                  node = self.\_stack.pop()                  current\_node = node                  self.\_current = node.Rchild                  # print(current\_node.datum)                  return current\_node          raise StopIteration |

|  |
| --- |
| **Testing Code** |
| from tree import Tree  def main():      tree = Tree('pie')      tree.add\_item('cake')      tree.add\_item('cookie')      for item in tree:          print(f'{item.datum}, {item.height()}')      #Should print:      #cake: 0      #pie: 1      #cookie: 0      tree.add\_item('cupcake')      for item in tree:          print(f'{item.datum}, {item.height()}')      #Should print:      #cupcake: 0      #cake: 1      #pie: 2      #cookie: 0      #this constructs the tree shown in the exercise      other\_tree = Tree(1)      for i in [2, 3, 4, 6, 5, 7]:          other\_tree.add\_item(i)      print(list(map(lambda d: d.datum, other\_tree)))      print(f"Post Order:{other\_tree.post\_order\_traversal()}")      # print(other\_tree)      #Should print: [4, 2, 5, 1, 6, 3, 7]  if \_\_name\_\_ == '\_\_main\_\_':      main() |

|  |
| --- |
| **Output** |
| A screenshot of a computer  Description automatically generated |

# Task 3:

|  |
| --- |
| **Code** |
| from tree import Tree  def postfixToTree(string:str):      bucket=string.split(' ')      stack=[]      for b in bucket:          if b.isalnum():              stack.append(Tree(b))          elif b in '\*+/-':              right=stack.pop()              left=stack.pop()              stack.append(Tree(b,Lchild=left,Rchild=right))          else:              raise ValueError("Invaid Expression")      if stack:          final\_tree=stack.pop()          return final\_tree    print(postfixToTree("B B \* A C 4 \* \* -"))    print(postfixToTree("91 95 + 15 + 19 + 4 \*")) |

|  |
| --- |
| **Output** |
|  |



# Task 4:

|  |
| --- |
| **Code** |
| class TreeNode:      def \_\_init\_\_(self, key):          self.key = key          self.left = None          self.right = None  class BST:      def \_\_init\_\_(self):          self.root = None      def insert(self, key):          self.root = self.\_\_insert\_recursive(self.root, key)      def \_\_insert\_recursive(self, root, key):          if root is None:              return TreeNode(key)          if key < root.key:              root.left = self.\_\_insert\_recursive(root.left, key)          else:              root.right = self.\_\_insert\_recursive(root.right, key)          return root      def delete(self, key):          self.root = self.\_\_delete\_recursive(self.root, key)      def \_\_delete\_recursive(self, root, key):          if root is None:              return root          if key < root.key:              root.left = self.\_\_delete\_recursive(root.left, key)          elif key > root.key:              root.right = self.\_\_delete\_recursive(root.right, key)          else:              if root.left is None:                  return root.right              elif root.right is None:                  return root.left              temp = self.\_\_min\_value\_node(root.right)              root.key = temp.key              root.right = self.\_\_delete\_recursive(root.right, temp.key)          return root      def search(self, key):          return self.\_\_search\_recursive(self.root, key) is not None      def \_\_search\_recursive(self, root, key):          if root is None or root.key == key:              return root          if key < root.key:              return self.\_\_search\_recursive(root.left, key)          return self.\_\_search\_recursive(root.right, key)      def sort(self):          result = []          self.\_\_inorder\_recursive(self.root, result)          return result      def \_\_inorder\_recursive(self, root, result):          if root:              self.\_\_inorder\_recursive(root.left, result)              result.append(root.key)              self.\_\_inorder\_recursive(root.right, result)      def display(self):          self.\_\_display\_recursive(self.root)      def \_\_display\_recursive(self, root):          if root:              self.\_\_display\_recursive(root.left)              print(root.key, end=' ')              self.\_\_display\_recursive(root.right)      def \_\_min\_value\_node(self, node):          current = node          while current.left:              current = current.left          return current  def main():      bst = BST()      while True:          print("\nBinary Search Tree Operations:")          print("1. Create a BST")          print("2. Insert a new element")          print("3. Delete an element")          print("4. Search for an element")          print("5. Sort the BST")          print("6. Display the BST")          print("7. Exit")          choice = int(input("Enter your choice: "))          if choice == 1:              elements = list(map(int, input("Enter space-separated elements: ").split()))              for elem in elements:                  bst.insert(elem)              print("BST created.")          elif choice == 2:              key = int(input("Enter element to insert: "))              bst.insert(key)              print(f"{key} inserted.")          elif choice == 3:              key = int(input("Enter element to delete: "))              if bst.search(key):                  bst.delete(key)                  print(f"{key} deleted.")              else:                  print(f"{key} not found in BST.")          elif choice == 4:              key = int(input("Enter element to search: "))              if bst.search(key):                  print(f"{key} found in BST.")              else:                  print(f"{key} not found in BST.")          elif choice == 5:              sorted\_elements = bst.sort()              print("Sorted BST:", sorted\_elements)          elif choice == 6:              print("BST elements:")              bst.display()          elif choice == 7:              print("Exiting the program.")              break          else:              print("Invalid choice. Please enter a valid option.")  if \_\_name\_\_ == "\_\_main\_\_":      main() |

|  |
| --- |
| **Output** |
|  |

# Task 5

|  |
| --- |
| **Code** |
| class Node:      def \_\_init\_\_(self, val):          self.val = val          self.left = None          self.right = None  def build\_tree(arr):      if not arr:          return None      nodes = [None if val == 'n' else Node(val) for val in arr]      root = nodes[0]      for i in range(len(arr)):          if nodes[i]:              left\_index = 2 \* i + 1 # left ndex at 2n+1 position and right at 2n+2              right\_index = 2 \* i + 2              if left\_index < len(arr):                  nodes[i].left = nodes[left\_index]              if right\_index < len(arr):                  nodes[i].right = nodes[right\_index]              if i != 0 and not nodes[(i - 1) // 2]:                  raise ValueError("Tree cannot be built")      return root  def print\_tree(root):      if not root:          return      queue = [root]      while queue:          node = queue.pop(0)          if node:              print(node.val, end=" ")              queue.append(node.left)              queue.append(node.right)          else:              print("n", end=" ")      print()  # Test cases  test\_cases = [      [],      ['n', 'n', 'n'],      [55, 12, 71],      [55, 12, 'n', 4],      [55, 12, 'n', 4, 'n', 'n', 'n', 'n', 8, 'n', 'n', 'n', 'n', 'n', 'n', 'n', 'n', 6, 'n'],      [55, 12, 'n', 'n', 'n', 'n', 4, 'n', 8, 'n', 'n', 'n', 'n', 'n', 'n', 'n', 'n', 6, 'n']  ]  for arr in test\_cases:      try:          tree\_root = build\_tree(arr)          print\_tree(tree\_root)      except ValueError as e:          print(e) |

|  |
| --- |
| **Output** |
|  |

