February 29, 2024

Department of Computer Science and Engineering
Islamic University of Technology (IUT)
A subsidiary organ of OIC

Algorithm Engineering Lab 02

Azmayen Fayek Sabil, 190042122

Contents

1 Task 1

1.1 Question

Implement the problem for BST, defined in the class material (Slide) with AVL tree, so that $O(h)$ is always equal to $O(\log n)$ where h is the height of the tree and n is the number of nodes of a tree.

1.2 Implementation

```
class TreeNode:
    def __init__(self, key):
        self.key = key
        self.left = None
        self.right = None
        self.height = 1
        self.count = 1  # Number of nodes in the subtree rooted at this node

def getHeight(node):
    if not node:
        return 0
    return node.height

def printTree(root, level=0, prefix="Root: "):
    if root is not None:
        print(" " * (level * 4) + prefix + str(root.key))
        if root.left is not None or root.right is not None:
            printTree(root.left, level + 1, "L--- ")
            printTree(root.right, level + 1, "R--- ")

def getBalance(node):
    if not node:
        return 0
    return getHeight(node.left) - getHeight(node.right)

def updateHeight(node):
    if not node:
        return 0
    node.height = 1 + max(getHeight(node.left), getHeight(node.right))
    node.count = 1 + getCount(node.left) + getCount(node.right)
    return node.height

def rightRotate(y):
    x = y.left
    T2 = x.right

    x.right = y
    y.left = T2

    updateHeight(y)
    updateHeight(x)

    return x

def leftRotate(x):
    y = x.right
    T2 = y.left

```

```python
        y.left = x
        x.right = T2

        updateHeight(x)
        updateHeight(y)

        return y

    def insert(root, key):
        if not root:
            return TreeNode(key)

        if key < root.key:
            root.left = insert(root.left, key)
        elif key > root.key:
            root.right = insert(root.right, key)
        else:
            root.count += 1  # Duplicate keys are allowed

        updateHeight(root)

        balance = getBalance(root)

        if balance > 1 and key < root.left.key:
            return rightRotate(root)

        if balance < -1 and key > root.right.key:
            return leftRotate(root)

        if balance > 1 and key > root.left.key:
            root.left = leftRotate(root.left)
            return rightRotate(root)

        if balance < -1 and key < root.right.key:
            root.right = rightRotate(root.right)
            return leftRotate(root)

        return root

    def getCount(node):
        if not node:
            return 0
        return node.count

    def inOrderTraversal(root, inorder_list):
        if root:
            inOrderTraversal(root.left, inorder_list)
            inorder_list.append(root.key)
            inOrderTraversal(root.right, inorder_list)

    def generateAVLTree(keys):
        root = None

        for key in keys:
            root = insert(root, key)

        return root

    # Example usage:
    keys = [4, 6, 7, 5, 1, 9]
    k = 3
```

```python

    root = generateAVLTree(keys)

    # Print the inorder traversal list
    inorder_list = []
    inOrderTraversal(root, inorder_list)
    print("Inorder Traversal:", inorder_list)

    # Get the sum of the first k-1 values
    sum_of_values = sum(inorder_list[:k])
    print("Sum of values that are less than or equal to", inorder_list[k-1],
    ":", sum_of_values)

    # Print the AVL tree structure
    print("AVL Tree Structure:")
    printTree(root)
```

1.3 Output

```
    INPUT: keys = [4, 6, 7, 5, 1, 9]


    Inorder Traversal: [1, 4, 5, 6, 7, 9]
    Sum of values that are less than or equal to 5 : 10
    AVL Tree Structure:
    Root: 6
        L--- 4
            L--- 1
            R--- 5
        R--- 7
            R--- 9
```

1.4 Explanation

The code implements an AVL tree, a self-balancing binary search tree, to address a problem related to Binary Search Trees (BST). I copied the implementation of AVL tree from `GeeksforGeeks`. Now Lets go through the explanation of the code itself;
The `TreeNode` class defines the structure of a node, including key value, left and right child pointers, height for balancing, and a count field for handling duplicate keys. Functions for calculating height, balancing factors, and performing rotations are included. The insertion function ensures the AVL tree remains balanced through rotations, aiming to maintain a logarithmic time complexity for search, insertion, and deletion operations. The example usage demonstrates generating an AVL tree from a list of keys, performing an in-order traversal, and calculating the sum of values smaller than or equal to the kth smallest element, thereby addressing the problem statement related to ensuring $O(h)$ is always $O(\log n)$ in AVL trees.

1.5 Complexity

This code implements an AVL tree that ensures $O(h)$ is always equal to $O(\log n)$ where h is the height of the tree and n is the number of nodes. The tree is constructed from a list of keys, and various operations such as in-order traversal and printing the tree structure are demonstrated.

2 Task 2

2.1 Question

Given an array `arr[]` of size N where each element denotes a pair in the form (price, weight) denoting the price and weight of each item. Given Q queries of the form [X, Y] denoting the price range. The task is to find the element with the highest weight within a given price range for each query.

2.2 Implementation

```python
import math

# Function to get mid
def getMid(start, end):
    return start + (end - start) // 2

# Function to fill segment tree
def fillSegmentTree(arr):
    arr.sort(key=lambda x: x[0])
    n = len(arr)
    maxHeight = math.ceil(math.log(n, 2))
    maxSize = 2 * (2 ** maxHeight) - 1
    segmentTree = [0] * maxSize
    fillSegmentTreeUtil(segmentTree, arr, 0, n - 1, 0)
    return segmentTree

# Function to utilise the segment tree
def fillSegmentTreeUtil(segmentTree, arr, start, end, currNode):
    if start == end:
        segmentTree[currNode] = arr[start][1]
        return segmentTree[currNode]
    mid = getMid(start, end)
    segmentTree[currNode] = max(fillSegmentTreeUtil(segmentTree, arr, start, mid, currNode * 2 + 1),
                                fillSegmentTreeUtil(segmentTree, arr, mid + 1, end, currNode * 2 + 2))
    return segmentTree[currNode]

# Function to find the maximum rating
def findMaxRating(arr, query, segmentTree):
    n = len(arr)
    return findMaxRatingUtil(segmentTree, arr, 0, n - 1, query[0], query[1], 0)

# Function to utilise the maxRating function
def findMaxRatingUtil(segmentTree, arr, start, end, qStart, qEnd, currNode):
    if qStart <= arr[start][0] and qEnd >= arr[end][0]:
        return segmentTree[currNode]
    if qStart > arr[end][0] or qEnd < arr[start][0]:
        return -1
    mid = getMid(start, end)
    return max(findMaxRatingUtil(segmentTree, arr, start, mid, qStart, qEnd, currNode * 2 + 1),
               findMaxRatingUtil(segmentTree, arr, mid + 1, end, qStart, qEnd, currNode * 2 + 2))

# Driver code
if __name__ == '__main__':
```

```
    arr = [[1000, 300],
           [1100, 400],
           [1300, 200],
           [1700, 500],
           [2000, 600]]
    segmentTree = fillSegmentTree(arr)
    queries = [[1000, 1400],
               [1700, 1900],
               [0, 3000]]
    for query in queries:
        print(findMaxRating(arr, query, segmentTree))
```

2.3 Output

```
400
500
600
```

2.4 Explanation

The code tackles a problem involving an array `arr[]` that represents pairs of (price, weight) for items. The objective is to find the element with the highest weight within a specified price range for each query. We used a segment tree data structure to efficiently answer these queries. The `fillSegmentTree` function is used for constructing the segment tree. It sorts the input array based on the price values. The `findMaxRating` function alo uses the constructed segment tree to find the maximum weight within a given price range specified by the queries. The code then demonstrates the functionality with a sample array and queries. The output, namely 400, 500, and 600 for the respective queries, signifies the highest weights within the specified price ranges.

Let's go through the process:

1. **Sorting by Price:** The initial step involves sorting the input array `arr[]` based on the price values.

2. **Segment Tree Construction:** The `fillSegmentTree` function constructs the segment tree. It recursively divides the array into segments, determining the maximum weight within each segment. This information is stored in the segment tree nodes, facilitating efficient query processing.

3. **Query Processing:** The `findMaxRating` function utilizes the constructed segment tree to answer queries efficiently. For each query [X, Y] (representing a price range), the function traverses the segment tree based on the specified range. It identifies the maximum weight within the given price range.

4. **Output:** The code executes a set of sample queries on the provided array, producing the output: 400, 500, and 600. Each output corresponds to the highest weight within the specified price range for the respective queries.

In summary, the code employs a segment tree to efficiently find the maximum weight within specified price ranges. The sorting step enhances the overall efficiency, and the output reflects successful query processing.

3 Task 3

3.1 Question

Given a binary 2D matrix, find the number of islands. A group of connected 1s forms an island.

3.2 Implementation

```python
def num_islands(mat):
    if not mat:
        return 0

    rows, cols = len(mat), len(mat[0])
    visited = [[False] * cols for _ in range(rows)]

    def is_valid(i, j):
        return 0 <= i < rows and 0 <= j < cols and mat[i][j] == 1 and not visited[i][j]

    def dfs(i, j):
        visited[i][j] = True
        directions = [(1, 0), (-1, 0), (0, 1), (0, -1), (1, 1), (-1, -1), (1, -1), (-1, 1)]

        for dir_i, dir_j in directions:
            new_i, new_j = i + dir_i, j + dir_j
            if is_valid(new_i, new_j):
                dfs(new_i, new_j)

    island_count = 0
    for i in range(rows):
        for j in range(cols):
            if mat[i][j] == 1 and not visited[i][j]:
                island_count += 1
                dfs(i, j)

    return island_count

# Example usage:
matrix = [
    [1, 1, 0, 0, 0],
    [0, 1, 0, 0, 1],
    [1, 0, 0, 1, 1],
    [0, 0, 0, 0, 0],
    [1, 0, 1, 0, 0]
]

result = num_islands(matrix)
print("Number of islands:", result)
```

3.3 Output

```
    Number of islands: 4
```

3.4 Explanation

The code solves the problem of finding the number of islands in a binary 2D matrix. An island is defined as a group of connected 1s. The code uses a depth-first search (DFS) approach to traverse the matrix, marking visited elements and counting the number of islands.

The `num_islands` function initializes a matrix of boolean values to keep track of visited elements. The `is_valid` function checks if a given position is within the matrix boundaries, has a value of 1, and has not been visited. The `dfs` function performs a depth-first search from a given position, marking visited elements.

The main loop iterates through each element in the matrix. If an unvisited 1 is encountered, the `dfs` function is called to explore and mark all connected 1s, incrementing the island count.

The example usage demonstrates the functionality on a sample matrix, and the result is printed, indicating the number of islands present in the given matrix.

4 Task 4

4.1 Question

The right view of a Binary Tree is a set of nodes visible when the tree is visited from the right side. Given a Binary Tree, print the right view of it.

4.2 Code

```
from collections import deque

class TreeNode:
    def __init__(self, val):
        self.val = val
        self.left = None
        self.right = None

def build_tree(edges):
    if not edges:
        return None

    adjacency_list = {}
    for edge in edges:
        x, y = edge
        if x not in adjacency_list:
            adjacency_list[x] = []
        if y not in adjacency_list:
            adjacency_list[y] = []
        adjacency_list[x].append(y)

    root = TreeNode(1)
    queue = deque([root])

    while queue:
        node = queue.popleft()
        neighbors = adjacency_list.get(node.val, [])

        for neighbor in neighbors:
            child = TreeNode(neighbor)
            setattr(node, 'left' if not node.left else 'right', child)
            queue.append(child)

    return root

def right_view_bfs(root):
    if not root:
        return []

    result = []
    queue = deque([(root, 0)])

    while queue:
        level_size = len(queue)
        for i in range(level_size):
            node, level = queue.popleft()
            if i == level_size - 1:
                result.append(node.val)

            if node.left:
```

```
                queue.append((node.left, level + 1))
            if node.right:
                queue.append((node.right, level + 1))

    return result

# INPUT
edges = [(1, 2), (1, 3), (2, 4), (2, 5), (3, 6), (3, 7), (6, 8)]
tree = build_tree(edges)
result = right_view_bfs(tree)
print("Right view of the tree:", result)


edges2 = [(1, 7), (7, 8)]
tree2 = build_tree(edges2)
result2 = right_view_bfs(tree2)
print("Right view of the tree:", result2)
```

4.3 Output

```
    Right view of the tree: [1, 3, 7, 8]
    Right view of the tree: [1, 7, 8]
```

4.4 Explanation

The code constructs a binary tree and prints its right view using a breadth-first search (BFS) approach. Here's a breakdown of the code:

1. **Building the Tree:** The `build_tree` function takes a list of edges representing the connections between nodes and constructs a binary tree. It uses a queue to iteratively build the tree based on the provided edges.

2. **Right View Calculation (BFS):** The `right_view_bfs` function calculates the right view of the binary tree using BFS. It traverses the tree level by level, and for each level, it appends the last node (rightmost node) to the result. This ensures that only the rightmost node at each level is considered for the right view.

The output demonstrates the right view of the given binary trees.