DAA - Homework 5

Adnan Jakati, Praneeth Kumar Thummalapalli, Sai Nishanth Mettu

November 2023

1 a) Question

a) Binary Search Tree - After Every 4 Inserts.

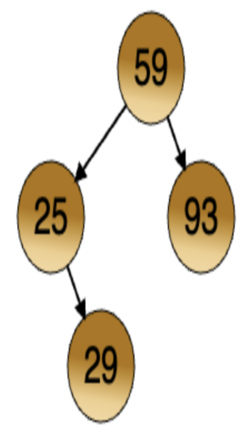

Figure 1: First 4 Inserts

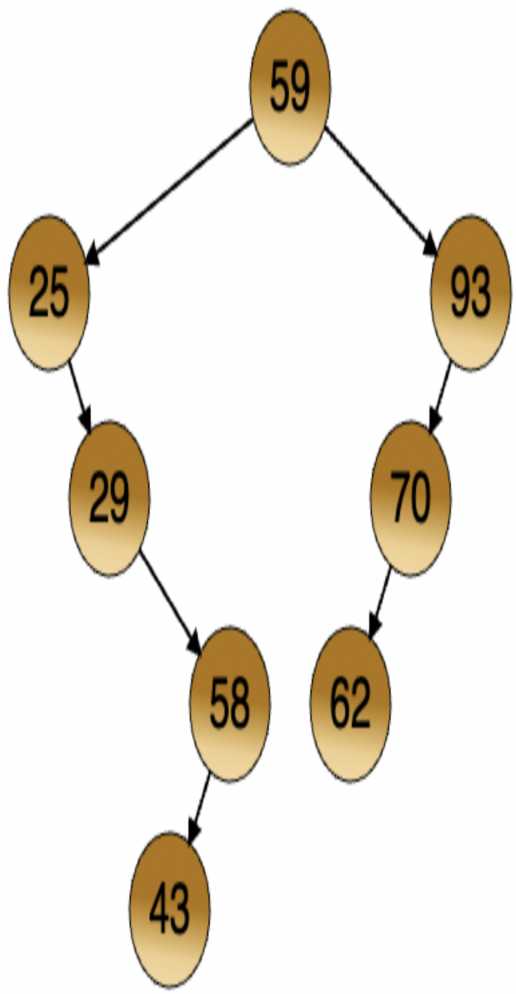

Figure 2: First 8 Inserts

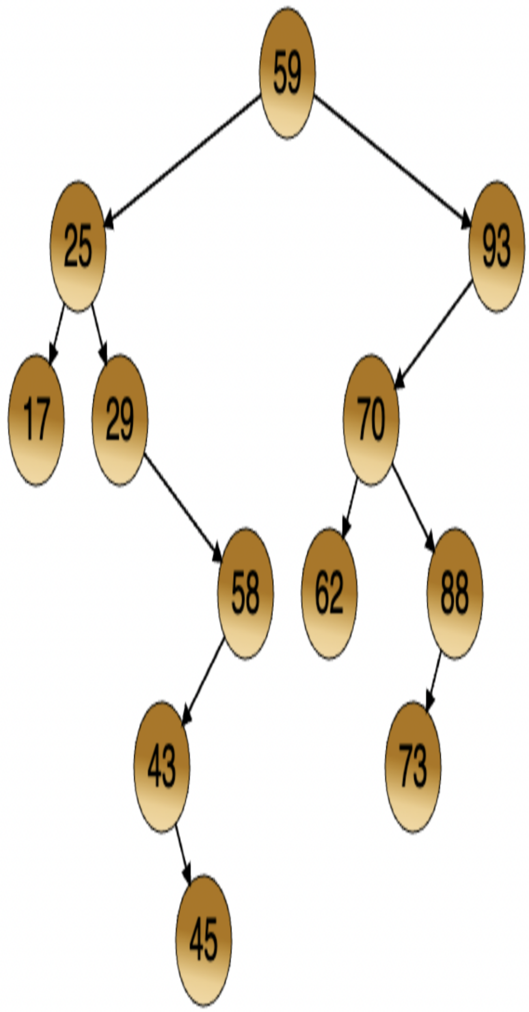

Figure 3: First 12 Inserts

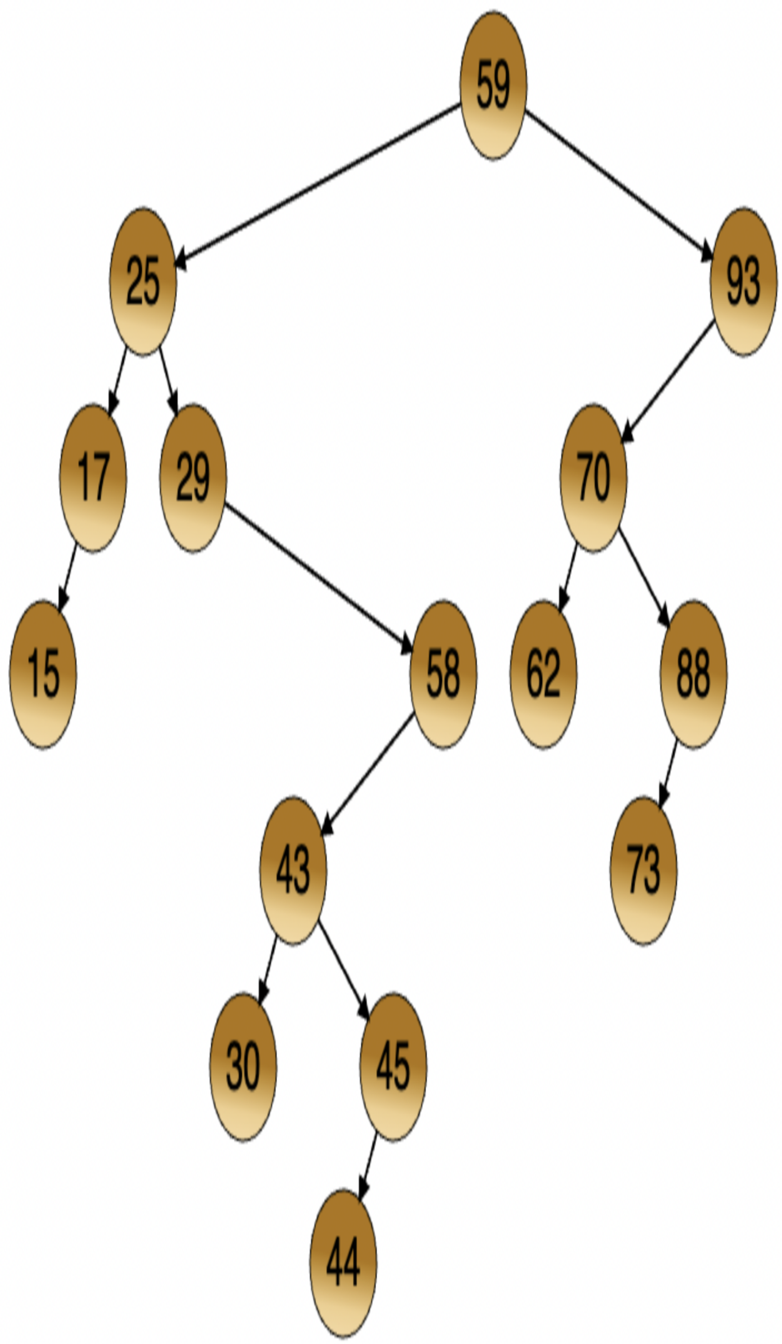

Figure 4: Final BST

1 b) Question

Red and Black Tree- After Every 4 Inserts.

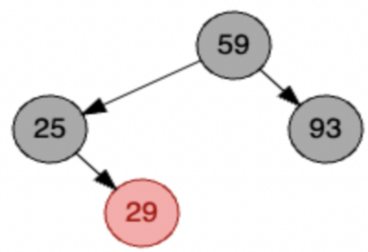

Figure 1: First 4 Inserts

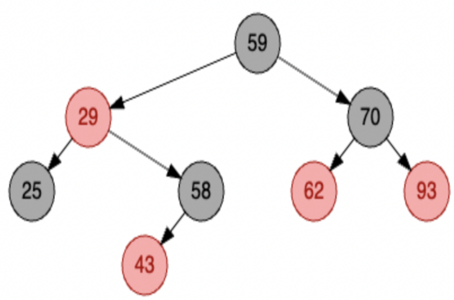

Figure 2: First 8 Inserts

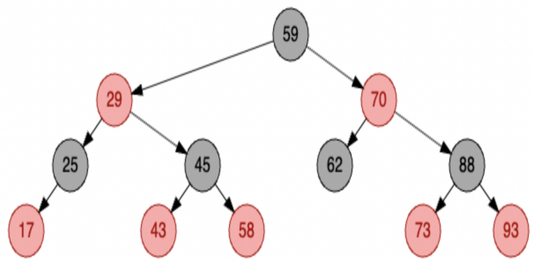

Figure 3: First 12 Inserts

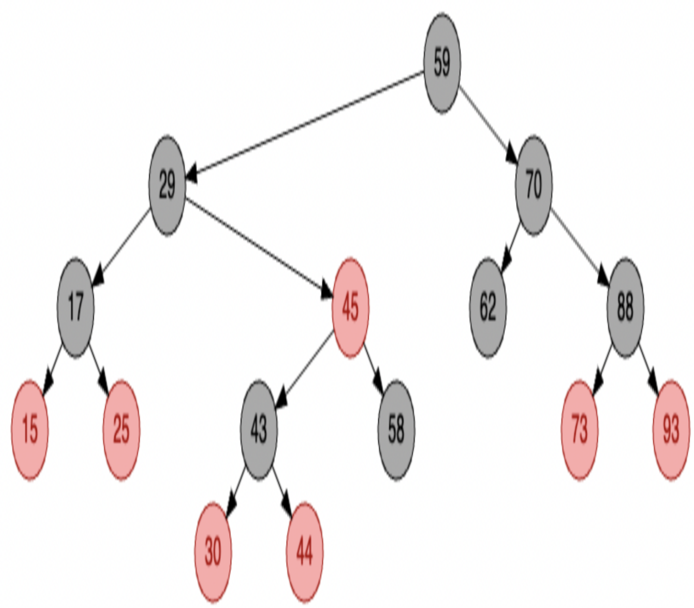

Figure 4: Final Tree

Question 2) Solution

Deletion is not commutative as it will result in a different structure of the tree. Below is the example.

```
+------------------+--------+--------+-----+
|                  |   A    |        |     |
|                  |  / \   |   C    | C   |
| Delete A, then B | B   D  |  / \   |  \  |
|                  |    /   | B   D  |   D |
|                  |   C    |        |     |
+------------------+--------+--------+-----+
|                  |   A    | A      |     |
|                  |  / \   |  \     |   D |
| Delete B, then A | B   D  |   D    |  /  |
|                  |    /   |  /     | C   |
|                  |   C    | C      |     |
+------------------+--------+--------+-----+
```

As you can see, the final tree structures are different depending on the order of deletion. This example shows that the commutative property does not hold for deletion in a BST. The resulting tree can be different depending on the order of deletion and the relationship between the elements being deleted.

3 Question

Algorithm 1 TransformTreeToAllLeftNIL(T, root)

Require: T: the binary tree, $root$: the root node
Ensure: Transformed tree with left children as NIL

procedure TransformTreeToAllLeftNIL($T, root$)
    Initialize a variable *node* to the root of the tree
    Initialize a counter *count* to 0
    while *node* is not NIL **do**
        if *node* has a left child ($node.left$) **then**
            Perform RIGHT-ROTATE($T, node, node.left$)
        end if
        Update *node* to be the right child of the original *node* ($node = node.right$)
        Increment *count* by 1
    end while
    return Transformed tree T
end procedure

The algorithm 'TransformTreeToAllLeftNIL' takes as input a binary tree 'T' and the root node 'root'. It uses a 'node' variable to traverse the tree from the root. The goal is to make the left child of every node NIL by repeatedly performing RIGHT-ROTATE operations.

The algorithm iterates through the leftmost path of the tree, checking if each node has a left child. If a left child exists, it performs a RIGHT-ROTATE operation to make the left child NIL. Then, it updates the 'node' to the right child of the original 'node'. The process continues until the entire leftmost path has been processed.

The algorithm returns the transformed tree 'T', where all left children are NIL. The number of RIGHT-ROTATE operations is equal to 'n - 1', ensuring an O(n) time complexity.

```python
class TreeNode:
    def __init__(self, key):
        self.key = key
        self.left = None
        self.right = None

def insert(root, key):
    if root is None:
        return TreeNode(key)
    if key < root.key:
        root.left = insert(root.left, key)
    else:
        root.right = insert(root.right, key)
```

```python
    return root

def left_rotate(root, u, v):
    if u is None or v is None:
        return

    # Perform the left-rotate operation
    if u.right:
        u.right, v.left, v, u.right = u.right, v, u, u.right
    else:
        u.right, v.left, v = v.left, v, u

def transform_tree_to_all_left_nil(root):
    node = root
    while node:
        if node.left:
            left_rotate(root, node, node.left)
            node.left = None  # Reset the left child to None
        node = node.right

def in_order_traversal(root):
    if root:
        in_order_traversal(root.left)
        print(root.key, end=" ")
        in_order_traversal(root.right)

def print_tree(root):
    in_order_traversal(root)
    print()

if __name__ == "__main__":
    keys = [5, 3, 8, 1, 4]
    root = None
    for key in keys:
        root = insert(root, key)

    print("Original BST:")
    print_tree(root)
    print()
    print("Transformed Left Nil BST:")
    transform_tree_to_all_left_nil(root)
    print_tree(root)
```

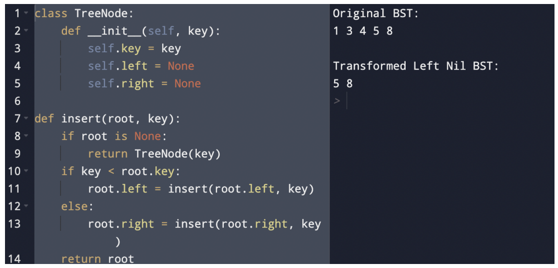

Figure 1: Output

4 Question

Full Code:

```
def merge(arr, p, q, r):
    n1 = q - p + 1
    n2 = r - q

    left = arr[p:p + n1]
    right = arr[q + 1:q + 1 + n2]

    i = j = 0
    k = p

    while i < n1 and j < n2:
        if left[i] <= right[j]:
            arr[k] = left[i]
            i += 1
        else:
            arr[k] = right[j]
            j += 1
        k += 1

    while i < n1:
        arr[k] = left[i]
        i += 1
        k += 1

    while j < n2:
        arr[k] = right[j]
        j += 1
        k += 1

def merge_sort(arr):
```

```
    n = len(arr)
    curr_size = 1

    while curr_size < n:
        left_start = 0

        while left_start < n - 1:
            mid = min(left_start + curr_size - 1, n - 1)
            right_end = min(left_start + 2 * curr_size - 1, n - 1)

            merge(arr, left_start, mid, right_end)

            left_start += 2 * curr_size

        curr_size *= 2

if __name__ == "__main__":
    arr = [38, 27, 43, 3, 9, 82, 10]
    print("Original array:", arr)

    merge_sort(arr)

    print("Sorted array:", arr)
```

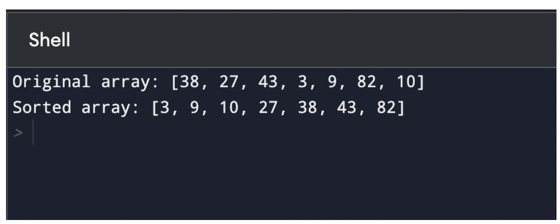

Figure 2: Output

Algorithm 2 Merge Sort Algorithm with Stack (Python Code)

```
function MERGE(arr, p, q, r)
    n1 ← q − p + 1
    n2 ← r − q
    left ← arr[p : p + n1]
    right ← arr[q + 1 : q + 1 + n2]
    i ← 0
    j ← 0
    k ← p
    while i < n1 and j < n2 do
        if left[i] ≤ right[j] then
            arr[k] ← left[i]
            i ← i + 1
        else
            arr[k] ← right[j]
            j ← j + 1
        end if
        k ← k + 1
    end while
    while i < n1 do
        arr[k] ← left[i]
        i ← i + 1
        k ← k + 1
    end while
    while j < n2 do
        arr[k] ← right[j]
        j ← j + 1
        k ← k + 1
    end while
end function
function MERGESORT(arr)
    n ← lengthofarr
    stack ← []
    stack.append((0, n − 1))
    while stackisnotempty do
        (p, r) ← stack.pop()
        if p < r then
            q ← ⌊(p + r)/2⌋
            stack.append(p, q)
            stack.append(q + 1, r)
            MERGE(arr, p, q, r)
        end if
    end while
end function
arr ← [38, 27, 43, 3, 9, 82, 10]
MERGESORT(arr)
Print(”Sorted array:”, arr)
```

5 Question

Algorithm 1 Compute Array B with Nearest Traps

```
procedure FindNearestTrap(A, k)
    n ← length of A
    B ← [None] * n                                    ▷ Initialize B with None values
    queue ← []
    for i ← 0 to n − 1 do
        while queue is not empty and queue[0][1] < i − k do
            queue.pop(0)          ▷ Remove elements that exceed the safety margin
        end while
        if A[i] is None then
            queue ← []               ▷ Reset the queue when a trap is encountered
        else
            if queue is not empty then
                B[i] ← queue[0][0]                      ▷ Store index of nearest trap
            else
                B[i] ← None
            end if
            queue.append((i, i))           ▷ Add the current element to the queue
        end if
    end for
    Return B
end procedure
```

Explanation

The FindNearestTrap algorithm computes an array B where $B[i]$ stores the smallest index j such that $A[j] = \text{None}$, $j \leq i$, and $i - j \leq k$.

- We initialize B as an array of None values and an empty queue.
- We iterate through the elements of array A from left to right (index i).
- For each element, we check whether it's a trap or not:
  - If the current element is a trap (None), we clear the queue to start fresh.
  - If the current element is not a trap, we check the queue to find the nearest trap within the safety margin:
    * We remove elements from the front of the queue until we find a trap or the elements exceed the safety margin ($i-$original_index $> k$).

* We store the index of the nearest trap in $B[i]$ if found; otherwise, $B[i]$ remains None.
* We add the current element to the queue with the original index.

Time Complexity

The time complexity of this algorithm is $O(n)$. This is because each element of array A is processed once, and each element is pushed and popped from the queue at most once. The queue operations are $O(1)$ due to the way we use it in this algorithm. Therefore, the overall time complexity is linear in the size of the input, making it $O(n)$.

Question 6)

Full code.

```
def findMutation(A):
    hashTable = {}
    stack = []

    for i in range(len(A)):
        if A[i] in hashTable:
            if len(stack) >= 2 and stack[-2] == hashTable[A[i]]:
                i2 = stack.pop()
                i1 = stack.pop()
                if A[i1] == A[i2 + 1] and A[i2] == A[i2+2]:

                    return (i1, i2, i2 + 1, i2+2)

        stack.append(i)
        hashTable[A[i]] = i

    return None

A = [5, 4, 5, 3,6,7,8,7,8,7,1,2,3,4,5,6,7]
result = findMutation(A)
if result:
    print("Indices of x-y-x-y pattern:", result)
else:
    print("No x-y-x-y pattern found.")
```

main.py

Save Run

```python
def findMutation(A):
    hashTable = {}
    stack = []

    for i in range(len(A)):
        if A[i] in hashTable:
            if len(stack) >= 2 and stack[-2] == hashTable[A[i]]:
                i2 = stack.pop()
                i1 = stack.pop()
                if A[i1] == A[i2 + 1] and A[i2] == A[i2+2]:

                    return (i1, i2, i2 + 1, i2+2)

        stack.append(i)
        hashTable[A[i]] = i

    return None

A = [5, 4, 5, 3,6,7,8,7,8,7,1,2,3,4,5,6,7]
result = findMutation(A)
if result:
    print("Indices of x-y-x-y pattern:", result)
else:
    print("No x-y-x-y pattern found.")
```

Example Input : [5,4,5,3,6,7,8,7,8,7,1,2,3,4,5,6,7]

Output : (5,6,7,8)

```
Indices of x-y-x-y pattern: (5, 6, 7, 8)
>
```

Algorithm: Find X-Y-X-Y Pattern Indices
Input: An array **A** of length **n** containing elements.
Output: A quadruplet **(i1, i2, i3, i4)** where **1 ≤ i1 < i2 < i3 < i4 ≤ n** and **A[i1] == A[i3] ≠ A[i2] == A[i4]** if such a pattern exists, or **None** if there is no such pattern.

Step-by-Step Explanation:

1. Initialize an empty hash table **hashTable** to store the last index of each element and an empty stack **stack** to keep track of interleaving.
2. Iterate through the elements of the array **A** from left to right using the index variable **i**.
3. For each element **A[i]**:
    - Check if **A[i]** is already present in the **hashTable**.
        - If it is, check whether there are at least two elements in the **stack** and if the second-to-last element in the stack matches

the last index of **A[i]** in the **hashTable**. This is the critical step where you look for the pattern x-y-x-y.

- If this condition is met, pop the last two elements from the stack and label them as **i1** and **i2**.
- Check if **A[i1]** is equal to **A[i2 + 1]** and if **A[i2]** is equal to **A[i2 + 2**. If this condition is met, you have found the x-y-x-y pattern.
  - In this case, return the quadruplet **(i1, i2, i2 + 1, i2 + 2)** as the result.

4. If no pattern is found after processing all elements, return **None** to indicate that there is no x-y-x-y pattern in the array.

Time Complexity Analysis:
The time complexity of this algorithm is O(n^2), where **n** is the length of the input array **A**. Here's why:

1. The primary loop iterates through each element in the array **A**, and for each element, you perform constant time operations (e.g., hash table lookups, comparisons, stack operations).
2. In the worst-case scenario, for each element in the array, you may need to check multiple elements in the stack to find a pattern, which results in O(n) work.
3. Therefore, the overall time complexity of the algorithm is O(n) * O(n) = O(n^2).

The algorithm has an expected O(n^2) time complexity as it potentially needs to examine many elements before finding the x-y-x-y pattern.