Assignment-1

Data structure and Algorithmic Techniques

Jayant parmar
postgraduate diploma in Artificial Intelligence
Indian Institute of Technology Jodhpur
Email: g25ait1072@gmail.com

Q.1. Let A[1:n] be an array of n distinct numbers. If i < j and A[i] > A[j], then the pair (i, j) is called an inversion of A. Give an O(nlogn) time algorithm that determines the number of inversions in A.

Soln:

```
FUNCTION countInversion (list):
    IF length(list) <= 1 THEN
        RETURN list, 0, empty_list.
    mid ← length(list) // 2

    left, left_inv_count, left_pairs ← CountInversion (sublist from 0 to mid)

    right, right_inv_count, right_pairs ← CountInversion( sublist from mid to end)

    merged_list, split_inv_count, split_pairs ← Merge (left, right)

    total_inv ← left_inv_count + right_inv_count + split_inv_count

    Total_pairs ← concatenate (left_pairs, right_pairs, split_pairs)
    RETURN Mergedlist, total_inv, total pairs

END FUNCTION
```

FUNCTION Merge (left, right):
    sorted_list ← Empty_list
    inversion_count = 0
    inversion_pairs ← empty_list
    i = 0
    j = 0
WHILE i < length(left) AND j < length(right)
    IF left[i] ≤ right THEN
        Append left[i] to sorted_list
    ELSE:
        Append right to sorted list
        inversion_count ← inversion_count + (~~left[i]~~
                length(left) - i)
        FOR K FROM i to length(left) - 1:
            Append (left[K], right[j]) to inversion
                pairs
        j = j + 1
    Append remaining elements of left starting at
    index i to sorted_list
    Append remaining elements of right starting at
    index j to sorted_list.

    RETURN sorted_list, inversion_count, inversion_pairs
END FONCTION

TIME COMPLEXITY :
T(n) = 2T(n/2) + O(n)
O(n log n) overall.

Q.2 Describe an O(nlogn) time and O(n) space algorithms that, given a set S of n integers and another integers x, determines whether S contains two elements that sum exactly x. (Hint: Read about binary search)

Soln:

```
FUNCTION FindSum (lst, i, j, x)
    WHILE i < j:
        sum = lst[i] + lst[j]
        IF sum == x THEN
            RETURN (True, (lst[i], lst[j]))
        ELSE IF sum < x THEN
            i = i+1
        ELSE:
            j = j-1
    RETURN (False, Empty-list)
END FUNCTION

lst = mergeSort (input)
FindSum (lst, i, j, x)
```

Explaination:

- mergeSort (S) recursively split & merge while sorting. time complexity O(nlogn)

- FindSum (lst, i, j, x) uses the two pointer method in O(n) time to find a pair whose sum equals x,

- overall time complexity: O(nlogn) due to sorting

Q.3 Let T be a BST. Describe an O(n) time algorithm that on input T.root can find the minimum absolute difference of any two keys of T. For instance, if keys of T are 3, 8, 1, 12, 7, 15 then answer will be 8-7 = 1

Soln :

```
FUNCTION min_abs_diff (value1, value2, current_diff):
    new_diff = ABS (value1 - value2)
    IF new < current_diff:
        RETURN new_diff.
    RETURN current_diff.
END FUNCTION


FUNCTION inorder (root, previous_value, min_diff)
IF root IS NOT NULL:
    previous_value, min_diff = inorder (root.left, previous value
                                        , min_diff)
    IF previous_value IS NOT NULL:
        min_diff = min_abs_diff (root.val, previous_valre,
                                 min_diff)
    previous_value = root.val

    previous_value, min_diff = inorder (root.right,
                                        previous_value, min_diff)
    RETURN previous_value, min_diff.
END FUNCTION
```

Explanation :- The inorder traversal ensures traversal of BST nodes in sorted ascending order. The diff. b/w adjacent value in this order gives the smallest possible differences. Tim complexity O(n)

Q.4 : An array A is called K-unique if it does not contain a pair of duplicate elements within K positions of each other, that is there is no i & j such that $A[i] = A[j]$ and $|i-j| \leq K$. Design an $O(n \log K)$ time algorithm to test if A is K-unique.

Soln :

```
FUNCTION IS_K_UNIQUE ( Array A, integer K ):
    Initialize empty balanced BST set WINDOW

    For i = 0 to length(A) - 1 do :
        if WINDOW contains A[i] then :
            Return False.

        Insert A[i] in WINDOW

        if size('window') > k then
            Remove A[i-K] from WINDOW

    Return TRUE
```

Explanation :

- Thes WINDOW set tracks the last k element as a sliding window.
- For each element in the array :
  if it exists in window set, a duplicate within K position is detected, so return False
  otherwise, insert the element into window.
  when the window size exceeds K, remove the oldest element to maintain only the latest K entries.

if the entire array is passed, without detecting duplicates within the window, return TRUE.

Time complexity : $O(n \log k)$

Q.4 : Second optimized algorithm:

```
FUNCTION find_kunique(list, K)
    create empty dictionary d

    FOR each index, value in list:
        IF value exists in d AND ABS(index - d[value]) <= K:
            RETURN FALSE
        d[value] = index
    RETURN TRUE
```

Explanation :

- Above implementation an efficient algo to determine whether an array is K-unique.

- This implementation runs in O(n) time using a has map (dict) to track recent occurence of each element.

Q.5 A node x is inserted into a red-black tree and then is immediately deleted using the procedures discussed in the class. Is the resulting red-black tree always the same as the initial red-black tree? Justify your answer.

Soln:

No, the resulting red-black tree is not always the same as the initial red black tree.

⇔ Justification:

- the reason lies in the fix up procedures for insertion and deletion. while the operation are designed to restore the red-black tree properties, they are not mathematical inverse of each other.

1. Insertion:

A new node x is always inserted as 'Red'. If this insertion creates a red-red violation, the insertion fix-up procedure is triggered. This fix-up can involve rotations and recoloring that alter the structure and colors of the tree, potentially involving the new node's parents, grandparent and uncle.

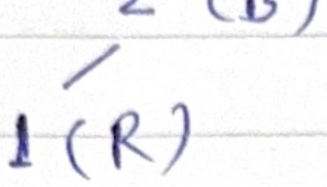
2. Deletion:

- If the node x is still red after the insertion fix-up its deletion is simple and require no further a fix-up
- However, the insertion fix-up might have changed the tree structure before the deletion occurs.

the structural changes made by the insertion fix-up are not necessarily reversed by the deletion of the newly added node.

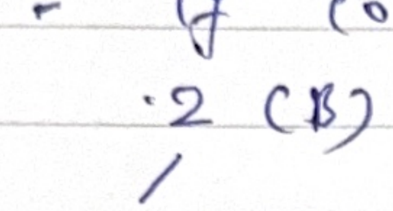
· example:

① Initial tree:



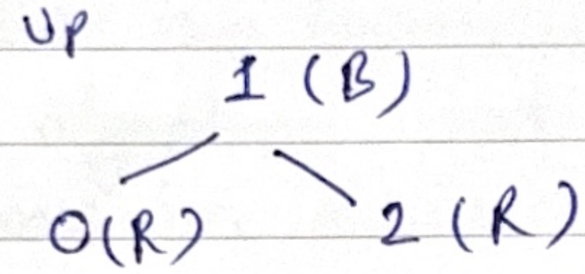
② Insert Node (0):

- if (0) is red.



← violation (red-red violation)

- fix-up

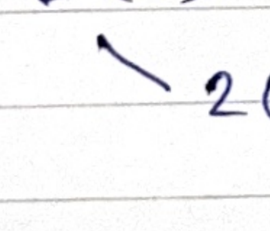


③ Delete Node (0):

Node (0) is red

- Deleting a red node does not ~~violation~~ violate red-black properties

④ Final tree:



⑤ Comparison:

Initial tree structure ≠ Final tree structure.

Q.6 : Given an element x in an node augmented tree. (the one which can find the rank of an element or find the element of the given rank) and a natural no. i, show how to determine the i^{th} successor of x in O(dogn) time.

soln

```
FUNCTION find_Kth (node, k) :
    IF node is NULL :
        Return NULL

    left_size = size of node.left IF EXists
                ELSE 0

    IF K == left_size + 1
        Return node.value

    Else if K ≤ left_size
        return find_Kth (node.left, K)

    ELSE :
        return : fnd. kth (node.right, K - left.size - 1)

END FUNCTION

FUNCTION find-Kth (K) ;
        RETURN : find. Kth (root, K)
END FUNCTION
```

Explanation

The algo takes advantage of node size augmentation to achieve algorithmic time for ranked based queries, Demonstrating efficient use in Binary search trees.

time complexity O(logn)