

Assignment-1

Data Structure and Algorithmic Techniques

Jayant Parmar
Postgraduate Diploma in Artificial Intelligence
Optimization for Data Science
Indian Institute of Technology Jodhpur, Jodhpur, India
Email: g25ait1072@iitj.ac.in

Vimal Raj Sharma
Indian Institute of Technology Jodhpur, Jodhpur, India

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 $O(n \log n)$ -time algorithm that determines the number of inversions in A .

FUNCTION CountInversion(list):

IF length(list) <= 1 THEN

 RETURN list, 0, empty_list

mid \leftarrow length(list) // 2

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

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

merged_list, split_inv_count, split_pairs \leftarrow Merge(left, right)

total_inv \leftarrow left_inv_count + right_inv_count + split_inv_count

total_pairs \leftarrow concatenate(left_pairs, right_pairs, split_pairs)

RETURN merged_list, total_inv, total_pairs

END FUNCTION

FUNCTION Merge(left, right):

sorted_list \leftarrow empty_list

inversion_count \leftarrow 0

inversion_pairs \leftarrow empty_list

i \leftarrow 0

j \leftarrow 0

WHILE i < length(left) AND j < length(right):

```

IF left[i] ≤ right[j] THEN
    Append left[i] to sorted_list
    i ← i + 1
ELSE:
    Append right[j] to sorted_list
    inversion_count ← inversion_count + (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 FUNCTION

Time Complexity:

$$T(n) = 2T(n/2) + O(n)$$

$O(n \log n)$ overall.

```

IIT-Jodhpur > 1st year > Trimester-1 > DSAT > Assignments > 1 > Q1.py > merge
 1 def count_inversion(lst):
 2     mid = len(lst) // 2
 3     if len(lst) <= 1:
 4         return lst, 0, []
 5     left, left_inv, left_pairs = count_inversion(lst[:mid])
 6     right, right_inv, right_pairs = count_inversion(lst[mid:])
 7     sorted_lst, inv, inv_pairs = merge(left, right)
 8     total_inv = left_inv + right_inv + inv
 9     total_pairs = left_pairs + right_pairs + inv_pairs
10     return sorted_lst, total_inv, total_pairs
11
12 def merge(left, right):
13     sorted_list = []
14     count = 0
15     inversion_pairs = []
16     i = j = 0
17     while i < len(left) and j < len(right):
18         if left[i] <= right[j]:
19             sorted_list.append(left[i])
20             i += 1
21         else:
22             sorted_list.append(right[j])
23             count += len(left) - i
24             for k in range(i, len(left)):
25                 inversion_pairs.append((left[k], right[j]))
26             j += 1
27     sorted_list.extend(left[i:])
28     sorted_list.extend(right[j:])
29     return sorted_list, count, inversion_pairs
30
31 if __name__ == "__main__":
32     lst = [1, 2, 4, 3, 5, 9, 7, 8, 6]
33     print("Original list:", lst)
34     _, total_inversions, inversion_pairs = count_inversion(lst)
35     print("Total inversions:", total_inversions)
36     print("Inversion pairs:")
37     for pair in inversion_pairs:
38         print(pair)
39

```

PROBLEMS 1 OUTPUT DEBUG CONSOLE TERMINAL PORTS

```

• (base) jayantparmar@Jayants-Air 1 % python Q1.py
Original list: [1, 2, 4, 3, 5, 9, 7, 8, 6]
Total inversions: 6
Inversion pairs:
(4, 3)
(8, 6)
(7, 6)
(9, 6)
(9, 7)
(9, 8)

```

○ (base) jayantparmar@Jayants-Air 1 % []

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

```
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, EmptyList)
END FUNCTION
```

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

MergeSort(s) recursively splits and merges while sorting — time complexity $O(n \log n)$.
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: $O(n \log n)$ due to sorting.

```

1  def mearge_sort(s):
2      mid = len(s) // 2
3      if len(s) <= 1:
4          return s
5      left = mearge_sort(s[:mid])
6      right = mearge_sort(s[mid:])
7      sorted_lst = merge(left, right)
8      return sorted_lst
9
10 def merge(left, right):
11     sorted_list = []
12     i = j = 0
13     while i < len(left) and j < len(right):
14         if left[i] <= right[j]:
15             sorted_list.append(left[i])
16             i += 1
17         else:
18             sorted_list.append(right[j])
19             j += 1
20     sorted_list.extend(left[i:])
21     sorted_list.extend(right[j:])
22     return sorted_list
23 def find_sum(lst, i, j, x):
24     sum = 0
25     while(i<j):
26         sum = lst[i] + lst[j]
27         if sum == x:
28             return True, (lst[i],lst[j])
29         elif sum < x:
30             i += 1
31         else:
32             j -= 1
33     return False, []
34
35 if __name__ == "__main__":
36     input_set = set([8, 20, 3, 14, 6])
37     x = 11
38     print("Original Set:", input_set)
39     input_list = list(input_set)
40     sorted_list = mearge_sort(input_list)
41     status, sum_pairs = find_sum(sorted_list, 0, len(input_list)-1, x)
42     print("Sum exist status :", status)
43     print("Sum pairs:", sum_pairs)

```

PROBLEMS 1 OUTPUT DEBUG CONSOLE TERMINAL PORTS

- (base) jayantparmar@Jayants-Air 1 % python Q2.py

Original Set: {3, 6, 8, 14, 20}

Sum exist status : False

Sum pairs: []
- (base) jayantparmar@Jayants-Air 1 % python Q2.py

Original Set: {3, 6, 8, 14, 20}

Sum exist status : True

Sum pairs: (3, 8)
- (base) jayantparmar@Jayants-Air 1 % []

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 $8 - 7 = 1$.

FUNCTION min_abs_diff(value1, value2, current_diff):

 new_diff \leftarrow ABS(value1 - value2)

 IF new_diff < 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 \leftarrow inorder(root.left, previous_value, min_diff)

 IF previous_value IS NOT NULL:

 min_diff \leftarrow min_abs_diff(root.val, previous_value, min_diff)

 previous_value \leftarrow root.val

 previous_value, min_diff \leftarrow inorder(root.right, previous_value, min_diff)

 RETURN previous_value, min_diff

END FUNCTION

The inorder traversal ensures traversal of BST nodes in sorted ascending order.

The difference between adjacent values in this order gives the smallest possible differences.

Time complexity: $O(n)$ (each node visited once)

```
IIT-Jodhpur > 1st year > Trimester-1 > DSAT > Assignments > 1 > Q3.py > ...
1  class Node:
2      def __init__(self, key):
3          self.left = None
4          self.right = None
5          self.val = key
6  def insert(root, key):
7      if root is None:
8          return Node(key)
9      if root.val == key:
10         return root
11     if root.val < key:
12         root.right = insert(root.right, key)
13     else:
14         root.left = insert(root.left, key)
15     return root
16 def min_abs_diff(n1, n2, diff):
17     new_diff = abs(n1 - n2)
18     if new_diff < diff:
19         return new_diff
20     return diff
21 def inorder(root, pre_val, diff):
22     if root:
23         pre_val, diff = inorder(root.left, pre_val, diff)
24         if pre_val is not None:
25             diff = min_abs_diff(root.val, pre_val, diff)
26         pre_val = root.val
27         print(pre_val)
28         pre_val, diff = inorder(root.right, pre_val, diff)
29
30     return pre_val, diff
31 if __name__ == '__main__':
32     r = Node(50)
33     r = insert(r, 30)
34     r = insert(r, 29)
35     r = insert(r, 40)
36     r = insert(r, 70)
37     r = insert(r, 60)
38     r = insert(r, 80)
39     r = insert(r, 8)
40     print("Inorder Traversal : ")
41     _, diff = inorder(r, None, float('inf'))
42     print(f"Minimum absolute difference is {diff} ")
43
```

PROBLEMS 1 OUTPUT DEBUG CONSOLE TERMINAL PORTS

● (base) jayantparmar@Jayants-Air 1 % python Q3.py

Inorder Traversal :

```
8
29
30
40
50
60
70
80
```

Minimum absolute difference is 1

○ (base) jayantparmar@Jayants-Air 1 %

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 and j such that $A[i] = A[j]$ and $|j - i| \leq k$. Design an $O(n \log k)$ time algorithm to test if A is k -unique.

```
FUNCTION find_kunique(list, k):
    CREATE empty dictionary d

    FOR each index, value in list:
        IF value exists in dictionary d AND ABS(index - d[value]) ≤ k THEN
            PRINT "value at index", index, "and index", d[value],
                  "is same and li-jl is", ABS(index - d[value]),
                  "which is less than or equal to k =", k
        RETURN False

        d[value] ← index // Update the last occurrence index of the value

    RETURN True
END FUNCTION
```

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.

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 operations are designed to restore the red-black tree properties, they are not mathematical inverses of each other.

1. **Insertion:** A new node X is always inserted as **red**. If this insertion creates a **red-red violation** (i.e., its parent is also red), the insertion fix-up procedure is triggered. This fix-up can involve **rotations** and **recolorings** that alter the structure and colors of the tree, potentially involving the new node's parent, grandparent, and uncle.

2. **Deletion:** When the *same* node \$X\$ is immediately deleted, the deletion procedure is applied.

* If the node \$X\$ is still **red** after the insertion fix-up (or if no fix-up was needed), its deletion is simple and requires no further fix-up.

* However, the insertion fix-up *might have changed the tree structure* (due to rotations) before the deletion occurs.

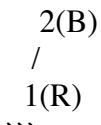
The structural changes made by the insertion fix-up are not necessarily reversed by the (often trivial) deletion of the newly added node.

Counterexample

Let's walk through a case where the final tree is different from the initial one.

1. Initial Tree

Consider the following valid red-black tree. (NIL leaves are omitted for clarity).

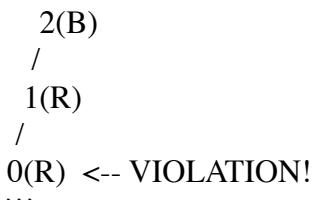


- * **Root:** 2 (Black)
- * **Node 1:** 1 (Red)
- * This tree satisfies all RBT properties.

2. Insert Node (0)

- * We insert `0` as a **red** node, which becomes the left child of `1`.

<!-- end list -->



- * This creates a **red-red violation** (node `0` and its parent `1` are both red).

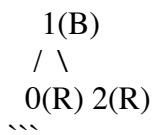
- * The **insertion fix-up** procedure is triggered (this is Case 3: the uncle is black (NIL)).

1. Recolor parent `1` to **black**.

2. Recolor grandparent `2` to **red**.
3. Perform a **right rotation** on the grandparent `2`.

* **Tree After Insertion and Fix-up:** The tree is now:

<!-- end list -->



* This is a valid red-black tree.

3. Delete Node (0)

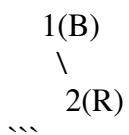
* Now, we delete the node we just inserted, `0`.

* Node `0` is **red**.

* Deleting a red node **does not violate any red-black properties** and requires no fix-up procedure. We simply remove it.

* **Final Tree:**

<!-- end list -->



4. Comparison

* **Initial Tree:** Root `2(B)` with left child `1(R)`.

* **Final Tree:** Root `1(B)` with right child `2(R)`.

The final tree is structurally different from the initial tree, even though they contain the same set of keys (1 and 2). Therefore, inserting and then immediately deleting a node does not always result in the original tree.

6. Given an element x in an n -node augmented tree (the one which can find the rank of an element or find the element of the given rank) and a natural number i , show how to determine the i th successor of x in $O(\log n)$ time.

FUNCTION _find_kth(node, k):

 IF node IS NULL:

 RETURN NULL

 left_size \leftarrow size of node.left IF EXISTS ELSE 0

 IF k == left_size + 1:

 RETURN node.value

 ELSE IF k \leq left_size:

 RETURN _find_kth(node.left, k)

 ELSE:

 RETURN _find_kth(node.right, k - left_size - 1)

END FUNCTION

FUNCTION find_kth(k):

 RETURN _find_kth(root, k)

END FUNCTION

find_kth / rank / kth_successor: $O(\log n)$

The algorithm takes advantage of node size augmentation to achieve logarithmic time for rank-based and order-statistic queries, demonstrating efficient augmentation use in binary search trees.

```

IIT-Jodhpur > 1st year > Trimester-1 > DSAT > Assignments > 1 > Q6.py > AugmentedBST > _rank

32     def _find_kth(self, node, k):
33         if not node:
34             return None
35         left_size = node.left.size if node.left else 0
36         if k == left_size + 1:
37             return node.value
38         elif k <= left_size:
39             return self._find_kth(node.left, k)
40         else:
41             return self._find_kth(node.right, k - left_size - 1)
42
43     def find_kth(self, k):
44         return self._find_kth(self.root, k)
45
46     def _rank(self, node, key):
47         if not node:
48             return 0
49         if key < node.value:
50             return self._rank(node.left, key)
51         elif key > node.value:
52             left_size = node.left.size if node.left else 0
53             return left_size + 1 + self._rank(node.right, key)
54         else:
55             left_size = node.left.size if node.left else 0
56             return left_size + 1
57
58     def rank(self, key):
59         return self._rank(self.root, key)
60
61     def kth_successor(self, x, k):
62
63         r = self.rank(x)
64         if r == 0:
65             print(f"Element {x} not found in the tree.")
66             return None
67         target_rank = r + k
68         if not self.root or target_rank > self.root.size:
69             return None # No such successor
70         return self.find_kth(target_rank)
71
72 if __name__ == "__main__":
73     tree = AugmentedBST()
74     for val in [20, 8, 22, 4, 12, 10, 14]:
75         tree.insert(val)
76
77     x, k = 10, 2
78     print(f"The {k}-th successor of {x} is:", tree.kth_successor(x, k))
79

```

PROBLEMS 1 OUTPUT DEBUG CONSOLE TERMINAL PORTS

- (base) jayantparmar@Jayants-Air 1 % python Q6.py

The 2-th successor of 10 is: 14
- (base) jayantparmar@Jayants-Air 1 %