CS345: Algorithms - II Due: 13/09/19

Assignment #2

Name: **Ayush Hitesh Soneria** Instructor: Prof. Nitin Saxena

Roll-No: **170192** TA: Pranav Bisht

Email: ayushhs@iitk.ac.in OR ayushhs@cse.iitk.ac.in TA's Email: pbisht@cse.iitk.ac.in

Max Points: 50

Question 1: MinGap of Set

[6 points]

We will design a tree structure as an augmented AVL tree. Every node in this tree will have the following fields: Augmentation (w.r.t to each node, say v) as follows:

- left: pointer to left child of v
- right: pointer to right child of v
- key: number stored in the v
- maxkey: maximum key in the subtree rooted at v
- minkey: minimum key in the subtree rooted at v
- mingap: minimum positive difference in the subtree rooted at v

The 3 additional fields, maxkey, minkey and mingap need to be updated for insertion and deletion operations without increasing time complexity which is O(logn). Hence we will do the update them recursively:

```
   Data: v ← node to be inserted or deleted
 1 /* In this algorithm we are using minimum function which gives minimum of all its
      arguments                                                                    */
 2 INT_MAX ← Maximum possible integer in a computer;
 3 if node v has no children then
 4 |   v.maxkey = v.key;
 5 |   v.minkey = v.key;
 6 |   v.mingap = INT_MAX;
 7 end
 8 else if node v has no right child then
 9 |   v.maxkey = v.key;
10 |   v.minkey = v.left.minkey;
11 |   v.mingap = minimum(INT_MAX, v.left.mingap, v.key-v.left.maxkey);
12 end
13 else if node v has no left child then
14 |   v.maxkey = v.right.maxkey;
15 |   v.minkey = v.key;
16 |   v.mingap = minimum(INT_MAX, v.right.mingap, v.right.minkey-v.key);
17 end
18 else                                                        /* both children present */
19 |   v.maxkey = v.right.maxkey;
20 |   v.minkey = v.left.minkey;
21 |   v.mingap = minimum(v.right.mingap, v.right.minkey-v.key, v.left.mingap, v.key-v.left.maxkey);
22 end
```

Algorithm 1: updation for standard tree operations in $O(logn)$

Question 1: Continued [6 points]

Proof of Correctness & Analysis

We need maxkey and minkey in addition to mingap is because we need to also include the node.key itself in our calculations when getting mingap, not just its subtree's mingaps. So we now can also include v.right.minkey-v.key and v.key-v.left.maxkey in finding mingap of v.
During standard insertions and deletions in order to balance AVL tree height rotations may be done. The above algorithm needs to be implemented on the two rotated nodes. As we can see atmost O(1) such rotations may be done, and only the nodes in the path from updated nodes to root have to be updated as well. Meaning we will have to update only O(logn) nodes with above algorithms which itself runs in O(1) time. Hence updation for standard tree operations is done in O(logn) with our augmented AVL tree.
Once the elements of set T are inserted in augmented AVL tree (pre-processing) then MinGap(T) is mingap field of the root of AVL tree, hence MinGap(T) can be done in O(1).

Question 2: Orthogonal Range Search [5 points]

As seen in class for Orthogonal Range Search, we use an augmented AVL tree, Xtree, where each node v contains the x-coordinate value and an AVL tree w.r.to y-coordinates, Ytree(v). The Ytree(v) is not augmented, it only contains the y-coordinate in our implementation done in class.
But let us modify our algorithm, suppose now we augment the Ytrees as well, with a size field stored in each node of the Ytree. The size field of a node v, will contain the sum of number of nodes in subtrees of v. Hence now we can tweak the pseudo-code discussed in class to get required output as follows:
Suppose given rectangle coordinates are (x1,y1) and (x2,y2). In Xtree suppose node of x1 is s and node of x2 is t. Let p be the LCA(lowest common ancestor) of s and t. We need the points which lie between x1 and x2, as well as y1 and y2.

Pseudo-code

```
   Data: Xtree ← Augmented AVL tree with x-coordinates and Ytrees
   Data: Ytree ← Augmented AVL tree with y-coordinates and size
   Result: Number of points in a given rectangle
 1 Function Count(T,x1,x2,y1,y2):
 2     int ans=0;
 3     if right child of s is not NULL then
 4         /* checking for blue nodes and blue subtrees as discussed in class                */
 5         ans += the number of nodes in s.right.Ytree between y1 and y2 using size field;
 6         /* to know which nodes are between y1 and y2, do RangeSearch(Ytree,y1,y2) as
                discussed in class and keep a counter using the size field,will take O(logn)   */
 7     end
 8     while s is not equal to p do                                  /* traverse from s to LCA */
 9         if s is left child of its parent then
10             /* checking for blue nodes and blue subtrees as discussed in class            */
11             ans += the number of nodes in s.parent.Ytree and s.parent.right.Ytree between y1 and y2 using
                size field;
12             /* to know which nodes are between y1 and y2, do RangeSearch(Ytree,y1,y2) as
                   discussed in class and keep a counter using the size field,will take O(logn)
                   */
13         end
14         s = s.parent;
15     end
16     similarly for path t to p;
17     return ans;
18 End Function
```

Algorithm 2: Orthogonal Range Count in $O(log^2 n)$

Question 2: Continued [5 points]

Proof of Correctness & Analysis

After insertion/deletions of a node in AVL, the tree can become height unbalanced. In order to balance it, rotations may be necessary. During rotations, all the fields of the node have to be carefully updated, especially the size field. Right-Rotate(y):

$$x = y.left$$
$$y.size = y.left.size + y.right.size$$
$$x.size = x.left.size + x.right.size$$

So rotations take O(1) time, also standard tree operations can be done on the Ytree in O(logn) time as only size field updation required. The two while loops in the algorithm itself have O(logn) iterations and in each iteration RangeSearch(Ytree,y1,y2) will take O(logn), hence time complexity of Orthogonal Range Count is $O(log^2 n)$.

Question 3: Data Structure that stores Sequence of Numbers [13 points]

The data structure D we will design will be an augmented AVL tree, which will perform store sequence of numbers and do the asked operations in O(logn) time.
We will store the following fields in each node in D:
Augmentation (w.r.t to each node, say v) as follows:

- left: pointer to left child of v
- right: pointer to right child of v
- parent: pointer to parent of v
- value: number stored in the v
- size: size of subtree rooted at v
- minsub: minimum value in the subtree rooted at v

Let us initialize the the fields in tree D (pre-processing). The left,right,parent and value fields are trivial and should be stored such that when the general InOrder Traversal is applied to D, the output is the sequence of numbers remembering the array order. We will initialise the min and size fields recursively:

if node v is leaf then v.minsub = v.value, else

$$v.minsub = min(v.value, v.left.minsub, v.right.minsub)$$

if node v is leaf then v.size = 1, else

$$v.size = v.left.size + v.right.size + 1$$

Sub - Problem (1): Insert(D,i,x)

Please look at Algorithm for Insert(D,i,x) on next page before going through analysis given next.

Proof of Correctness & Analysis

After insertion of a node in AVL, the tree can become height unbalanced. In order to balance it, rotations may be necessary. During rotations, all the fields of the node have to be carefully updated, especially the size and minsub fields (given on next page).
This algorithm is very similar to the insertion algorithm in a normal AVL tree whose complexity if O(logn). During balancing, extra field updating also will take only O(logn) time as O(1) update operations have to be performed for O(logn) nodes. Hence the time complexity of Insert(D,i,x) algorithm is O(logn).

Question 3: Continued

[13 points]

Update during Right-Rotate(y):

$$x = y.left$$
$$y.size = y.left.size + y.right.size + 1$$
$$x.size = x.left.size + x.right.size + 1$$
$$y.minsub = min(y.value, y.left.minsub, y.right.minsub)$$
$$x.minsub = min(x.value, x.left.minsub, x.right.minsub)$$

Similarly for left-rotate, also here we take size(NULL)=0 and minsub(NULL)=0. We can see here that balancing during single rotation will take O(1) time.

Sub - Problem (1): Insert(D,i,x): Pseudo-code

```
   Data: D ← Augmented initialized AVL tree
   Data: root ← root of D
   Result: updated D after insertion of x at i-th location
 1 Function Insert(root,i,x):                    /* algorithm as a recursive function */
 2     if (root == NULL) then
 3         p ← allocate new node;
 4         p.left = NULL;
 5         p.right = NULL;
 6         p.parent = NULL;
 7         p.value = x;
 8         p.size = 1;
 9         p.minsub = x;
10         return p;
11     end
12     else
13         root.size = root.size + 1;
14         if (root.minsub > x) then
15             root.minsub = x;
16         end
17         if (root.left == NULL) then
18             a ← 0;
19         else                                                          /*  */
20             a ← root.left.size;
21         end
22         if (i <= a+1) then
23             root.left = Insert(root.left,i,x);
24             root.left.parent = root;
25         else                                                          /*  */
26             root.right = Insert(root.right,i-a-1,x);
27             root.right.parent = root;
28         end
29         return root;
30     end
31 End Function
32 Balance the Tree if it turns out to be unbalanced
```

Algorithm 3: Insert(D,i,x) in $O(logn)$

Sub - Problem (2): Delete(D,i): Pseudo-code

```
   Data: D ← Augmented initialized AVL tree
   Data: root ← root of D
   Result: updated D after Deletion of x at i-th location
 1 Function Delete(root,i):                          /* algorithm as a iterative function */
 2 |  if (root == NULL) then
 3 |  |  return;
 4 |  end
 5 |  p ← node storing i-th element;
 6 |  if (p.left != NULL) and (p.right == NULL) then
 7 |  |  if (p.parent.right == p) then
 8 |  |  |  p.parent.right = p.left;
 9 |  |  else
10 |  |  |  p.parent.left = p.left;
11 |  |  end
12 |  else if (p.left == NULL) and (p.right != NULL) then
13 |  |  if (p.parent.right == p) then
14 |  |  |  p.parent.right = p.right;
15 |  |  else
16 |  |  |  p.parent.left = p.right;
17 |  |  end
18 |  else if (p.left == NULL) and (p.right == NULL) then
19 |  |  if (p.parent.right == p) then
20 |  |  |  p.parent.right = NULL;
21 |  |  else
22 |  |  |  p.parent.left = NULL;
23 |  |  end
24 |  else                                              /* need to find successor of p */
25 |  |  q = p.right;
26 |  |  while (q.left != NULL) do
27 |  |  |  q.size = q.size - 1;   /* need to update size field in path from p to successor of p */
28 |  |  |  q = q.left;
29 |  |  end
30 |  |  p.value = q.value;
31 |  |  q.parent.left = q.right;                                  /* delete node q */
32 |  |  if (q.right != NULL) then
33 |  |  |  q.right.parent = q.parent.left;
34 |  |  end
35 |  |  q = q.parent;
36 |  |  while (q != p) do    /* need to update minsub field in path from p to successor of p */
37 |  |  |  q.minsub = minimum(q.value,q.left.minsub,q.right.minsub);
38 |  |  |  q = q.parent;
39 |  |  end
40 |  end
41 |  while (p != root) do    /* need to update minsub and size fields in path from p to root */
42 |  |  p.size = p.size - 1;
43 |  |  p.minsub = minimum(p.value,p.left.minsub,p.right.minsub);
44 |  |  p = p.parent;
45 |  end
46 |  root.size = root.size - 1;                              /* update root as well */
47 |  root.minsub = minimum(root.value,root.left.minsub,root.right.minsub);
48 |  return root;
49 End Function
```

Algorithm 4: Delete(D,i) in $O(logn)$

Question 3: Continued [13 points]

Sub - Problem (3): Delete(D,i): Continued

Proof of Correctness & Analysis

When the node p to be deleted has one or zero children, then trivial case. Just update the fields of minsub and size in the path from deleted node to root. When node p has two children then find successor (decrement size values along path of p to successor) and replace the value of p. Then delete successor node and replace minsub values from successor to p. Then update the fields of minsub and size in the path from p to root.

Finding successor will be O(logn) time. The two updation while loops will have max of log n iterations as there can be iterations = height of tree at max. Also just like insertion, balancing tree might be needed after deletion which again would take O(logn) time.
Hence time complexity of Delete(D,i) is O(logn).

Sub - Problem (3): Report(D,i)

Pseudo-code

```
   Data: D ← Augmented initialized AVL tree
   Result: Number at i-th locations
 1 Function Report(D,i):                          /* algorithm as a iterative function */
 2 |   root ← root of D;
 3 |   check ← 0;
 4 |   while (check == 0) do
 5 |   |   if (root.left == NULL) then
 6 |   |   |   a ← 0;
 7 |   |   else                                                              /*  */
 8 |   |   |   a ← root.left.size;
 9 |   |   end
10 |   |   if (i == a+1) then
11 |   |   |   check ← 1;
12 |   |   else if (i > a+1) then                                            /*  */
13 |   |   |   i = i - a - 1;
14 |   |   |   root = root.right;
15 |   |   else
16 |   |   |   root = root.left;
17 |   |   end
18 |   end
19 |   return root.value;
20 End Function
```

Algorithm 5: Report(D,i) in $O(logn)$

Proof of Correctness & Analysis

We are traversing some path from root down to the i-th location. The maximum we can traverse by taking any path in a tree is the height of the tree which is equal to log n.
Hence time complexity of Report(D,i) is O(log n).

Question 3: Continued [13 points]

Sub - Problem (4): Min(D,i,j)

Pseudo-code

```
   Data: D ← Augmented initialized AVL tree
   Result: Smallest number from the i-th to the j-th locations
 1 Function Min(D,i,j):                                  /* algorithm as a iterative function */
 2 |  /* In this algorithm we are using minimum function which gives minimum of all its
   |     arguments                                                                          */
 3 |  s ← node storing ith element;
 4 |  t ← node storing jth element;
 5 |  lca ← lowest common ancestor (LCA) of s and t;          /* this step takes O(logn) time */
 6 |  if (s != lca) then
 7 |  |  m1 = s.value;                  /* m1 will store the minimum value on left side of lca */
 8 |  |  if (s.right != NULL) then
 9 |  |  |  m1 = minimum(m1, s.right.minsub);
10 |  |  end
11 |  |  while (s.parent != lca) do                          /* we traverse from s to lca */
12 |  |  |  if (s == s.parent.left) then
13 |  |  |  |  /* positions of s.parent and subtree s.parent.right belong in range i to j   */
14 |  |  |  |  m1 = minimum(m1, s.parent.value, s.parent.right.minsub);
15 |  |  |  end
16 |  |  |  s = s.parent;
17 |  |  end
18 |  end
19 |  if (t != lca) then
20 |  |  m2 = t.value;                  /* m2 will store the minimum value on right side of lca */
21 |  |  if (t.left != NULL) then
22 |  |  |  m2 = minimum(m2, t.left.minsub);
23 |  |  end
24 |  |  while (t.parent != lca) do                          /* we traverse from t to lca */
25 |  |  |  if (t == t.parent.right) then
26 |  |  |  |  /* positions of t.parent and subtree t.parent.left belong in range i to j    */
27 |  |  |  |  m2 = minimum(m2, t.parent.value, t.parent.left.minsub);
28 |  |  |  end
29 |  |  |  t = t.parent;
30 |  |  end
31 |  end
32 |  return minimum(m1, m2, lca.value);                          /* Required Output */
33 End Function
```

Algorithm 6: Min(D,i,j) in $O(logn)$

Proof of Correctness & Analysis

In the algorithm we only consider nodes (and their right subtrees) whose left child is in the path from LCA to s and the nodes (and their left subtrees) whose right child is in the path from LCA to t. One can easily visualize these nodes and subtrees lie between s and t. Meaning these nodes and subtrees will contain values which lie between the i-th (node s) and j-th (node t) locations. Hence the minimum value which we obtain is provably correct.

The two while loops in the algorithm take O(logn) time because maximum traversal can be from leaf to root, which has height log n. Finding out the LCA also takes O(logn) time. Hence time complexity of Min(D,i,j) is O(logn).

Question 4: Overlapping Rectangles [12 points]

Algorithm

Data: $n \leftarrow$ number of rectangles
Data: $x1, x2, y1, y2 \leftarrow$ coordinates of each given rectangles
1 Store the y-coordinates (y1,y2) of all rectangles in an array, A. So A will have 2n elements
2 Associate fields (maybe using pointers or structs), color and id with each element, s.t. if y-coordinate is lower edge of its rectangle then color will be blue else color red and id will be unique number distinguishing each rectangle coordinates from others (each rectangle will be given a unique id when taken as input)
3 Sort (in non-decreasing order) the array A according to its y-coordinate value
4 Create a empty augmented balanced BST such as augmented AVL tree, say T
5 T will take as input x-coordinates of rectangles as keys and will also keep a count field which will be zero initially
6 Start **vertical line sweep** from bottom, meaning start iteration from first element of A (horizontal rectangle edge with least y-coordinate)
7 Here we implement a **while loop** such that if we ever see the red edge of last element of A (horizontal rectangle edge with highest y-coordinate), we break loop
8 Every time we come across a **blue edge** of rectangle R (identified using id of edge from A), insert x1 and x2 of R into T
9 Also increment (++) count field in the nodes of T which are between the x1,x2 of R
10 We also need to keep a variable, **MX**, which keeps the maximum out of all the node's (maximum attained) count fields
11 Every time we come across a **red edge** of rectangle R (identified using id of edge from A), delete x1 and x2 of R from T
12 Also decrement (- -) count field in the nodes of T which are between the x1,x2 of R
13 To check the nodes which are between x-coordinates of R we first find the LCA (lowest common ancestor) of x1,x2 and then looking at suitable ancestors and ancestral subtrees (as done in class or even in Q3. sub-problem 4) we update count field as needed
14 Our output will be MX once the loop breaks

Algorithm 7: Maximum overlapping rectangles at any point in $O(nlogn)$

Here algorithm written in words, because pseudo-code would be unnecessarily complex to explain

Proof of Correctness & Analysis

The fields augmented in T, do not make the insert and delete time complexity of T different from standard AVL tree insert/delete time which is O(logn) time. This is because the count field doesn't depend on any other nodes in the tree. It only increments or decrements based on its location w.r.to the nodes being inserted or deleted and hence insertion/deletion do not have to take special care of count field. (unlike size field like we saw in Q3)

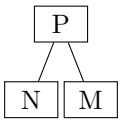
The while loop implemented in the algorithm above does 2n iterations as A contains 2n elements.
Lines 9 and 12 using method from line 13 (as we have seen in Q3), will take O(logn) time.
Line 10 happens hand-in-hand with lines 9 and 12.
Lines 8 and 11 are insert and delete operations which also take O(logn) time.
So for each iteration of while loop takes O(logn) time. Hence the while loop overall takes O(nlogn) time.
Sort in line 3 takes O(nlogn) time also.
Hence the time complexity of maximum number of rectangles that overlap at any point (maximized over all points in the plane) is O(nlogn) time.

Question 5: Synchronized Circuit [14 points]

Sub - Problem 1

Terminology used here,
FL(P,N) : final delay (after synchronization) incurred from node P to a leaf node N.
MaxDL(P) : Maximum delay which can be incurred from node P to any leaf node.
P.left.DL : Delay incurred by the left subtree of P.
P.right.DL : Delay incurred by the right subtree of P.

To establish a greedy approach, let us look at a smaller instance of the problem. Let us look at the smaller tree of two sibling leaf nodes, say N & M, and their common parent, say P.



Suppose before synchronization the delay from P to N is pn, and from P to M is pm. If this tree was synchronized, then FL(P,N) would be equal to FL(P,M). Meaning after some delay enhancement on the edges P to N and P to M,

$$FL(P, N) = pn + k_1 = pm + k_2 = FL(P, M)$$

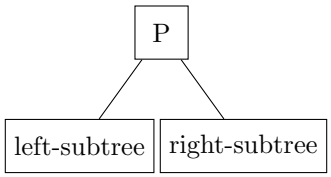
Where k_1 and k_2 are positive delay enhancements
But is it necessary that both k_1 and k_2 are needed? Also is this delay enhancement minimum? Let us look into maximum delay from P before synchronization. Suppose, $pn > pm$, then MaxDL(P) = pn. Now when doing delay enhancement, do we necessarily need to enhance pn? What if we only enhanced pm to become pn, that way synchronization is achieved as well as lower delay enhancement done. Meaning we take $k_1 = 0$ and $k_2 = pn - pm$. Hence,

$$FL(P, N) = pn = pm + (pn - pm) = FL(P, M) = MaxDL(P)$$

So we are reaching a claim here that in the optimal solution, the Max delay from node P before synchronization is equal to Final delay from node P to any leaf node after synchronization. Let us generalize this notion and then prove the claim and that doing delay enhancement this way will indeed give us the optimal solution.

Sub - Problem 2

Let P be any non-leaf node, which is not synchronized yet, with synchronized left and right subtrees.



Claim: In the optimal solution, MaxDL(P) = FL(P,N) where N is any leaf node
Proof: We will try and use contradiction.
Let pl be delay on edge P to P.left and pr be delay on edge P to P.right. We can say here that

$$MaxDL(P) = Max(P.left.DL + pl, P.right.DL + pr)$$

W.l.o.g lets say (P.left.DL + pl) > (P.right.DL + pr) then MaxDL(P) = P.left.DL + pl. Now in the optimal solution,
if MaxDL(P) ! = FL(P,N) [Assumption taken, which is contradictory to our claim]
then MaxDL(P) < FL(P,N). Meaning (P.left.DL + pl) < FL(P,N).
Let A be any leaf node in left subtree and B be any leaf node in right subtree. Now lets do some delay enhancement in order to make this tree synchronised.

$$FL(P, A) = P.left.DL + pl + k_1 = P.right.DL + pr + k_2 = FL(P, B).$$

Where k_1 and k_2 are positive delay enhancements and we also deduce $k_2 > k_1$

Question 5: continued [14 points]

Sub - Problem 2: continued

Looking at both edges of tree, total delay enhancement will be $k_1 + k_2$. But this total delay enhancement isn't minimum, as enhancement of one out of the two edges can clearly be made zero, further reducing enhancement. In doing do so, we are indirectly saying that in order to get minimum enhancement and hence to get optimal solution we need

$$k_1 = 0, k_2 = (P.left.DL + pl) - (P.right.DL + pr)$$

We can't minimize this enhancement more, as in that case k_1 would be negative. So taking this enhancement for our optimal solution, gives us that

$$FL(P, A) = P.left.DL + pl = P.right.DL + pr + k_2 = FL(P, B)$$

$$MaxDL(P) = P.left.DL + pl = FL(P, A)$$

Hence we get, $FL(P, N) = MaxDL(P)$ where N is any leaf node.
Hence here we reach a contradiction in our assumption that in optimal solution MaxDL(P) ! = FL(P,N).
Hence our claim is proved. Hence overall we obtain that in the optimal solution, for every non-leaf node P, the minimum delay enhancement $= k_1 + k_2 = k_2 = |(P.left.DL + P.left.edge.delay) - (P.right.DL + P.right.edge.delay)|$
This is an important relation we derive here.

Sub - Problem 3

Pseudo-code (we will use the relation derived above)

```
   Data: P ← root of T (given complete binary tree), Global variable: D=0
   Result: Synchronized T and Minimum Total Delay Enhancement in D
 1 Function Enhance(P):                          /* algorithm as a recursive function */
 2     if P is a leaf node then                          /* reached end of path */
 3         return 0;
 4     else
 5         edgeDelay_{P.left} ← delay on edge between P and P.left;
 6         edgeDelay_{P.right} ← delay on edge between P and P.right;
 7         /* following 2 Variables store minimum enhanced Delay incurred on left and right
              side of P and P's edge delays                                          */
 8         P.left.DELAY = edgeDelay_{P.left} + Enhance(P.left);
 9         P.right.DELAY = edgeDelay_{P.right} + Enhance(P.right);
10         /* Enhance(P.left) = P.left.DL, Enhance(P.right) = P.right.DL, from above proof   */
11         if P.left.DELAY > P.right.DELAY then          /* MaxDL(P) = P.left.DELAY */
12             edgeDelay_{P.right} = edgeDelay_{P.right} + P.left.DELAY − P.right.DELAY;  /* Enhancement */
13             D = D + P.left.DELAY − P.right.DELAY;
14             return P.left.DELAY;                                  /* return MaxDL(P) */
15         else                                          /* MaxDL(P) = P.right.DELAY */
16             edgeDelay_{P.left} = edgeDelay_{P.left} + P.right.DELAY − P.left.DELAY;    /* Enhancement */
17             D = D + P.right.DELAY − P.left.DELAY;
18             return P.right.DELAY;                                 /* return MaxDL(P) */
19         end
20     end
21     return D;                                  /* Output Minimum Total Delay Enhancement */
22 End Function
```

Algorithm 8: Minimum Total Delay Enhancement to get Synchronization

Analysis

The recursive function Enhance is called by each of the n nodes. And in each function call O(1) is the time taken. Henc the time complexity of this algorithm is O(n) where n = $|T|$.