IIT Kanpur

CS345A - AlgorithmsII

Assignment 2

Anjani Kumar
11101

Sumedh Masulkar
11736

January 22, 2014

Contents

1 Multi-Swap in dynamic sequence

Overview

The data structure used for saving the sequence will be an augmented red black(height balanced) binary tree. The design of the tree would be such that inorder traversal of the tree anytime would output the sequence.

The data structure will perform the following operations, all in $O(log\ n)$:

- Insert(D, i, x).
- Delete(D, i).
- Multi-Swap(D, i, j).
- Report(D, i).

Augmentation of binary tree (Every node will store these keys in addition to the standard keys kept in nodes of a red-black binary tree):

- **flip bit:** This variable will act as a flag to tell whether the subtree of this node are to be flipped or not. If $flip$ is 1, this means the subtrees of this node are flipped, *i.e.* left child anywhere in the subtrees of this node will actually be the right child, and similarly the right child will be the left child of its parent. If $flip$ is 0, then the relationship between the parent and children nodes will be normal(as always).
- **size:** This variable stores the size of a subtree at a given node. The size of the subtree means numbers of nodes in the subtree. Hence, size at a given node will be (1 + size of left subtree + size of right subtree).
- Other standard fields:
  - val(u): value of the node.
  - left(u): pointer to left child of the node.
  - right(u): pointer to right child of the node.
  - parent(u): pointer to parent of the node.
  - color(u): color of the node.

1.1 Insert(D, i, x)

Insert an element x at i^{th} position in the sequence.

Insert operation in this data structure will be according to the size variable and flip bits. Traversal of the tree during insertion will be according to the flip bit. If number of flips encountered in the way are odd, then the left and right child will have to be considered as reversed, if number of flips are even, then we will traverse normally. This means if a node has to be flipped, its left child will be considered as right, and vice versa.

Position of insertion will be found according to the size variable.
Suppose, we are at a node u. Let s be the size of left subtree of u, and i is the position where x is to be inserted. If $i \leq s + 1$, then x should be inserted in left subtree of u. Otherwise, if $i > s$, then x will be inserted in the right subtree of u.

Time Complexity:
For any insertion, we will be traversing from root to leaf, thus taking $O(h)$ time. Hence, time complexity for this operation will be $O(log\ n)$.

Algorithm 1: Pseudo code for insert operation

```
1  Initialize flips← 0;          //this variable flips is global, where flip(T) refers to flip
                                   //bit of that node.
2  Insert(T, i, x){
3  begin
4      flips ← (flips + flip(T))%2;
5      size(T)←size(T)+1;
6      if T==NULL then
7          create a new node u;
8          val(u)← x; flip(u)←0;
9          left(u)←NULL; right(u)←NULL;
10         return u;
11     else if flips==0 then
12         if left(T)==NULL then
13             s←0;
14         else
15             s← size(left(T));
16         if i ≤ s + 1 then
17             left(T)← Insert(left(T), i, x);
18         else
19             right(T)← Insert(right(T), i − s − 1, x);
20         return T;
21     else
22                 //flip is 1.
23         if right(T)==NULL then
24             s←0;
25         else
26             s← size(right(T));
27         if i ≤ s + 1 then
28             right(T)← Insert(right(T), i, x);
29         else
30             left(T)← Insert(left(T), i − s − 1, x);
31         return T;
32
33 }
```

1.2 Delete(D, i)

Delete i^{th} element from the sequence.

Delete operation also will be similar to Insert() operation.
The flip bits along the path if even, then traversal will be normal else if number of flips encountered are odd, the right child will actually be the left child and vice-versa. The i^{th} element will be found similarly with the help of the size variables stored in the nodes.

- **Remove():** Function Remove() will be used as a black box in Delete() Operation. This Remove() function is the standard deletion of a node in a Red-Black tree, *i.e.* the node will be removed and height balancing will be done as usual. The size fields and flip bits will also be updated accordingly. This operation has time complexity $O(log\ n)$.

Time Complexity:
For any deletion, we will be traversing from root to leaf, thus taking $O(d)$ time, where d is depth of tree. Then, Remove() which is normal deletion in a red-black tree has time complexity $O(log\ n)$. Hence, time complexity for this operation will be $O(log\ n)$.

Algorithm 2: Pseudo code for Delete operation

```
1  Initialize flips← 0;          //this variable flips is global, where flip(T) refers to flip
                                   //bit of that node.
2  Delete(T, i){
3  begin
4      flips ← (flips + flip(T))%2;
5      size(T)←size(T) - 1;
6      if flips==0 then
7          if left(T)==NULL then
8              s←0;
9          else
10             s← size(left(T));
11         if i < s + 1 then
12             left(T)← Delete(left(T), i, x);
13         else if i > s + 1 then
14             right(T)← Delete(right(T), i − s − 1, x);
15         else
16             Remove(T);
17         return T;
18     else
19                 //flip is 1.
20         if right(T)==NULL then
21             s←0;
22         else
23             s← size(right(T));
24         if i < s + 1 then
25             right(T)← Delete(right(T), i, x);
26         else if i > s + 1 then
27             left(T)← Delete(left(T), i − s − 1, x);
28         else
29             Remove(T);
30         return T;
31 }
```

1.3 Report(D, i):

Report i^{th} element from the sequence.

Traversal will be similar to Insert() and Delete() operations.
If flip is 1, consider tree to be flipped. Else traverse normally.
And for selecting the direction, check the size of left subtree. If size is smaller than i, go to right subtree, else traverse left subtree.

Time Complexity:
For any report, we will be traversing from root to leaf, thus taking $O(h)$ time. Hence, time complexity for this operation will be $O(log\ n)$.

Algorithm 3: Pseudo code for Report operation

```
1  Initialize flips← 0;          //this variable flips is global, where flip(T) refers to flip
                                   //bit of that node.
2  Report(T, i){
3  found←false; u ← T;
4  while not found do
5      flips ← (flips + flip(T))%2;
6      if flips==0 then
7          if left(u)==NULL then
8              s←0;
9          else
10             s←size(left(u));
11         if s== i-1 then
12             found←true;
13         else if s > i-1 then
14             u ← left(u);
15         else
16             u ← right(u);
17             i ← i -s -1;
18
19     else
20         if right(u)==NULL then
21             s←0;
22         else
23             s←size(right(u));
24         if s== i-1 then
25             found←true;
26         else if s> i-1 then
27             u ← right(u);
28         else
29             u ← left(u);
30             i ← i -s -1;
31
32 return val(u);
33 }
```

1.4 MultiSwap(D, i, j)

Swap all elements from i^{th} place to j^{th} place. For example, if the sequence is (x, a, e, b, f, h, z, d). Then, after MultiSwap(D, 3, 7), the sequence becomes (x, a, z, h, f, b, e, d).

The algorithm for MultiSwap() uses two other following operations as black boxes:

1. **Merge(T_1, T_2):** Given two trees T_1 and T_2 having the same height h, we can find an element x and the added constraint that all elements of T_1 are smaller than x, and all elements of T_2 are larger than x, merge these into a single red-black tree. This operation can be done in $O(log\ n)$ time, as we have seen earlier in CS210 course.

2. **Split(T, x):** This operation is just the opposite of Merge() described above. The aim is to split T into two red-black trees T_1 and T_2 such that T_1 stores all the elements of T which are smaller than x and T_2 stores elements of T greater than x. Using $O(log\ n)$ time algorithm for Merge() just as a black box, it is possible to perform Split(T, x) in $O(log\ n)$ time, as was implemented in CS210.

3. **Height(T):** Returns black height of the tree rooted at T. Takes $O(log\ n)$ since black height from root to any leaf will be same as any other. Thus, this would be as simple as going to the left of node, while left node is null and incrementing height variable.

What needs to be done for MultiSwap(T, i, j)?

1. Let T be the tree which stores sequence S. Note that inorder traversal of this tree gives sequence S. We will have to keep in mind the flip bits stored at various nodes to get the correct sequence from T.

2. Split the tree T into T_1 and T' such that T_1 stores the first i-1 elements, and T' stores the rest.

3. Split T' further into T_2 and T_3, such that T_2 stores next j-i+1 elements of the sequence after elements in T_1.

4. The operations till now have been such that all the elements to be swapped are contained in T_2.

5. Reverse flip bit of root of T_2,*i.e.* flip(T_2)$\leftarrow$ *not*(flip(T_2)).

6. Merge T_1 with T_2 to get T', and then T_3 with T' to get T. Thus, the sequence has been swapped.

7. Note that for each operation of split and merge, we need to take care of flip bit stored in respective nodes (or roots). It takes $O(log\ n)$ time to split a height balanced trees around any element and $O(log\ n)$ time for merging two height balanced trees of combined size n.

8. Point to note: While Merging, if black height of T_1 is greater than that of T_2, then root of T_2 will be successor of root of T_1, hence T_1 doesn't need to be flipped. But if black height of T_2 is greater, then T_1 will be successor of T_2, now since T_2 is flipped, T_1 will also get flipped. To avoid this, we should also flip T_1. Thus resetting flip for nodes in T_1. Similar method should be adopted while merging T' and T_3.

Algorithm 4: Pseudo-code for MultiSwap operation

```
1  MultiSwap(D, i, j){
2  begin
3      x ← Report(T, i);                    //O(log n)
4      T1, T' ← Split(T, x);                //O(log n)
5      y ← Report(T', j − i + 1);           //O(log n)
6      T2, T3 ← Split(T', y);               //O(log n)
7      flip(T2) ← not(flip(T2));            //where not(0)=1, not(1)=0.
8      h1 ← Height(T1);                     //O(log n)
9      h2 ← Height(T2);                     //O(log n)
10     h3 ← Height(T3);                     //O(log n)
11     if h1 > h2 then
12         T' ← Merge(T1, T2);              //O(log n)
13     else
14         flip(T1) ← not(flip(T2));
15         T' ← Merge(T1, T2);              //O(log n)
16     h ← Height(T');                      //O(log n)
17     if h < h3 then
18         T ← Merge(T', T3);               //O(log n)
19     else
20         flip(T3) ← not(flip(T3));
21         T ← Merge(T', T3);               //O(log n)
22     return T;
23 }
```

Time Complexity:
As can be seen in the algorithm, all steps have maximum $O(log\ n)$, and each step is carried out only once. Thus, time complexity of the operation is $O(log\ n)$.

2 Point of Maximum Overlap: A Model Solution

We are using an Augmented Red Black Tree.

2.1 Fields stored in a node:

- **point**: x coordinate of the node.
- **identifier**: +1 for a start point and -1 for an end point.
- **overlapCount:** $\sum_{i \in subtree\ of\ x} (identifier(i))$
  overlapCount(x) = overlapCount(left(x))+overlapCount(right(x))+identifier(x)
- **max:** stores the maximum count of overlaps in the subtree of node.
  max = $max_{\{j=1,2,3,...\}} \sum_{i=1}^{j} (identifier(i))$
- **pointMax:** Point of maximum overlap in the subtree of x.

2.2 Brief Description:

We are using Augmented Red Black Tree with fields mentioned above, to find the point of maximum overlap. For n intervals, the tree contains 2n nodes with the start point given the token 1 and end point is given -1. We do a line sweep from left to right and take the sum of these tokens from 1 to j for each endpoint. The point upto which this sum is maximum will be the point of maximum overlap.

2.3 Functions:

- **Update:**
  - Update is used to update the fields after insertion, deletion and rotation due to height balancing.
  - Assuming the augmented fields for children of a node(let it be x) has been calculated. Then the values in x will be:
    1. overlapCount: overlapCount(x) = overlapCount(left(x))+overlapCount(right(x))+identifier(x)
    2. max:

$$max(x) = max \begin{cases} max(left(x)); \\ \qquad\qquad \textbf{if maximum is in the left sub tree} \\ overlapCount(left(x)) + identifier(x); \\ \qquad\qquad \textbf{if maximum is the point x} \\ overlapCount(left(x)) + identifier(x) + max(right(x)); \\ \qquad\qquad \textbf{if maximum is in the right subtree} \end{cases}$$

3. pointMax:

$$pointMax(x) = \begin{cases} pointMax(left(x)); & \textbf{if maximum is in the left sub tree} \\ x; & \textbf{if maximum is the point x} \\ pointMax(right(x)); & \textbf{if maximum is in the right subtree} \end{cases}$$

– Time Complexity for Update:
In order to compute field values, only the values of children are needed. Therefore each call to update function takes only $O(1)$ time.

- **Insert:**
  - In order to insert an interval, insertion of start point is done before the end point of the interval.
  - A new node is inserted normally in the Red Black Tree T and after its insertion, the fields of the nodes lying on the path from the inserted node to the root are updated in a bottom-up fashion.

Algorithm 5: insert(T, x, id)

Input: A Pointer to a node of the tree,x coordinate of point,id to differentiate between start and end point

```
1  if T = NULL then
2      point(T) ← x;
3      identifier(T) ← id;
4      overlapCount(T) ← id;
5      max(T) ← 1;
6      left(T) ← NULL;
7      right(T) ← NULL;
8      return T;
9  else
10     if x<value(T) then
11         insert(left(T),x,id);
12     else if x>value(T) then
13         Insert(right(T),x,id);
14     else if id = 1 then
15         insert(left(T),x,id);
16     else
17         insert(right(T),x,id);
18     Update(T);
19     return T;
```

- Time Complexity for Insert:
  For every interval, 2 insert operations are performed and for each point, traversal of RBT T takes $O(log\ n)$ time, where n is the number of intervals. Every update operation takes $O(1)$ time. Therefore overall time complexity is $O(log\ n)$.

- **Delete:**
  - In order to delete an interval, the start point and end point are deleted separately.
  - While searching for the node to be deleted, we match its value as well as the id(+1 or -1).
  - The node is then swapped with its predecessor and then deleted.
  - After the deletion, fields of nodes that lie in the path from the node's and its predecessors parent to the root, are updated in bottom-up fashion.
  - **search(T,x,id):** It returns the node such that point(node) = x and identifier(x) = id.

Algorithm 6: Delete(node,id)

```
   Input: The node to be deleted,id
 1 node ← search(T, x, id);
 2 if node is a Leaf Node then
 3 |   u ← (Node → parent);
 4 |   Delete node;
 5 |   while u≠ ROOT do
 6 |   |   update(u);
 7 |   |   u ← (u → parent);
 8 |   update(ROOT)
 9 else
10 |   u ← (Node → parent);
11 |   temp ← (Node → predecessor);
12 |   v ← (temp → parent);
13 |   swap(temp,Node) ;
14 |   Delete temp ;
15 |   while u≠ ROOT do
16 |   |   update(u);
17 |   |   u ← (u → parent) ;
18 |   while v≠ ROOT do
19 |   |   update(v);
20 |   |   v ← (v → parent);
21 |   update(ROOT);
```

– Time Complexity for Delete:
Deletion of an interval takes 2 Delete operations, one for each end point. Traversing in the RBT T to search for points takes $O(h)$ time where h is the height of T. Updating in Bottom-up fashion from the nodes take $O(d)$ time, where d is the depth of node. Therefore overall time complexity is $O(h) = O(log\ n)$ where n is the number of intervals.

- **Height Balance Rotation:**

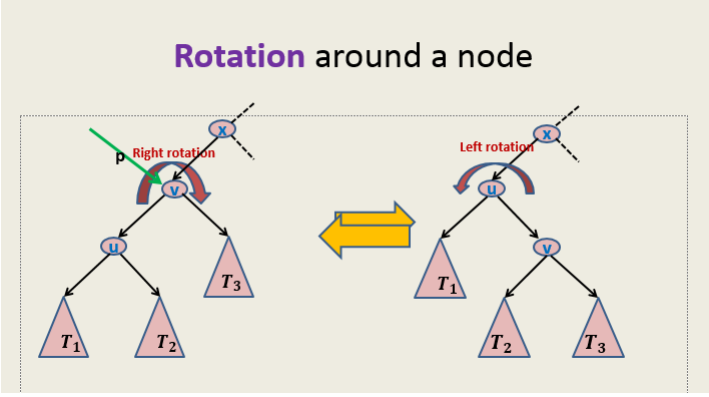

Figure 1: Rotation

– During Rotation, only 2 nodes are affected and rest of the subtree's fields will remain the same as shown in figure-1.To update their values, update function is called for the 2 nodes u and v. Hence it takes $O(1)$ time.

- **MaxOverlap:**

– At any node v, the Augmented field pointMax(v) stores the point with maximum overlap in the subtree of v. Therefore pointMax(ROOT) will give the point with maximum overlap in the entire interval tree. This can be achieved in $O(1)$ time.