Chapter 8

Data Structures

8.1 segment trees

8.2 interval trees

8.3 Fenwick Trees

Consider a problem which takes arbitrary values in a list and requires us to do all of the following:

1. Compute the sum of all the values between any pair of indices
2. Update any values of the array
3. Remove values from any point in the array
4. Perform any of these operations in sub-linear incremental time
5. Don't use more than linear memory
6. Don't use quadratic time preprocessing

A prefix-sum, which would allow us to compute the sum of (i, j) as `prefix_sum(y)-prefix_sum(i)`, it would take linear time to update or remove any value. The raw list enables us to update and remove any value, but takes linear time to compute any range query.

In many cases where we need sub-linear time for seemingly competing operations, a typical approach is to use a tree to reduce each of the competing operations to logarithmic time. The approach is also correct here, where a *Fenwick tree* provides the necessary structure to perform all operations in incremental logarithmic time.

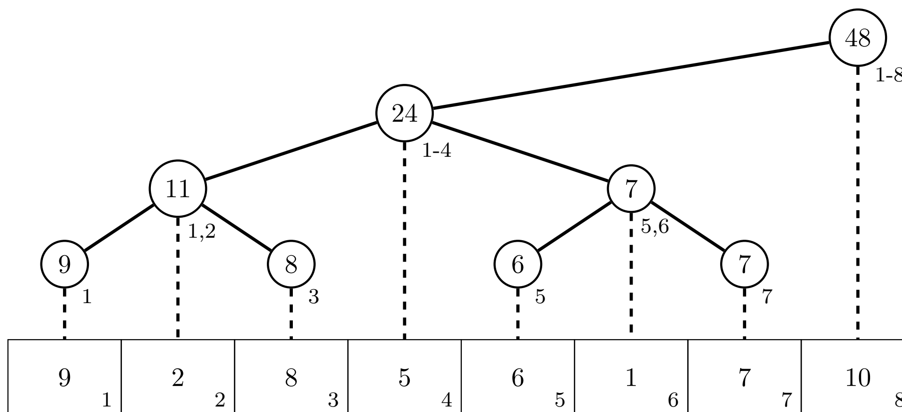
8.3.1 Idea

Fundamentally, the Fenwick tree allows us to compute a prefix sum, or modify an element, in logarithmic time. This enables querying individual values (by subtracting the prefix sum of consecutive elements), and removal(by updating the value to 0).

From a high level, we will construct a binary tree with the following rules:

- An in-order traversal of the tree maps to the indices of the list
- Each node contains the sum of all elements in its left subtree and the value representing the node itself

Lets see this in action with a tree mapping a corresponding array. The dashed lines indicate the node corresponding to each element, and the small labels on each node represent the array elements that comprise that node. As with the definition, this will be the index of the node itself and all indices of the left subtree.



To be perfectly clear where these values come from, we will walk through the list and explain.

1. There is no subtree, so this node contains only the value of the corresponding array element
2. The sum of all the elements corresponding to the left subtree is 9 (just the first elemenrt), and the value of the corresponding array element is 2. $9 + 2 = 11$
3. There is no subtree, so this node contains only the value of the corresponding array element
4. The left subtree contains 3 elements, 9, 2, and 8, summing to 19. We then add 5 for ourselves. $19 + 5 = 24$

5. There is no subtree, so this node contains only the value of the corresponding array element

6. The sum of all the elements corresponding to the left subtree is 6 (just the fifth element), and the value of the corresponding array element is 1. $6 + 1 = 7$

7. There is no subtree, so this node contains only the value of the corresponding array element

8. The left subtree contains all the elements of the list but for the 10. We add the 10 for ourselves, and are left with 48.

Hopefully it is clear how we have obtained these values based on the description of the construction.

1. Compute a sum starting with the value at the node representing the element you want the prefix sum for.

2. Recurse up the tree

   - If you reached a node from the left subtree, then add nothing. The elmenets comprising this node overlap with the sum we have already computed, and contain further nodes from that left subtree that we do not wish to add in.
   - If you reached a node from the right subtree, then the value at this node includes all the values from the left subtree, none of which we have yet added to our sum. Add this value.

As we progress up the tree, we add in larger and larger chunks of the prefix sum, ensuring at every level we have added in all the necessary values for the subtree rooted at the node we reached. As the tree is balanced, this operation is logarithmic.

Lets walk through a specific example, computing the prefix sum at $i = 5$.

1. The node we start at contains 6. We begin with this as our sum. $s = 6, v = (5)$

2. We recurse to node 6. We arrived from the left subtree, so we do nothing. This is natural as this node covers the values $i = (5, 6)$, one of which our sum has already included,and the other of which is out of range for the sum we are trying to compute. $s = 6, v = (5)$

3. We recurse to node 4. We arrived from the right subtree, so we add the 24. This node covers the values $i = (1, 2, 3, 4)$, all of which we need. $s = 30, v = (1, 2, 3, 4, 5)$

4. We recurse to node 8. We arrived from the left subtree, so do nothing. As with step 2, we either have already included all values covered by this node, or they are out of range. $s = 30, v = (1, 2, 3, 4, 5)$

5. We have reached the root, so return the proper sum of 30. $9+2+8+5+6 = 30$

Using the same understanding of which elements a given node encompasses, we can update using a similar method. When we reach a node whose some should include our value, we modify it accordingly. This means that when we reach a node from the left subtree, we update (as the invariant is that the value at a node must represent the left subtree), but we do not if we reached from the right subtree. As we compute this in a single pass up the tree, it is also logarithmic in time.

Given that we can perform updates in linear time, we can trivially construct a fenwick tree from scratch by initializing a tree of the appropriate size to all 0's, then iteratively "updating" each element of the list in turn to its actual value (each requiring a walk up the tree.

TODO: add a graph with indices. add a graph indicating which elements a node "covers". Add the bar graph representation.

8.3.2 Implementation

While one could construct the graph for the tree as described, the construction is such that this is unnecessary. To see why, lets look at which nodes end up added in order to compute the prefix sum for all elements. We will look at both the decimal and binary representations.

index	included nodes	binary index	binary included nodes
1	1	0001	0001
2	2	0010	0010
3	3,2	0011	0011,0010
4	4	0100	0100
5	5,4	0101	0101,0100
6	6,4	0110	0110,0100
7	7,6,4	0111	0111,0110,0100
8	8	1000	1000

While we may not notice a particular pattern in the decimal representation, there is a clear pattern in the binary. Namely, once we add a node, the next node we add is found by converting the least-significant 1's bit to a 0. This leads to the following algorithm

```
// assume the tree is stored in an array indexed by the nude number
void query(int e){
  int ans=0;
  for(int i=0;i<31;i++){ // iterate over all bits
    if(e&(1<<i)!=0)ans+=ft[e]; // add the value at the node if this
        bit is a 1
    e&=~(1<<i); // clear this bit
  }
}
```

TODO: bithacks; e&=e-1 zeros the last 1 bit. removing need for most of the work

Because we index into this array based on the binary representation of a number, it is also called a *Binary Index Tree*

8.4 prefix trees

8.4.1 Description

8.4.2 Use Cases

8.4.3 Linear-time Construction

8.5 Trie