Competitive Programming 2

Week 2: Special Trees

Announcements

- We’ve updated the first week’s attendance on the Blackboard.
    - Each student has got 45 point for the first week’s attendance.
- Sign the attendance sheet before you leave!
    - Attendance will be graded from this week

Content

- Segment Tree
  - Build
  - Query
  - Update
  - Lazy Propagation
- Fenwick Tree
- Trie

Segment Trees - Motivation

- Suppose we have an array
  - A = [5, 6, 1, 25, -2, 105, 0]

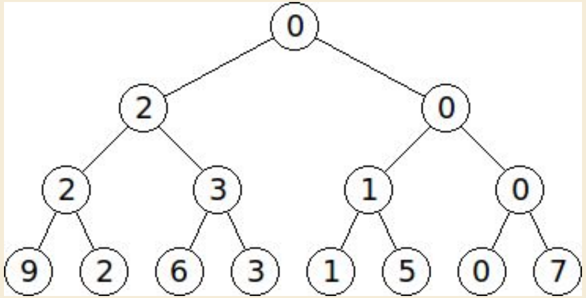
- We want these operations
  - min(i, j) - get the smallest element between position i and j
  - sum(i, j) - get the sum of elements between position i and j
  - update(i, x) - update the value of element i to x
  - add(i, j, x) - add value x to elements between position i and j

Segment Trees

- Supports Range Queries in O(log n)
- Supports Range Updates in O(log n) with lazy propagation
- Builds in O(n) time

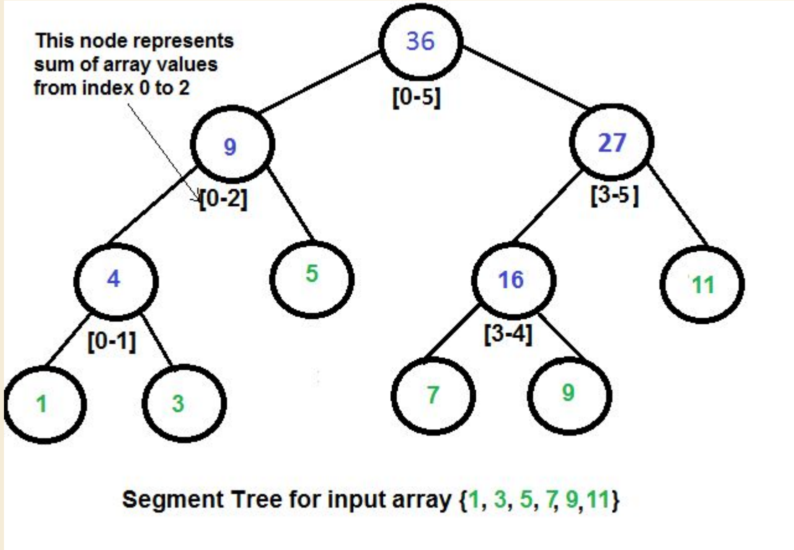
High Level Overview

- Usually build recursively
- Leaf nodes are the original array values
- Intermediate values are sums, mins, etc
- We'll just talk about RMQ (Range Min Query) from this point on, but others are similar



High Level Overview

- Does not have to be a perfect tree



High Level Overview

- When building, we start with a segment [0...n-1], then divide in half and recurse down both sides, for each segment storing the min

- Note that root node stores min for [0...n-1].
    - Left child of root stores min for [0…(n - 1)/2]
    - Right child of root stores min for [(n - 1)/2 + 1...n - 1]
    - ...

- Number of nodes in the worst case is ~2n
    - Tree height is O(logn)

Building the Segment Trees

- Can easily do recursive

- From the bottom up, fill in the leaves. Then, move up the tree and calculate the parent of two nodes a and b as min(a,b), and keep moving up the tree.

- We can easily do this in O(n)

Low level overview

- Stored as an array starting from index **1**. The size of this array is ~4n.
- At each node, left child is 2*i, right child is 2*i + 1. Invoke Build(1, 0, n - 1)

```
def Build(p, l, r)
    if l == r then
        st[p] ← A[l]  // A is starting from index 1.
        return
    mid ← (l + r) / 2
    pl ← 2 * p
    pr ← 2 * p + 1
    Build(pl, l, mid)
    Build(pr, mid + 1, r)
    st[p] ← min(st[pl], st[pr])
```

Query

- The input is two numbers L and R. In O(logn), we can calculate the minimum element of segment A[L..R].

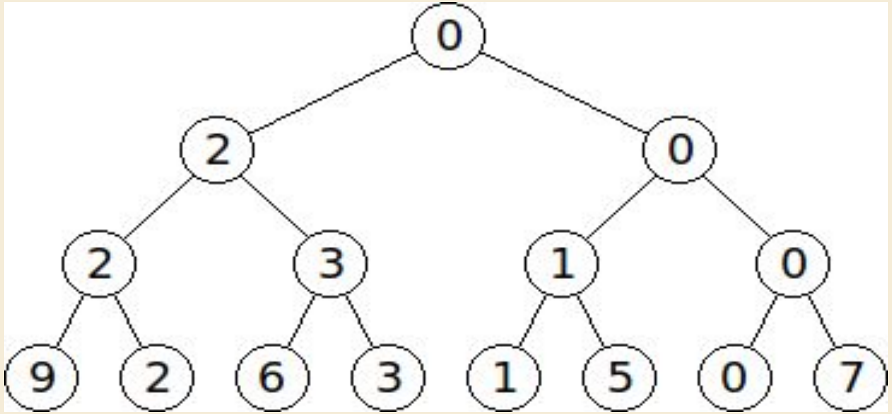
- Recursively:
  - If your range is within the segment completely, return the value at that node.
  - If it is in one of the children, query on that child
  - If it is both children, query both

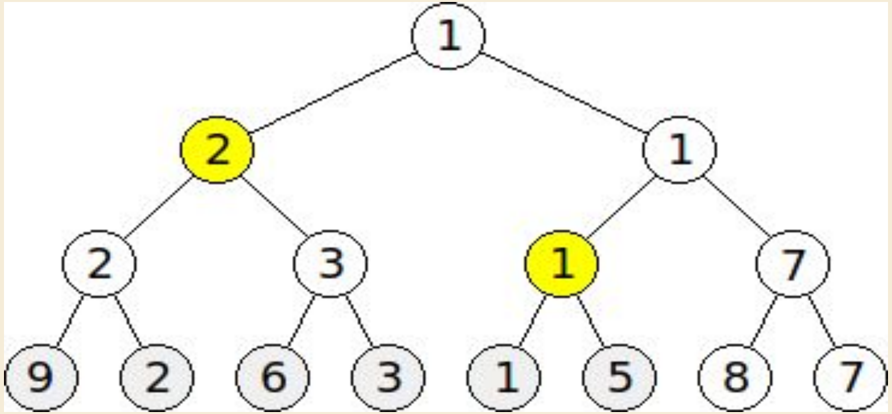
Query

```
def query(p, l, r, L, R) // Gets min element within [L, R]
    if L <= l and r <= R then // point is within range
        return st[p]

    mid ← (l + r) / 2
    pl ← 2 * p
    pr ← 2 * p + 1
    ret ← +inf

    if L <= mid then
        ret ← min(ret, query(pl, l, mid, L, R))
    if mid < R then
        ret ← min(ret, query(pr, mid + 1, r, L, R))
    return ret
```

Query example

- Finding the RMQ in the range [0...5], we just need to traverse and find two nodes in O(2*logn) then return min(2, 1)





Updates

- Remember that we said segment trees can efficiently answer dynamic range queries.

- This means that if the array on which we are performing RMQs changes, we can efficiently update the segment tree.

- If an element in the array changes, we start from the leaf node representing that element and move up the tree, updating nodes as we go.

- Thus, this takes O(log n) time.

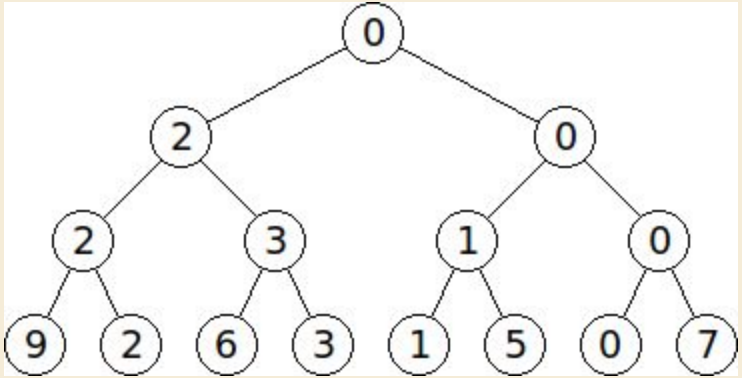
Updates

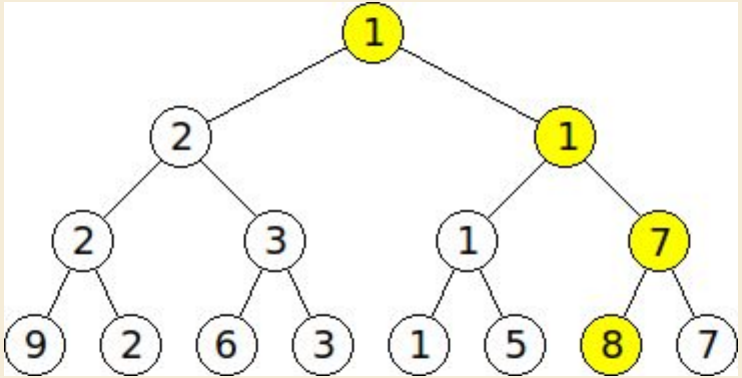
```
def update(p, l, r, idx, num)
    if l == r then
        st[p] ← num
        return
    mid ← (l + r) / 2
    pl ← 2 * p
    pr ← 2 * p + 1
    if idx <= mid then
        update(pl, l, mid, idx, num)
    else
        update(pr, mid + 1, r, idx, num)
     st[p] ← min(st[pl], st[pr])
```

Updates - Example

- Let’s say we wanted to update the original 0 (index 6) to 8.

- We only need to update nodes on the path from the leaf to the root, so log(n) time





Code example

https://gist.github.com/husseincoder/4369150#file-segment_tree-cpp

Lazy Propagation

- Allows you to perform range updates much faster
- Basically, you store updates in a separate array, effectively postponing them
- Only perform those updates on the tree when you need to
- http://www.geeksforgeeks.org/lazy-propagation-in-segment-tree/

Lazy Propagation

```
def add(p, l, r, L, R, x)
     if L <= l and r <= R then
          st[p] ← st[p] + x
          add[p] ← add[p] + x      // put a mark on this node
          return
     pushdown(p, l, r)     // pushdown is to propagate lazy mark into children’s nodes
     mid ← (l + r) / 2
     pl ← 2 * p
     pr ← 2 * p + 1
     if L <= mid then
          add(pl, l, r, L, R, x)
     if mid < R then
          add(pr, l, r, L, R, x)
     st[p] ← min(st[pl], st[pr])
```

More on Segment Trees

- http://codeforces.com/blog/entry/18051
- http://wcipeg.com/wiki/Segment_tree

- Covers the iterative, low LOC way of doing segment trees

- Possibly better for contests, as it is faster to type

- Downside is Lazy Propagation is slightly harder, and it's much less intuitive

Fenwick Trees

- Data structure that allows efficient querying/updating of prefix sums / maximum

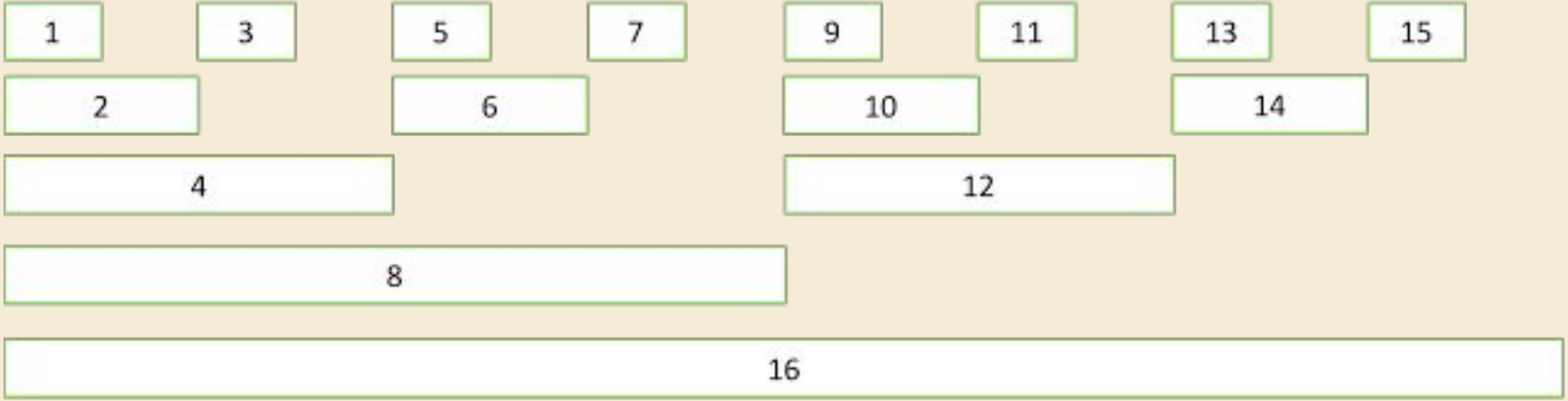
- sum(i) - get the sum of elements between position 1 and i
  - log(N)
- update(i,x) - update element at position i with x
  - log(N)

- Builds in O(n) time

- EASY TO CODE
  - Harder to understand

Fenwick Trees

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
1	2	1	4	1	2	1	8	1	2	1	4	1	2	1	16



High level pseudocode

sum(i):

Set the counter to 0.

Repeat the following while $i \neq 0$:

Add the value at node i into counter

Clear the rightmost 1 bit from i. $(i \leftarrow i - (i \& (-i)))$

High level pseudocode

update(i, x):

Write out node i in binary.

Repeat the following while i <= length:

Add x into node i.

Add the rightmost bit to i. (i ← i + (i & (-i)))

Building the Tree

Loop through the n array items in increasing index order, always adding the sum only to the next smallest index that it should be added to, instead of to all of them:

```
for i = 1 to n:
    j = i + (i & -i)
    # Finds next higher index that this value should contribute to
    if j <= n:
        x[j] += x[i]
```

Code example

http://petr-mitrichev.blogspot.com/2013/05/fenwick-tree-range-updates.html

More Information

- https://www.topcoder.com/community/data-science/data-science-tutorials/binary-indexed-trees/
- http://bitdevu.blogspot.com/
- http://cs.stackexchange.com/questions/10538/bit-what-is-the-intuition-behind-a-binary-indexed-tree-and-how-was-it-thought-a

Trie

Dictionary:

the
a
there
by
their
answer
bye
any

```
      root
   /   \    \
   t   a     b
   |   |     |
   h   n     y
   |   |  \  |
   e   s  y  e
/  |   |
i  r   w
|  |   |
r  e   e
       |
       r
```

Trie

- Key Insert and Search time cost : O(len(key))

- Memory requirement
  - Let S be the number of characters in the dictionary
  - Let c be the number of characters in the set. E.g. all lowercase English letters: c = 26.
  - O(S * c)

```
// Trie node

struct Node {

     Node *children[ALPHABET_SIZE];  // ALPHABET_SIZE is the number of characters in the scope

     bool isEndOfWord; // True if the node represents end of a word

}root;
```

Insert

```
def insert(node: Node, key: str, value: Any) -> None:

    for char in key:

        if char not in node.children:
            node.children[char] = Node()
        node = node.children[char]

    node.isEndOfWord = true
```

Search

```
def find(node: Node, key: str) -> Boolean:

    for char in key:
        if char in node.children:
            node = node.children[char]
        else:
            return False

    return node.isEndOfWord
```