Advanced Data Structures

Alex Chen

October 28, 2011

1 Introduction

A common problem that comes in many different forms is the "range query" problem—a problem consisting of repeated queries describing some range of data. The data structures used to solve this problem have many applications, such as in inversion counting. Of course, there are many data structures that can be considered "advanced," and there is an entire Wikipedia article that you can read if you want to learn about all the data structures, but this lecture will just cover a few. Range query data structures are definitely less common and useful than stacks, queues, sets, heaps, and trees, but will nonetheless appear every once in a while in USACO Gold.

2 The Range Query Problem

The most basic range query problems appear in this basic form:

1. You are given a list of cows with some characteristic.
2. You get some large number of queries of these two forms:
   - Change some characteristic of a cow.
   - Find some aggregate property of a consecutive group of cows: a "range" of cows.

In other words, you are given a list of cows and a series of *many* commands that you have to perform in order. One type of command asks for information. The other type changes the list of cows.

2.1 Sample Problem

Problem: There are N cows ($1 \leq N \leq 100,000$), each with a happiness value H_i ($1 \leq H_i \leq 1,000,000,000$). The cows are numbered from 1 to N. Farmer John has Q queries ($1 \leq Q \leq 100,000$). There are two types of queries. For queries of type A, print the sum of the happiness values of all cows from the given left endpoint to the given right endpoint. For queries of type B, change the happiness value of cow i to a different value. Solve the queries in the order provided.

2.2 Naive Solution

For each query, traverse all the cows mentioned in the query and find the sum of their happiness values. Each query can take up to $O(N)$ to process, so the total runtime is $O(QN)$. This is too slow.

2.3 Better Solutions

There are several very good solutions that are beyond the scope of this lecture. They will not be described in detail to allow you to think about them more yourselves or do your own research.

- $O(N + Q)$: use prefix sums. This only works for the range *sum* query problem, though. In addition, prefix sums are not dynamic, so they will fail for the second type of query.

- $O(N + Q\sqrt{N})$: store information about the list in segments of size $\sqrt{N}$.
- $O(N \log N + Q)$: create a **sparse table**, which is an $N \times \log N$ table where the item in row i and column j is the sum (or whatever else you are looking for) of the first 2^j elements starting at i. Notice how row i can easily be generated from row $i - 1$. This method is useful because queries can be performed in constant time.
- $O((N + Q) \log N)$: use a **binary-indexed tree**. This data structure is mostly useful for dynamic sum queries.

3 Augmented Binary Tree

Note: these are **not** actually range trees. Range trees are something else, despite what many people might think. Yes, this is especially weird since you would think the *range* tree is useful for the range query problem. This is not exactly a segment tree either, and more of *an augmented binary tree*. A segment tree is a more general version of what we will be using.

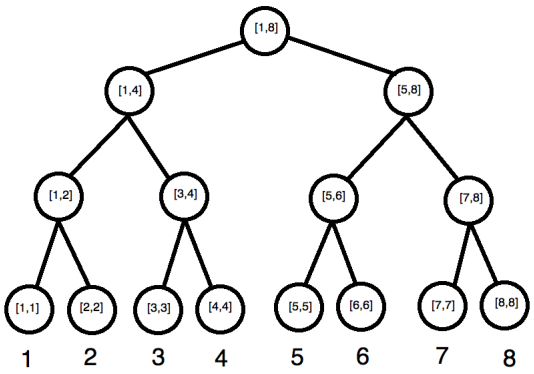
We can build a *tree atop an array*, essentially forming an augmented binary tree, to create a data structure that will allow us to dynamically update and query information about ranges of data.

An augmented binary tree for range queries can be defined recursively as follows, given a list of elements from 1 to N:

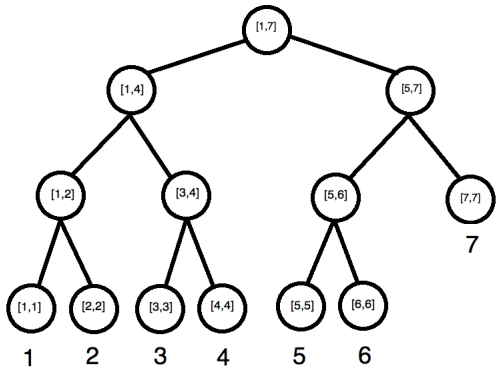
- Each node has (at least) five properties: left endpoint, right endpoint, left child, right child, and *value*.
- Node 1 contains the sum of elements 1 to N.
- A node containing information about the interval from i to j, if $i < j$, has a left child for $[i, \frac{i+j}{2}]$ and a right child for $[\frac{i+j}{2} + 1, j]$.
- Calculate the value for each node by considering the values of its children.

With this formulation, we get a tree where the leaves each represent a single element of the list and all other nodes contain children that split their intervals approximately in half. The power of the range tree is the ability for each parent node to easily compute its value by considering only two other nodes. We could also define an augmented binary tree bottom-up.

Let's consider some examples: when the number of elements in the list is a power of 2, then we end up with a perfect-looking binary tree:



Sometimes, it's not as pretty.



How do we use this to solve the summation problem? We let the *value* of each node be the sum of all the list values within its range. The values of all the children are equal to their list value, and we can calculate the values of all parent nodes by summing the values its two children.

3.1 Querying

How do we use this augmented binary tree once we have built it? Consider the query $[i, j]$. We can find the total sum of at most two nodes, and sometimes only one. While not every range is represented by a node in the graph, each range is the composite of at most two nodes. For example, if we wanted to find the sum of $[3, 4]$, we can simply find the $[3, 4]$ node in the graph. If we wanted to find the sum of $[3, 8]$, we would have to find the sum of the $[3, 4]$ node and the $[5, 8]$ node. This can be coded with **recursion**. Here is pseudo-code:

```
def query(node, left, right):
    if node's range does not overlap with [left, right]:
        return 0
    else if node's range is within [left, right]:
        return the value of the node
    else
        return query(left child, left, right) + query(right child, left, right)
```

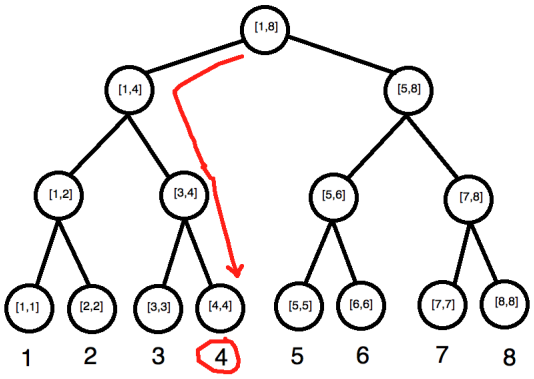
Complexity: $O(\log N)$. This is not immediately obvious, because a query can sum up to $O(\log N)$ different values. That is, the query can stop at up to $O(\log N)$ different nodes. As an example, consider the binary tree on a list of length 64. If we want to query the range $[2, 63]$, we will have to check a lot of nodes. Each node takes $O(\log N)$ to reach. Does this mean that a single query can take up to $O(\log^2 N)$? **No!** It is faster. A query takes worst case $O(\log N)$ time because the *total* number of nodes we have to process is logarithmic in N. See this **proof**.

- Case 1: When querying at a node, we only recur to one of its children. In this case, the total number of nodes we must process is 1 more than the number of nodes we have to process in the subtree. Because the height of the tree is at most $O(\log N)$, this step cannot add more than $O(\log N)$ nodes. Then, to figure out the runtime complexity of querying the subtree, we "recur" and consider these two cases again.
- Case 2: When querying at a node, we recur to both of its children because the range extends on both sides. In this case, consider the deepest node we must process on the left side and the deepest node we must process on the right side. The paths to reach both of these two nodes form a boundary path, such

that any *other* node we visit is *at most one* edge away from the boundary path. If some other node is more than one edge away from the path, then this contradicts the fact that the interval is entire and has no gaps.

3.2 Updating

Updating a single element is simple. We work our way bottom-up, first updating the leaf and then updating all the nodes on the way to the root. For each node that we update, we simply recalculate its *value* from its children. The complexity here is $O(\log N)$ because of the bound on the height of the tree.



3.3 Complexity

In this problem, we have a list of N elements and Q queries, each of which can be either a "query" or an "update."

- Creating the tree: $O(N)$, which is proportional to the number of nodes in a binary tree with N leaves.
- Height of the tree: $O(\log N)$.
- Querying a sum: $O(\log N)$ (see above).
- Updating a single value: $O(\log N)$.
- **Total:** $O(N + Q \log N)$.

This is must faster than the naive method of $O(QN)$.

3.4 Lazy Propagation

Consider the previous problem of finding range sums. Let's add a query type:

A: Find the sum of happiness values within a range $[i, j]$.

B: Change the value of happiness of cow i to some new value.

C: Add k to all the happiness values within a range $[i, j]$.

Query type C is especially tricky, because if we update all the values within the range, it can take up to $O(N \log N)$ instead of the usual $O(\log N)$. We can keep it in $O(\log N)$ per query by using a technique called *lazy propagation*.

We add a new variable to each node that represents a lazy *change* amount. Whenever we want to add k to all the nodes within a range, we find the one or two nodes that represent that range and set its *change* value but we do *not* do anything with the children. The key to lazy propagation is **updating only when we have to**. That means the next time we visit to any node with a lazy value set, we "propagate" its lazy value to its children in $O(1)$, updating its own *value*, adding its current *change* value to the *change* values of its children, and then setting its *change* value back to 0.

4 Problems

1. You are given a list of cows, each with its own happiness value just like above. There are three operations: change the happiness value of a single cow, query the sum of a range of cows, and query the maximum of a range of cows. How would you go about implementing this?

2. We want to perform an insertion sort on a list of unique positive integers, each below $100,000$. Find out the total number of swaps to sort the array. **Challenge:** solve the same problem but without a bound on the largest possible integer in the list.

3. You are given a tree with N ($1 \leq N \leq 100,000$) nodes. The tree is rooted. Given any two nodes, find the *least common ancestor* of the two nodes in $O(\log N)$. How do we do this? Hint: consider a certain traversal of the tree.

4. Consider the problem in the "Lazy Propagation" section. Add a fourth query type, D: set all the happiness values within the range $[i, j]$ to k. How do we solve this problem now?

5. (Long Fan, 2008) There are N ($1 \leq N \leq 100,000$) lights, each either on or off. There are M commands to execute. The first type of commands toggles all lights within some contiguous range. The second type of command asks for the number of lights within some range that are turned on. Help Farmer John answer his queries. When a light is toggled, it switches states from on to off or off to on.

6. (Zhou Yuan, 2007) There is some huge hotel for cows, containing N ($1 \leq N \leq 50,000$) rooms in a linear hallway. There are M ($1 \leq M \leq 50,000$) queries. For some queries, a group of cows checks into the hotel, and each group demands a consecutive group of D_i rooms. Canmuu must assign each group of cows the first set of D_i consecutive available rooms. Cows may also leave in groups, but not necessarily in the same groups they arrived. Only contiguous ranges of cows leave together. Help Canmuu decide which check-in requests can be granted and which rooms the entering groups of cows get.