Segment Trees

Charles Zhao

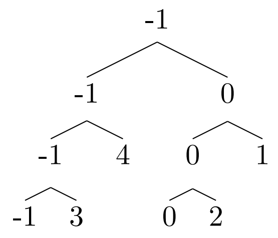
October 28, 2016

1 Introduction

A *segment tree* is a data structure for storing intervals, or *segments*. Segment trees can efficiently answer dynamic range queries. We will use a segment tree to solve the Range Minimum Query (RMQ) problem, which is the problem of finding the minimum element in an array within a given range i to j. Other range queries include the maximum within a range or the sum of a range. A naive solution to RMQ is to iterate from index i to j, which takes $O(n)$ per query. This is too slow if n is large or if there are many queries. Another solution is to build a 2D matrix containing every single RMQ, which would be able to answer queries in $O(1)$ time. However, it would take $O(n^2)$ time to build this matrix and $O(n^2)$ space to store the matrix. Therefore, neither of these solutions scales well. Segment trees solve the problems of both time and space.

2 Constructing the Tree

A segment tree is a balanced binary tree in which each leaf represents an element in the array. The root of the tree represents segment $[0, n-1]$, and for each segment$[l, r]$ represented by the node at index p, the left child represents the segment $[l, (l+r)/2]$ and the right child represents the segment $[(l+r)/2+1, r]$. In the case of RMQ, "represents" means the value of the node is the minimum of the segment it represents. For example, for the array $[-1, 3, 4, 0, 2, 1]$, the tree would look as follows:



Constructing this tree takes $O(n)$ time and $O(n)$ space. In the pseudocode below, we build the tree recursively. The tree is represented as an array *st* where index 1 is the root of the tree and the left and right children of index p are indices $2 \times p$ and $(2 \times p) + 1$, respectively. l and r are the left and right bounds of the current segment, respectively.

Algorithm 1 Segment Tree Construction

```
function BUILD($p$, $l$, $r$)
    if $l = r$ then
        $st[p] \leftarrow A[l]$
    else
        $pl \leftarrow 2 \times p$
        $pr \leftarrow 2 \times p + 1$
        BUILD($pl, l, (l + r)/2$)
        BUILD($pr, (l + r)/2, r$)
        return MIN($st[pl], st[pr]$)
```

3 Solving Queries

There are three cases that we must consider when traversing a segment tree: when part of the segment represented by the node is within the query; when the segment is completely within the query; and when the segment is completely outside of the query. If part of the segment is within the query, we must check both of the node's children. If the segment is completely within the query, we return the node's value, which is the minimum of the segment it represents. If the segment is completely outside of the query, we return some very large number. In the pseudocode below, we traverse the tree recursively. With the segment tree built, solving an RMQ takes $O(\log n)$ time. This is because segment trees allow us to avoid traversing unrelated parts of the tree. In the worst case, in which only part of every segment we reach is within the query, we traverse two root-to-leaf paths, taking $O(2 \times \log n) = O(\log n)$ time.

Algorithm 2 Range Minimum Query Using a Segment Tree

```
function RMQ($p$, $l$, $r$, $i$, $j$)
    if $i > r$ or $j < l$ then
        return $\infty$
    if $l >= i$ and $r <= j$ then
        return $st[p]$
    $pl \leftarrow 2 \times p$
    $pr \leftarrow 2 \times p + 1$
    $minl \leftarrow$ RMQ($pl, l, (l + r)/2, i, j$)
    $minr \leftarrow$ RMQ($pr, (l + r)/2 + 1, r, i, j$)
    return MIN($minl, minr$)
```

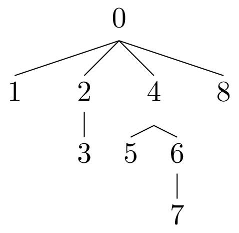
4 Modifying the Tree

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.

5 Fenwick Trees

A *Fenwick Tree*, also known as a *Binary Indexed Tree (BIT)*, is essentially a faster and easier to code (although a bit more complicated to understand) segment tree that takes advantage of extremely efficient bit manipulation. A Fenwick tree can be used when the operation is both associative and has an inverse, e.g. addition[1]. We will use a Fenwick tree to solve the Range Sum Query (RSQ) problem, which is the problem of finding the sum of a segment within an array from index i to j. A naive solution would be to generate a cumulative frequency table c such that $c[n] = c[0] + c[1] + ... + c[n]$. Then, $\text{RSQ}(i, j) = c[j] - c[i-1]$. If the frequencies are *static*, then this solution is fast because the cumulative frequency table can be generated in $O(n)$. However, updating the table also takes $O(n)$. We can use a Fenwick tree to implement an efficient *dynamic* cumulative frequency table.

A Fenwick tree is typically implemented as an array. As implied by its alternative name—a binary indexed tree—a Fenwick tree is indexed by the bits of its integer keys. Let us say the $\text{LSO}(n)$ returns the least significant one-bit in n. For example, if $n = 6 = 110_2$, then $\text{LSO}(1\underline{1}0_2) = 10_2 = 2$. This can be computed easily (and very quickly) with $\text{LSO}(n) = n \ \& \ (-n)$. The element at index i represents the elements in the range $[i - \text{LSO}(i) + 1, i]$. Thus, in the array representation of the tree, the parent of the node at index i is the node at index $i - \text{LSO}(i)$. In other words, every time you move up the tree, you strip off the least significant one-bit. The tree below shows how an array of size 8 would be indexed in a Fenwick tree. Since an integer contains $O(\log n)$ bits, we can answer RSQs in $O(\log n)$ time with a Fenwick tree.



Now, we also need an efficient way to build the tree. We initialize the tree with each node having a value of 0. To obtain the sequence of nodes whose values include the element at index i, we use the iterative bit manipulation expression $i' = i + \text{LSO}(i)$, updating the Fenwick tree array ft at $ft[i], ft[i'], ft[i''], ...$ until i^k exceeds the length of ft. Since adding each element to the tree takes $O(\log n)$ time, building the tree requires $O(n \log n)$ time. If one of the elements changes, we can efficiently update the tree in the same way we added each element to the tree, thus taking $O(\log n)$ time.

Algorithm 3 Range Sum Query Using a Fenwick Tree

```
function RSQ($i$)
    $sum \leftarrow 0$
    while $i > 0$ do
        $sum \leftarrow sum + ft[i]$
        $i \leftarrow i - \text{LSO}(i)$
    return $sum$
function RSQ($a$, $b$)
    return $\text{RSQ}(b) - \text{RSQ}(a-1)$
function MODIFY($i$, $v$)
    while $i < \text{SIZE}(ft)$ do
        $i \leftarrow i + \text{LSO}(i)$
        $ft[i] \leftarrow ft[i] + v$
```

[1]Sam Hsiang's *Crash Course Coding Companion*