Data Structure III

Segment Trees

Section 1

Segment Trees

Introduction

We introduce another data structural, segment trees, to support the interval operation. Compared with the block list, it is usually more efficient.

We will illustrate the idea by showing how to support the following operations on an integer array $A[1, 2, \ldots, n]$.

- Update (i, j, k). Setting $A_i, A_{i+1}, \ldots, A_j$ to k.
- Sum (i, j). Return the value of $\sum_{l=i}^{j} A_l$.

By using a segment tree, we can perform each operation in time $O(\log n)$, with an $O(n)$ pre-processing (construction) time.

Definition

The segment tree is a binary tree. Each node in the tree represents an interval $[i, j]$ where $1 \leq i \leq j \leq n$.

If the interval $[i, j]$ represented by a node satisfies $i \neq j$, then it has exactly two child nodes. The left node represents interval $[i, \lfloor \frac{i+j}{2} \rfloor]$, and the right node represents interval $[\lfloor \frac{i+j}{2} \rfloor + 1, j]$.

Otherwise, if some node represents an interval $[i, i]$, then it is a leaf and has no child nodes.

Construction

We let the root represent interval $[1, n]$. Then all other nodes can be constructed recursively.

It can be shown that the height of the segment tree is $O(\log n)$ Moreover, there are at most $4n$ nodes in the segment tree, and hence the construction takes linear time, and linear space.

Covering An Interval

Given any interval $[i, j]$ for $1 \leq i \leq j \leq n$, there exists an algorithm that finds $O(\log n)$ nodes that partitions the interval $[i, j]$, running in time $O(\log n)$.

We describe the procedure COVER(r, i, j), that takes i, j as the input interval, and r as the root of the segment tree. It returns a set S of nodes that represent the partition of $[i, j]$.

Covering An Interval: Algorithm

Step 1. If r represents $[i, j]$, then return $\{r\}$. Otherwise, go to Step 2.
Step 2. Suppose r represents $[L, R]$, and define $M := \lfloor \frac{L+R}{2} \rfloor$.

- If $j \leq M$, then it means that $[i, j]$ lies in the left child of r entirely, and we return $\mathrm{COVER}(r.\mathrm{leftChild}, i, j)$.
- If $M + 1 \leq i$, then it means that $[i, j]$ lies in the right child of r entirely, and we return $\mathrm{COVER}(r.\mathrm{rightChild}, i, j)$.
- Otherwise, return $\mathrm{COVER}(r.\mathrm{leftChild}, i, M) \cup \mathrm{COVER}(r.\mathrm{rightChild}, M + 1, j)$.

Covering An Interval: Analysis

Obviously, the algorithm returns a set of nodes representing intervals that partition $[i, j]$.

We shall show that the algorithm runs in $O(\log n)$ time, and hence the size of the output is $O(\log n)$.

If the algorithm always goes into the first two "if" branches of Step 2, then it is immediate that the algorithm runs in $O(\log n)$ time.

Then we look at the case when the algorithm goes into the "otherwise" branch of the Step 2, in which case COVER(r.leftChild$, i, M$) and Cover(r.rightChild$, M + 1, j$) are invoked. We observe that if for all such cases, either r.leftChild represents $[i, M]$, or r.rightChild represents $[M + 1, j]$, then the algorithm still runs in $O(\log n)$.

It remains to consider the case that in the “otherwise” branch, neither r.leftChild represents $[i, M]$, nor r.rightChild represents $[M + 1, j]$.

We claim that this can only happen once among all the invocation of the algorithm. To see it, we consider the first time this case happends. Then r.leftChild represents an interval that ends at M, and r.rightChild represents an interval that starts at $M + 1$. One can verify that the subsequent invocations cannot fall in this case again.

Maintaining Sum

Now we are to define some variable for each node in order to support the desired operations.
We maintain a variable, sum, in each node. For a node u, u.sum denotes the sum of the elements in the interval that u represents.

Query

Suppose we have maintained the sum variables for all nodes. Then for a query (i, j), we can invoke COVER(root, i, j) to find $O(\log n)$ nodes that represent a partition of $[i, j]$, and then we sum up the "sum" variable in the nodes to get the answer.

Issues in Update

However, when it comes to the update, it seems updating the $O(\log n)$ intervals returned by the COVER procedure is not sufficient.

As an example, suppose we are going to update $(1, n, x)$, which asks us to set every element in the segment tree to be x. To make every "sum" variable correct, we have to go through the tree, and it takes $O(n)$.

Lazy Variables

Can we just make *some* of the “sum” variables correct? Yes, we can. Actually, we can just update $O(\log n)$ nodes in the update proceudre.

For each node, we give it an additional boolean variable “lazy”, and an additional integer variable “lvalue” which intends to mean “lazy value”.

Whenever the “lazy” variable indicates “true”, then the sum value of the current node, as well as its child nodes’, may not yet be correctly set, but they are supposed to be set to “lvalue”.

Push Down The Lazy Value

For some node r, the procedure PUSHDOWN(r) is used to update the sum field of r as well as the other information for child nodes of r, using the lazy value of it.

PUSHDOWN(r) If r.lazy = true, then

- set r.sum to be r.lvalue times the length of the interval that r represents;
- if r is not a leaf, then set the both the “lazy” variable of the two child nodes to be true, set both the “lvalue” variable of the two child nodes to be r.lvalue;
- set r.lazy = false.

Lazy Update

Combining with the idea introduced in COVER, we describe the update procedure UPDATE(r, i, j, k) that runs in time $O(\log n)$, where r is the root node. The procedure will set $A_i, A_{i+1}, \ldots, A_j$ to k.

Step 1. If r represents the interval $[i, j]$ exactly, then set r.lazy := *true* and r.lvalue := k. Return.
Step 2. Otherwise, PUSHDOWN(r).
Step 3. Suppose r represents interval $[L, R]$, and define $M := \lfloor \frac{L+R}{2} \rfloor$. If $j \leq M$ then UPDATE$(r.\text{leftChild}, i, j, k)$. If $M + 1 \leq i$ then UPDATE$(r.\text{rightChild}, i, j, k)$. Otherwise, UPDATE$(r.\text{leftChild}, i, M, k)$ and UPDATE$(r.\text{rightChild}, M + 1, j, k)$.
Step 4. PUSHUP(r).

Push Up The Update

We elaborate the PUSHUP procedure used in Step 4. Since we update the sum of some descendants of r, we need to update the sum variable of r after updating the descendants. We use the PUSHUP procedure to do so, which is used in Step 5 of the UPDATE procedure.

PUSHUP(r) If r is a leaf, then return. Otherwise, PUSHDOWN(r.leftChild), PUSHDOWN(r.rightChild), and set $r.\text{sum} := r.\text{leftChild.sum} + r.\text{rightChild.sum}$.

We note that the PUSHDOWN in the procedure is necessary to get the correct "sum" value of the child nodes.

Query Procedure

Since we use lazy update, we need to change the query procedure QUERY(r, i, j) accordingly, where r is the root, and the procedure returns the sum of elements of indices from i to j.

Step 1. PUSHDOWN(r).
Step 2. If r represents the interval $[i, j]$ exactly, then return r.sum. Return.
Step 3. Otherwise, suppose r represents interval $[L, R]$, and define $M := \lfloor \frac{L+R}{2} \rfloor$.
Step 4. If $j \leq M$ then return QUERY$(r.\text{leftChild}, i, j)$. If $M + 1 \leq i$ then return QUERY$(r.\text{rightChild}, i, j)$. Otherwise, return QUERY$(r.\text{leftChild}, i, M) +$ QUERY$(r.\text{rightChild}, M + 1, j)$.

Conclusion

We have described the update and query procedure with lazy update technique. We note that the lazy update technique is necessary in order to achieve $O(\log n)$ time complexity.

The segment tree is not limited to support the example operations discussed so far. To support other kinds of operations, we note that it is usually only necessary to change the PUSHUP and PUSHDOWN procedure as well as the variables defined in the node.

You can explore how the segment trees support other kinds of operations in the exercises.