Range queries and segment trees

Mihir Raj 20ME10115
Shivam Dwivedi 20CY20029
Aditya Chari 20MF10002

Motivation

Many situations necessitate providing responses based on a query across a certain range or section of data. This can be a time-consuming and slow operation, especially if you have a big number of inquiries to handle. As a result, we want a data structure that allows us to effectively execute such requests.

In the fields of computational geometry and geographic information systems, these issues become critical. We could have a vast number of points in space at different distances from a central reference/origin point, for example:

Motivation

Assume we need to find locations within a specified range of distances from our starting point. A traditional lookup table would need a linear scan of all conceivable locations or distances (think hash-maps). We'll go through a data structure that allows us to do this in logarithmic time and with a lot less space. Planar Range Searching is the name for such a problem. It's vital to solve such issues quickly, especially when working with dynamic data that changes quickly and unexpectedly (for example, a radar system for air traffic control).

Problem statement

With the following easy scenario, we'll try to explain how data structures function and why they're important.

There is an initial sequence of integers A. This is followed by a two-type operation sequence:

Type 1 -> Make a change to one of the sequence's entries.

Type 2 -> in A, report the lowest value in a defined range of indices.

Naive implementation-1

Input the array 'a' with min=inf for all entries “e" in array 'a' and one by one, locate min, and return min

```
Pseudo Code: minimum(){
    Input the array ‘a’
    min=inf
    for all elements e’’ in array ‘a’: find
    min one by one
    return min
}
```

Naive implementation-1

update() takes the 'a' array as an argument.

Pseudo Code:

```
update(){
    Input the array ‘a’
    Input index ‘i’ to update and
    value ‘v’
    a[i]=v;
    return a[i]
}
```

Time complexity -1

update() - O(1)

minimum() - O(n)

As can be seen, the update process is quite quick, but the minimal function will take O(n) time to complete.

Naive implementation-2

If we limit our range queries to [0..n-1] min queries, we may utilise the following.

```
minimum(){
    Input the array 'a' in the form of a
    suffix min, i.e., the element at
    index I will be the minimum among
    all elements from 0 to I and minimum
    at index 'n-1' will be returned.
}
```

Naive implementation-2

```
update(){
    Input the array 'a' in the form of
    a suffix min, where the element at
    index I is the minimum of all
    entries from 0 to i.
    Input index ‘i’ to update and
    value ‘v’
    a[i]=v;
    After index I for each subsequent
    index:
    Return a[i] if the minimum has to
    be adjusted.
}
```

Time complexity -2

update() - O(1)

minimum() - O(n)

The complications are traded here.

We have assumed, however, that the inputs are taken in the way described in the pseudocode, which may take longer but is unrelated to the DS we want to create.

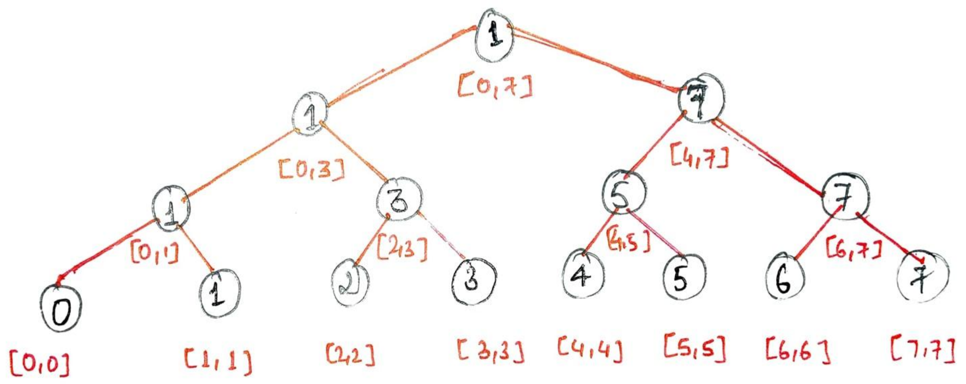
The functions mentioned may easily be changed to work with a range rather than the entire array.

Segment Tree

A segment tree is a data structure that does both of the aforementioned functions in O(log n) time.

Range(i, j) and update(i, val) both require log (n) time to complete, where n is the number of input items.

Segment Tree



A diagram depicting how the segment tree divides into two halves every time

Explanation & Observation

We begin with the arr[0...n-1] section. Every time we divide the current segment into two halves (assuming it hasn't already become a segment of length 1), we call the same method on both half, and we store the minimum in the associated node, which is determined as the minimum of both the child's values.

Except for the last level, the created segment tree will be totally filled. Also, because we consistently divide segments in two halves at every level, the tree will be a Full Binary Tree. There will be n-1 internal nodes since the created tree is always a complete binary tree with n leaves. As a result, the total number of nodes is 2*n – 1.It's worth noting that dummy nodes aren't included.

In this fashion, the tree's leaves represent the components, and each higher level is formed by combining two lower levels.

Query Explanation

We are given two indices l,r as input, and we must find the minimum of the segment a[l...r]. We'll achieve this by traversing the Segment Tree and using the segments' precomputed minima. Assume we're now at the vertex that spans the a[first...last] segment. There are three scenarios that might occur. When the segment a[first...last] is contained inside a[l...r] and corresponds to the appropriate segment of the current vertex (i.e. first>=l and last>=r), we are done and may return the precomputed minimum that is stored in the vertex.

Query Explanation

Alternatively, the query segment might lie entirely inside the domain of either the left or right child. Remember that the left child covers the a[first...mid] segment and the right vertex covers the a[mid+1...last] segment, with mid = (first + last)/2.

In this situation, we can simply go to the child vertex whose corresponding segment covers the query segment and use that vertex to run the technique given below.

Then there's the last instance, in which the query section crosses both children. We have no choice but to perform two recursive calls, one for each kid, in this scenario.

Query Explanation

We start with the left child and compute a partial answer for this vertex (i.e. the minimum of values of the intersection between the query segment and the segment of the left child), then move on to the right child and compute the partial answer using that vertex, and finally add the answers together.

In other words, we calculate the minimal query a[l...mid] using the left child, and the minimum query a[mid+1...r] using the right child, since the left child represents the segment a[first...mid] and the right child represents the segment a[mid+1...last].

Query Explanation

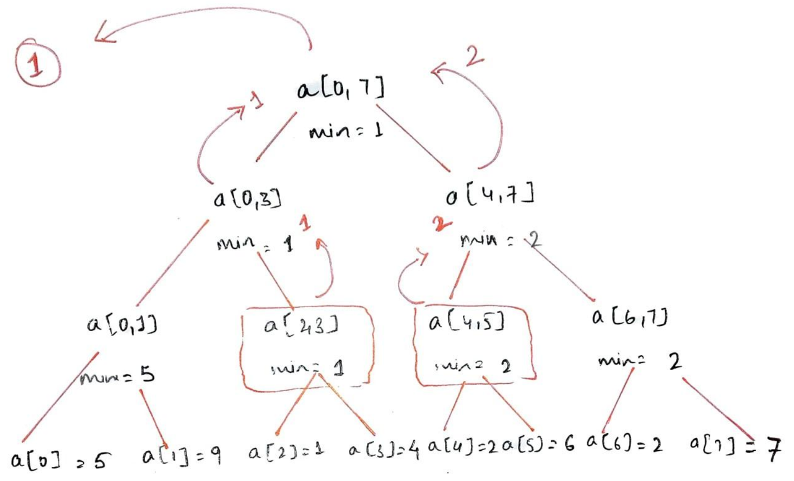
So, processing a minimal query is a function that recursively executes itself once (without modifying the query limits) with either the left or right child, or twice (once for the left and once for the right child) (by splitting the query into two subqueries). When the borders of the current query segment coincide with the bounds of the current vertex's segment, the recursion stops. The precomputed value of the minimum of this segment, which is kept in the tree, will be the answer in such scenario.

Query Explanation

In other words, the query's calculation is a tree traversal that goes through all of the tree's essential branches and uses the precomputed sum values of the tree's segments.

Obviously, we'll begin our traversal at the Segment Tree's root vertex.

Query Explanation



In the segment tree, the query for the range (2, 5) is shown in the diagram above.

Update Explanation

Let's imagine we wish to change a specific element in the array, thus we'll use the assignment a[i]=x. We also need to rebuild the Segment Tree so that it matches the new, changed array.

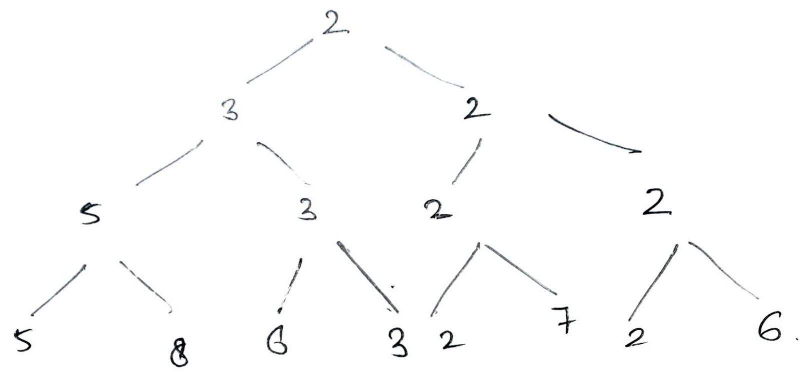
A partition of the array is formed by each level of a Segment Tree. As a result, each level's constituent a[i] only contributes to one segment. As a result, just the vertices must be modified.

Update Explanation

It's clear that a recursive function may be used to implement the update request. The function is supplied the current tree vertex, then it recursively runs itself with one of the two child vertices (the one containing a[i] in its segment), then recomputes its minimal value, much like the build method does (that is as the minimum of its two children).

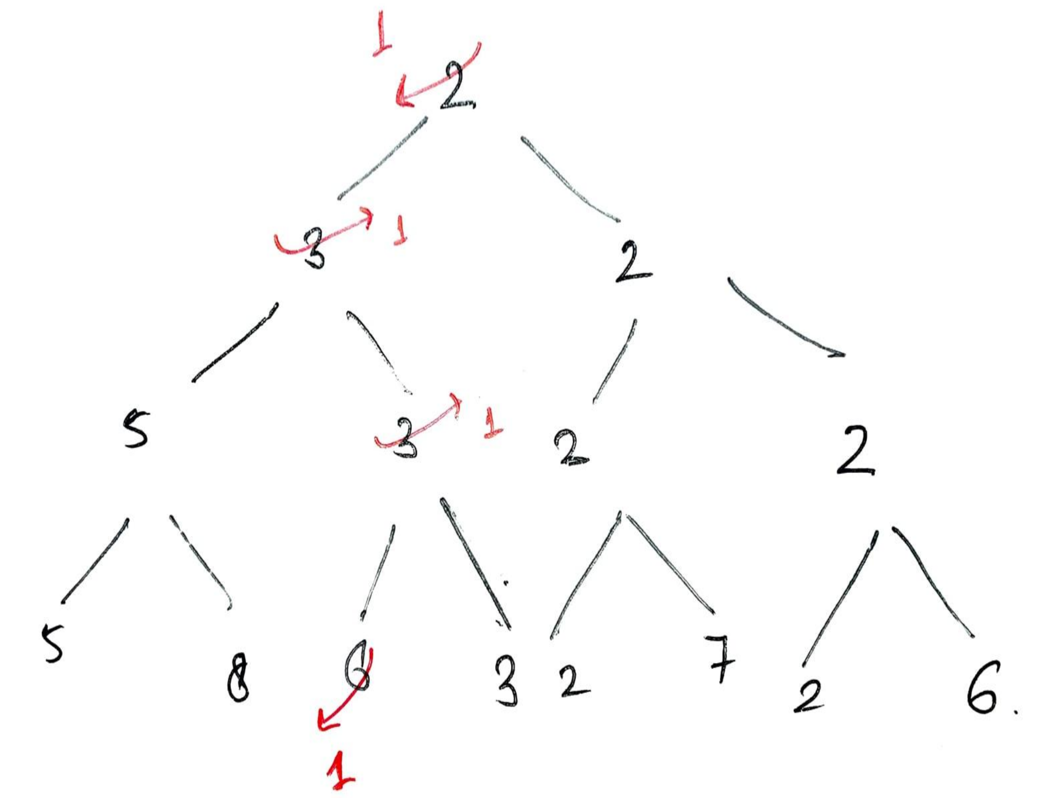
Here's another example utilising the same array. We'll alter a[2] = 6 to a[2] = 1 here, assuming this is the segment tree we built previously.

Update Explanation



We can see that if we change 6 to 1 in the second place, we must also change the parent from 3 to 1, because the parent includes the children's minimal value. As shown in the diagram below, the recursive method first discovers the particular location where we need to make changes in depth first and then updates the segment while backtracking.

Update Explanation



Pseudo Code

1. preProcessing fxn

```
preProcessing(node, first,last array,
SegmentTree){

    array= {$a_{first}$,$a_{first+1}$.$a_{first+2}$…………$a_{first+3}$}

    //base condition

    if(first=last){

        SegmentTree[node]=arr[first]

        return

    }
```

Pseudo Code

1. preProcessing fxn

```
    mid = (first+last) / 2 //mid of the array

    //recursive calls

    preProcessing(2*node, first, mid, array,
    SegmentTree)

    preProcessing(2*node + 1, mid+1, last,
    array, SegmentTree)

    //building the segment tree

    SegmentTree[node] =
    minimum(SegmentTree[2*node] ,
    SegmentTree[2*node + 1])

}
```

Pseudo Code
2. Update Fxn

```
update(node, first, last, index, newkey,
SegmentTree){

        //base conditions

        if(first = last){

        SegmentTree[node] = newkey

        return

    }
```

Pseudo Code
2. Update Fxn

```
    mid = (first + last) / 2 //

    recursive calls for updation

    if(index <= mid)

        update(2*node, first, mid, index,
        newkey, array, SegmentTree)

    else

        update(2*node+1, mid+1, last, index,
        newkey, array, SegmentTree)

    //updation while backtracking
    SegmentTree[node]=minimum(SegmentTree[2*node]
    , SegmentTree[2*node + 1])

}
```

Pseudo Code

3. Query Fxn

```
query(node, first, last, l, r,
SegmentTree){

     //base conditions

    if( first > r or last < l){

        return INFINITY

    }

    if(l = first and r = last){

        return SegmentTree[node]

    }
```

Pseudo Code
3. Query Fxn

```
        mid = (first + last) / 2

        //recursive calls for returning the
        minimum value

        return minimum (query(2*node, first,
        mid, l, r, SegmentTree),
        query(2*node+1, mid+1, last, l, r,
        SegmentTree))

    }
```

Time Complexity

Now let's look at the Segment Tree time and space complexity -

Because our data structure performs three functions: **preprocessing, query, and update,** we must determine the time and space difficulties for each of them before determining the ultimate complications when the data structure is employed.

Space required by Segment Tree as an array Let S(n) be the space required .

Then, $S(n) = S(\lfloor n / 2 \rfloor) + S(\lceil n / 2 \rceil) + O(1)$ $n > 1$

$= 2 * S(n / 2) + O(1)$

$S(n) = O(1)$ $n = 1$

By, master theorem then ,$S(n) = O(n\log_2 2) = O(n)$

Time Complexity

1. preProcessing fxn

Let's say the temporal complexity is TP (n).

Because the Pre-processing phase is independent of the values contained in a range of the input array, the time is solely affected by its size.

The extent of the range is the only factor that matters.

Then, Tp (n) = Tp (⌊n / 2⌋) + Tp (⌈n / 2⌉) + O(1) n > 1 = 2 * TP (n / 2) + O(1) T p (n) = O(1) n = 1

So, Tp (n) = O(n) by master theorem

Time Complexity

1. preProcessing fxn

This may be seen as if we are only accessing each node once, and therefore time complexity is proportional to the number of nodes, which is O(n), as we demonstrated in the space complexity of segment trees.

Time Complexity
2. Update Fxn

Let its time complexity be T_u([st, end], [indx]).

There is no need to update this range and its subranges if the revised index does not fall within the current update range.

Alternatively, if the current range just contains our modified index, we simply change its value

$$T_u\ ([st,\ end],\ [indx]) = O(1)$$

otherwise

$$T_u\ ([st,\ (st + end) / 2],\ [indx]) + T_u\ ([(st + end) / 2 + 1,\ end],\ [indx]) + O(1)$$

Time Complexity
2. Update Fxn

We only go through one kid at a time for each Tree node, just like in the query function. Because both the child and the parent hold distinct subranges of the range at node, they cannot share an index.

Because we traverse the height of the segment tree and spend a consistent number of operations at each height, the time complexity is O(height), which is O(log n).

Pseudo Code
3. Query Fxn

Let its time complexity be Tq([st, end], [l, r]).

if the query range [l, r] and the search range[st, end] are disjoint, or if the search range is entirely inside the query range -

Tq (n) = O(1)

otherwise

Tq ([st,end], [l, r]) =

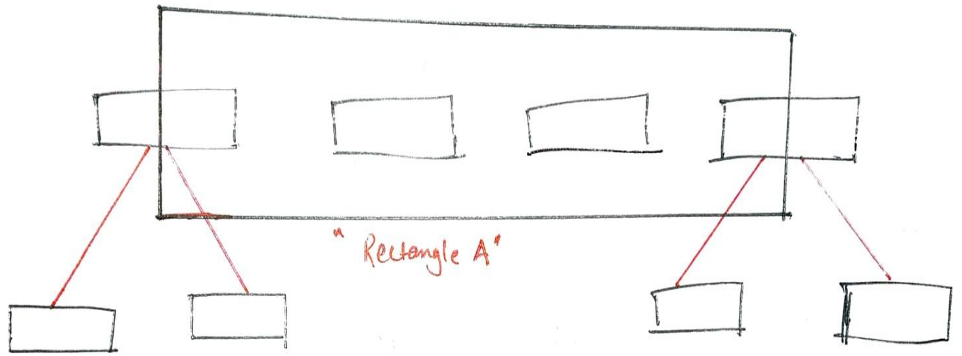
    T q ([st, (st + end) / 2], [l, r]) + Tq ([(st + end) / 2 + 1, end], [l, r]) + O(1)

Pseudo Code
3. Query Fxn

The query operation method in a segment tree never accesses more than four vertices at each level of the tree. As a result, the query operation's time complexity is O(height), which is O(log n), where n is the number of entries. Induction may be used to establish this.

Because we only visit one node at level one, the assertion is valid.

Pseudo Code
3. Query Fxn



→ The rrectangle A is query range and all other rectangle are presently called nodes. The rrect angles in same line are nodes at same level.

Pseudo Code
3. Query Fxn

As seen in the picture above, if we only visit four vertices at the current level, we will only visit four vertices in the next level. The nodes in the centre will be totally within the query range, therefore no more calls will be made. The rightmost and leftmost nodes can each make a maximum of two calls. As a result, we may infer that at each level, we can only visit four nodes.

As a result, O(height) = O is the time complexity (log n)

Extensions

1. Any associative function can be applied with these segment trees. Everything remains the same; the only difference is how we obtain the ultimate value of a node from the values of its children.

We could, for example, use segment trees to calculate total.

The total of the values present at both of a node's children will equal its value. The rest of the situation stays unchanged.

2. Range updates may be readily added to Segment Trees, allowing us to update a range of indices rather than simply one.

3. Segment Trees may be used to answer questions and provide updates in greater dimensions.