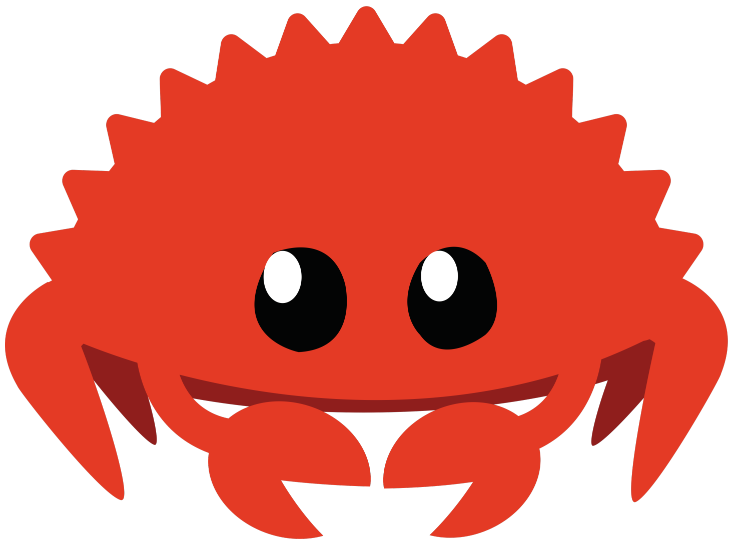
Hands-On 2 - Report

Giuseppe Di Palma - 635525



November 16, 2024

1 Introduction

This report evaluates the two different Segment Tree implementations used to solve the **Min and Max** and **Is There** problems proposed for this practical. The aim is to verify that both implementations satisfy the Segment Tree **Time Complexity** constraints and how effectively they can solve the associated problem.

1.1 Segment Tree Structure

Both implementations use the implicit representation of a segment tree without effectively implementing `nodes` and `edges`. It is possible to represent a segment tree using a vector. In particular, using the Euler tour traversal indexing, it is possible to have exactly $2*n-1$ elements, like the tree with node representation. In Euler indexing the left and right children of a node are respectively at positions $idx+1$ and $idx+2*(mid-left\_range+1)$ in the vector, where `left_range` is the minimum index of the range of the node and mid is $(right\_range - left\_range)/2$.

2 Min and Max

The problem statement is the following. Given an array $A[1,n]$ of n positive integers, we want to answer two different types of queries:

- **Update**$(\mathbf{i}, \mathbf{j}, \mathbf{T})$ that replace every value $A[k]$ with $min(A[k], T)$, where $i \leq k \leq j$
- **Max**$(\mathbf{i}, \mathbf{j})$ that return the largest value in $A[i...j]$

2.1 Tree Building

```rust
pub struct MinMaxSegmentTree {
    tree: Vec<u32>,
    lazy: Vec<Option<u32>>,
    range_len: usize,
}

impl MinMaxSegmentTree {
    // Initialize the segment tree and build it from the input vector
    pub fn new(vec: &Vec<u32>) -> Self {
        let tree_len = vec.len() * 2 - 1;
        let lazy = vec![None; tree_len];
        let mut tree = vec![0; tree_len];

        Self::build_segment_tree(&mut tree, vec, 0, 0, vec.len() - 1);

        Self {
            tree,
            lazy,
            range_len: vec.len(),
        }
    }

    // Build the segment tree recursively to store the maximum value
   in each range
    fn build_segment_tree(
        tree: &mut Vec<u32>,
        vec: &Vec<u32>,
        index: usize,
        left_range: usize,
        right_range: usize,
    ) {
        if left_range == right_range {
            tree[index] = vec[left_range];
        } else {
            let mid = left_range + (right_range - left_range) / 2;
            let right_index = index + 2 * (mid - left_range + 1);
            let left_index = index + 1;

            Self::build_segment_tree(
            tree, vec, left_index, left_range, mid);
            Self::build_segment_tree(
            tree, vec, right_index, mid + 1, right_range);
            tree[index] = cmp::max(
            tree[left_index],
            tree[right_index]);
        }
    }
    ...
```

Listing 1: `Min and Max Building`

The construction of the MinMaxSegmentTree is quite simple.
Starting from the root node, we recursively evaluate its value as the maximum between the values of its left and right children. We can also see that the data structure has a second vector called lazy, initialised with None, later used in both Max and Update. The resulting data structure is built in $O(n)$ time complexity because the `build_segment_tree` is called exactly once for each position in the tree vector, and the merge operation (in this case, the `cmp::min`) is performed in constant time. It also requires $O(n)$ memory usage because it only stores two different vectors of size $2 * n - 1$: `tree` and `lazy`.

2.2 Update

```
    ...
    // Public method to update range [i, j] with a minimum value t
    pub fn update(&mut self, i: usize, j: usize, t: u32) {
        self.rec_update(0, 0, self.range_len - 1, i, j, t)
    }

    // Recursive function to update a range with a minimum value,
   using lazy propagation
    fn rec_update(
        &mut self,
        index: usize,
        left_range: usize,
        right_range: usize,
        i: usize,
        j: usize,
        t: u32,
    ) {
        if i <= j {
            if i <= left_range && j >= right_range {
                self.tree[index] = self.tree[index].min(t);
                self.lazy[index] = self.lazy[index].map_or(
                Some(t), |x| Some(x.min(t)));
            } else {
                let mid = left_range + (right_range - left_range) / 2;
                let right_index = index + 2 * (mid - left_range + 1);
                let left_index = index + 1;

                self.push(index, mid, left_range);

                self.rec_update(
                left_index, left_range, mid, i, j.min(mid), t);
                self.rec_update(
                right_index, mid + 1, right_range,
                i.max(mid + 1), j, t);
                self.tree[index] = cmp::max(
                self.tree[left_index], self.tree[right_index]);
            }
        }
    }

    // Push lazy updates to children and reset the lazy value at the
   current node
    fn push(&mut self, index: usize, mid: usize, left_range: usize) {
        if let Some(lazy_value) = self.lazy[index] {
            let right_index = index + 2 * (mid - left_range + 1);
            let left_index = index + 1;
```

```

                self.tree[left_index] =
                self.tree[left_index].min(lazy_value);
                self.lazy[left_index] =
                self.lazy[left_index].map_or(
                Some(lazy_value), |x| Some(x.min(lazy_value)));

                self.tree[right_index] =
                self.tree[right_index].min(lazy_value);
                self.lazy[right_index] =
                self.lazy[right_index].map_or(
                Some(lazy_value), |x| Some(x.min(lazy_value)));

                self.lazy[index] = None;
            }
        }
        ...
```
Listing 2: `Min and Max, Update Query`

The `update` function works as follows.
When we want to update a given range of values, we do not perform the update all over the values of the requested range, but only on those nodes that overlap with the recursively partitioned requested range. This satisfies the update $O(logn)$ time complexity constraint, but we need a method to push the new information stored in the lazy vector to the children of the node when necessary. So, as we can see from the code above if the node overlaps the query, we store the new value in both `tree` vector and `lazy` vector (taking the minimum between them). Otherwise, there is no overlap, so before we recursively update the node's children, we push any previously pending updates to its children. The push function simply pushes the pending update, if any, to the node's children taking the minimum between their value and it, for both the main tree vector and the lazy support vector. It then restores the lazy default value.

2.3 Max

```rust
    ...
    // Public method to get the maximum value in a given range [i, j]
    pub fn max(&mut self, i: usize, j: usize) -> u32 {
        self.rec_max(0, 0, self.range_len - 1, i, j)
    }

    // Recursive function to retrieve the maximum value within range
   [i, j]
    fn rec_max(
        &mut self,
        index: usize,
        left_range: usize,
        right_range: usize,
        i: usize,
        j: usize,
    ) -> u32 {
        if i > j {
            return u32::MIN;
        }

        if i <= left_range && j >= right_range {
            return self.tree[index];
        }

        let mid = left_range + (right_range - left_range) / 2;
        let right_index = index + 2 * (mid - left_range + 1);
        let left_index = index + 1;

        self.push(index, mid, left_range);

        cmp::max(
        self.rec_max(left_index, left_range, mid, i, j.min(mid)),
        self.rec_max(
        right_index, mid + 1, right_range, i.max(mid + 1), j),
        )
    }
}
```

Listing 3: `Min and Max, Max Query`

The `max` function works as follows.
If the requested query range overlaps with the node range the function returns the value contained in that node. Otherwise, there is no overlap, the function, after calling the `push` utilities over the current node, returns the maximum between the recursive call to the node children. This operation requires $O(logn)$ time complexity.

3 Is There

The problem statement is the following. Given n segments $< l, r >$ such that $0 \leq l \leq r \leq n-1$, we want to answer:

- **IsThere**$(\mathbf{i}, \mathbf{j}, \mathbf{k})$ that return `true` if there exists a position p, with $0 \leq i \leq p \leq j \leq n-1$, such that exactly k segments contain position p, `false` otherwise.

3.1 Tree Building

```rust
pub enum Point {
    Start,
    End,
}

pub struct IsThereSegmentTree {
    tree: Vec<HashSet<u32>>,
    range_len: usize,
}

impl IsThereSegmentTree {
    // Initialize the segment tree and build it from the input vector
    pub fn new(
        segments: &Vec<(u32, u32)>,
        num_segments: usize) -> Self {

        // create a vector of points with their value and their type
        let mut points: Vec<(u32, Point)> = Vec::new();
        for (start, end) in segments {
            points.push((*start, Point::Start));
            points.push((*end + 1, Point::End));
        }

        // sort the vector with counting_sort, exploiting the fact
        // the maximum value of the endpoint is known
        points = counting_sort(points, num_segments);

        let mut values = vec![0; num_segments];
        let mut active_segments: u32 = 0;
        let mut current_position: u32 = 0;

        // apply the sweep line method to count the number of active
    segments at each position
        for (position, event) in points {
            while current_position < position {
                values[current_position as usize] = active_segments;
                current_position += 1;
            }

            match event {
                Point::Start => active_segments += 1,
                Point::End => active_segments -= 1,
            }
        }

        let tree_len = values.len() * 2 - 1;
        let mut tree = vec![HashSet::new(); tree_len];

        Self::build_segment_tree(&mut tree, &values, 0, 0, values.len
() - 1);

        Self {
            tree,
            range_len: values.len(),
        }
    }
    ...
```

Listing 4: `Is There Building`

As before, the construction of the segment tree itself is quite simple, but in this case, we need to perform some preliminary operations on our input. As mentioned in the problem statement, we are given n segments with a fixed maximum size of $n-1$. In this case, each node of the segment tree has to store not a single value but the set of unique values present in its subtree. To do this, we need to manipulate the segments to achieve this result. We start by creating a new vector called `points`, in which we insert all the points of all the segments, classified according to their type (start or end point). Then we can sort this vector, taking advantage of the fact that the maximum value in `points` is known, in $O(n)$ time with counting sort. Once we have the sorted vector we can use the sweep line method to create the final vector `values` which stores the number of active segments at each position $[0, n-1]$. The `build_segment_tree` function works exactly like the previous one, but in this case, each node contains a HashSet and the merge function performs the union between the node children HashSets.

The resulting data structure is built in $O(n * logn)$ time complexity. In this case, it is better to think in terms of levels rather than individual nodes. By construction, the data structure has a maximum of $O(logn)$ levels, and for each of them, there can be a maximum of n different values. `Union` operation on `HashSet` requires $O(t+s)$ time complexity, where t and s are the sizes of the HashSets to be unified respectively, so for what we said before this operation costs a maximum of $O(n)$ for each level. It also requires $O(n * logn)$ memory usage for the same reason as above, in fact, each of the $logn$ levels contains at most n different elements.

3.2 Is There

```rust
    ...
    // Check if a value exists in the given range [i, j]
    pub fn is_there(&self, i: usize, j: usize, k: u32) -> bool {
        self.rec_is_there(0, 0, self.range_len - 1, i, j, k)
    }

    // Recursive function to check if a value is in the set of a given
    range
    fn rec_is_there(
        &self,
        index: usize,
        left_range: usize,
        right_range: usize,
        i: usize,
        j: usize,
        k: u32,
        ) -> bool {
        if i > j {
            return false;
        }

        if i <= left_range && j >= right_range {
            return self.tree[index].contains(&k);
        }

        let mid = left_range + (right_range - left_range) / 2;
        let right_index = index + 2 * (mid - left_range + 1);
        let left_index = index + 1;

        self.rec_is_there(
            left_index, left_range, mid, i, j.min(mid), k)
            || self.rec_is_there(
                right_index, mid + 1, right_range,
                i.max(mid + 1), j, k)
    }
}
```

Listing 5: `Is There Query`

The `is_there` works as follows. If the requested range query overlaps with the node range the function returns if the value k is contained in the node. Otherwise, there is no overlap, the function returns if one of the recursive calls to the node children contains the value k. This operation requires O(logn) time complexity.

4 General Information

The main.rs file contains the implementation to perform the tests, assuming the test directories are in the same folder of the project.
The main resourse used is cp-algorithms.com