Cairo University
Faculty of Computers and Artificial Intelligence
Computer Science Department

Advanced Data Structures

Interval and Segment Trees

Dr. Amin Allam

[For more details, refer to "Introduction to Algorithms" by *Thomas Cormen*, et al.]
[For more details, refer to "Advanced Data Structures" by *Peter Brass*]

1 Interval tree

An *interval* $[lo, hi]$ consists of the *range* between lo and hi inclusive, where $lo \leq hi$. A *point* is a special case of an *interval* where $lo = hi$. Two *intervals* $[a \triangleright lo, a \triangleright hi]$ and $[b \triangleright lo, b \triangleright hi]$ *overlap* if and only if the *overlapping condition* is $a \triangleright lo \leq b \triangleright hi$ and $b \triangleright lo \leq a \triangleright hi$. A *dynamic interval tree* stores a set of *intervals* and allows the following operations:

- Insert an *interval* in $O(\log n)$ where n is the total number of stored *intervals*.
- Search for an *interval overlapping* with a query *interval* in $O(\log n)$.
- Search for all *intervals overlapping* with a query *interval* in $O(k \log n)$ where k is the number of stored *intervals* satisfying the query.

The *dynamic interval tree* is a *red-black* tree such that each *node* stores the interval *lo* and *hi* and *keyed* by *lo*. That is, an *inorder* (*depth first*) walk of the tree lists the *intervals* sorted by their low endpoints. The tree is *augmented* with an additional member $node \triangleright max$ which equals to the maximum *hi* of all *intervals* stored in the subtree *rooted* at *node*. The $node \triangleright max$ attribute can be easily updated while *insertion* and *deletion* without changing their $O(\log n)$ time complexity by the rule: $node \triangleright max = \max(node \triangleright hi, node \triangleright left \triangleright max, node \triangleright right \triangleright max)$.

The following procedure searches for an *interval overlapping* a *query interval* in $O(\log n)$:

```
node ← root
while (query does not overlap node)
    if (query▷lo > node▷left▷max)
    then node ← node▷right
    else node ← node▷left
return node▷interval
```

if $(query \triangleright lo > node \triangleright left \triangleright max)$ there is no point to search in the $node \triangleright left$ subtree since it can never *overlap* with any *interval* there, thus we only investigate $node \triangleright right$ subtree.

if $(query \triangleright lo \leq node \triangleright left \triangleright max)$, it is possible to find *overlapping intervals* with *query* in both $node \triangleright left$ and $node \triangleright right$ subtrees. However, if there exists an *interval overlapping* with *query* in $node \triangleright right$ subtree, there must be an *interval overlapping* with *query* in $node \triangleright left$ subtree *. Therefore it is safe to ignore $node \triangleright right$ subtree and only investigate $node \triangleright left$ subtree because we are mainly interested in finding one *interval overlapping* with *query*.

$*$ If ($query \triangleright lo \leq node \triangleright left \triangleright max$) and there exists an *interval overlapping* with *query* in $node \triangleright right$ subtree, there must be an *interval overlapping* with *query* in $node \triangleright left$ subtree.

Proof: We will prove that *query* overlaps with $[m \triangleright lo, m \triangleright hi]$ which is the *interval* in $node \triangleright left$ subtree having $m \triangleright hi = node \triangleright left \triangleright max$. Since $query \triangleright lo \leq node \triangleright left \triangleright max$, then $query \triangleright lo \leq m \triangleright hi$. To satisfy the *overlapping condition*, it remains to show that $m \triangleright lo \leq query \triangleright hi$ to prove that the *intervals query* and *m overlap*.

Suppose the *query interval overlaps* with an *interval* $[r \triangleright lo, r \triangleright hi]$ in $node \triangleright right$ subtree. From the *overlapping condition*: $query \triangleright hi \geq r \triangleright lo$. Since the tree is keyed by *lo*, $r \triangleright lo >$ the *lo* of all *intervals* in $node \triangleright left$ subtree. Therefore, $r \triangleright lo > m \triangleright lo$. Thus, $query \triangleright hi > m \triangleright lo$.

The following recursive procedure modifies the above procedure to search for the *interval* having the smallest *lo* endpoint *overlapping* with a *query interval* in $O(\log n)$. It differs from the above procedure only in checking the left subtree for *overlapping intervals* before checking the current node *interval*. The initial call should be Search(root, query):

```
Search(node, query)
---------------------------------------------------------
if (query▷lo ≤ node▷left▷max)
then
    result = Search(node▷left, query)
    if (result ≠ null) then return result
    if (query overlaps node) then return node▷interval
    return null
else
    if (query overlaps node) then return node▷interval
    return Search(node▷right, query)
```

Let the number of levels of the tree is $L = O(\log n)$. The time complexity of the above procedure is $S(L) = S(L-1) + O(1)$ where $S(1) = O(1)$. Thus, its time complexity is $L = O(\log n)$.

To search for all *intervals overlapping* with a *query interval*, we can execute one of the above procedures several times until it returns `null`, such that after each execution the found *interval* is removed from the tree. Thus, the complexity of such iterative procedure is $O(k \log n)$ where k is the number of stored *intervals overlapping* with the *query interval*.

It is possible to achieve that target without changing the tree structure, by storing - outside the tree - the *max* attribute values for nodes that need updating. The last procedure is easier to be used in such implementation, since in that case only the *max* attribute values of one path of the tree whose length is $O(\log n)$ need to be stored, and we know that the next *interval* to be retrieved is located somewhere on the stored path or to the right of it.

Since a *point* is a special case of an *interval*, all the above results hold if the *query* is a *point* instead of an *interval*, and it is required to retrieve *intervals* including the *query point*.

There is a variant of *static interval tree* which, given n *static* intervals, can be built in $O(n \log n)$ time and $O(n)$ space and allows the retrieval of all stored k *intervals overlapping* with a *query interval* in $O(\log n + k)$ time. But, it does not allow efficient insertion of new *intervals*.

2 Segment tree

Segment tree is a *static* data structure which stores *right-open* n *input intervals* in $O(n \log n)$ space and allows the retrieval of all stored k *input intervals overlapping* with a *query interval* in $O(\log n + k)$ time. It does not allow efficient insertion of new *intervals*. A *right-open interval* [*lo*, *hi*[consists of the *range* between its two *endpoints*: *lo* and *hi*, excluding *hi*, where *lo* < *hi*.

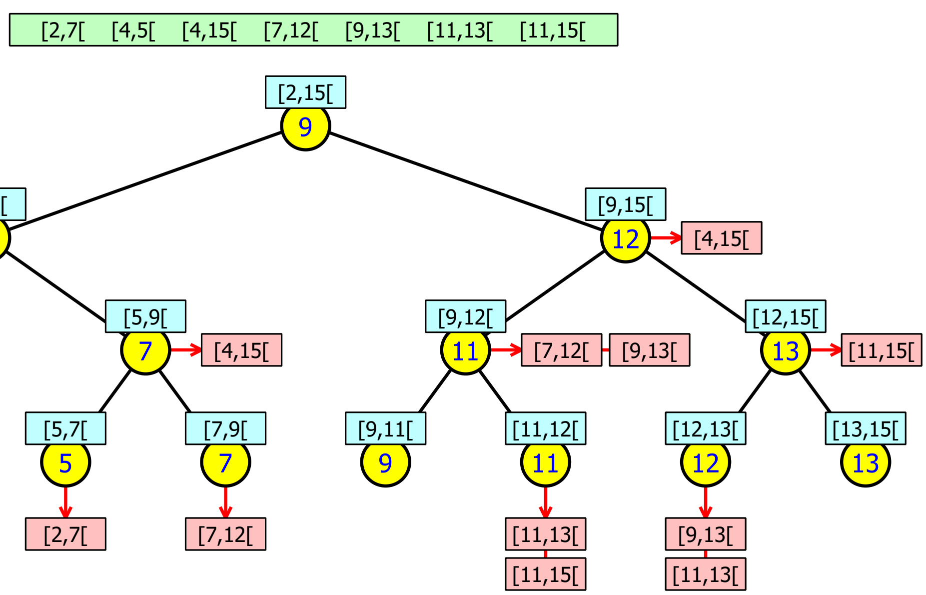
A *segment tree* is constructed in $O(n \log n)$ time by sorting the *endpoints* except the maximum one) of the n *input intervals* as keys of the lowest-level nodes of a *static binary tree*. Then, higher levels are constructed such that an *internal node* is keyed by the the smallest key of its right subtree.

Each *leaf node* corresponds to the right-open *segment tree interval* starting from its key to the key of its right adjacent *leaf* (or the *maximum* endpoint it is the right-most leaf). Each *internal node* corresponds to the *segment tree interval* of the union of the *segment tree intervals* of its children.

Note that the union of *segment tree interval* of all nodes in the same level is the right-open *segment tree interval* starting from the minimum *endpoint* to the maximum *endpoint*. Note also that all *segment tree intervals* of a subtree are subsets of the *segment tree interval* of its root.

Then, each *input interval* is represented as the union of the smallest number of *segment tree intervals* corresponding to *segment tree nodes*. Such representation is called the *canonical representation* of the *input interval* relative to that *segment tree*. Then each *input interval* is attached to all such *segment tree nodes* participated in its *canonical representation*.

The following figure contains *input interval* at the top light-green rectangle. Node keys are shown in yellow circles. *Segment tree intervals* corresponding to *segment tree nodes* are shown in light-blue rectangles. *Input Intervals* attached to *segment tree nodes* are shown in light-red rectangles.



The root corresponds to the *interval* $[2, 15[$ spanning the minimum and maximum endpoints of all *input intervals*. The root has the key 9 to indicate the separator between its left and right children *segment tree intervals* which are $[2, 9[$ and $[9, 15[$. The keys of the leaves are the endpoints of all *input intervals* except for the maximum. For example, the leaf having the key 5 corresponds to the *segment tree intervals* $[5, 7[$ where 7 is the key of the adjacent right leaf.

The *canonical representation* of the *input interval* $[4, 15[$ relative to the above *segment tree* are the *segment tree intervals* $[4, 5[$, $[5, 9[$, and $[9, 15[$ (light-blue rectangles). Thus, the *input interval* $[4, 15[$ (light-red rectangles) is attached to the *segment tree nodes* corresponding to such *segment tree intervals*. We described how to build the *segment tree*, except how to find *canonical representations* of *input intervals* to attach them to the corresponding *segment tree nodes*, which is described below:

To construct the *canonical representation* of an *input interval*, start from the *segment tree root* and consider the *segment tree interval* corresponding to the current *segment tree node*:

- If the *segment tree interval* is entirely contained in the *input interval*, attach the *input interval* to the *segment tree node* and stop following the path down.
- If the *segment tree interval* partially overlaps the *input interval*, follow both paths down.
- If the *segment tree interval* is disjoint from the *input interval*, stop following the path down.

Since the *segment tree* is balanced, the *segment tree* height is $O(\log n)$. The above procedure consumes $O(\log n)$ time because the maximum number of nodes investigated in each level is 4.

Proof: We will prove that there cannot be more than 2 nodes per level that both its paths need to be followed down. Thus, the maximum number of investigated nodes per level is 4. Suppose that there are ≥ 3 nodes to be investigated in the same level. All these nodes must be adjacent in the subtree (because *segment tree intervals* in the same level are sorted), and all these nodes (except for the farthest left and farthest right nodes) must be entirely contained in the *input interval* and their paths need not to be followed down.

The above proof also proves that the size of the *canonical representation* of each *input interval* is $O(\log n)$. Therefore, each *input interval* is attached to only $O(\log n)$ *segment tree nodes*. Thus, the *segment tree* space is $O(n \log n)$. Also, its construction time is $O(n \log n)$.

The following $O(\log n + k)$ procedure reports all *k input intervals overlapping* with *query interval*:

- If the *segment tree interval* is entirely contained in the *query interval*, report all the attached *input intervals* to all nodes in the subtree rooted by this *segment tree node*.
- If the *segment tree interval* partially overlaps the *query interval*, report all the attached *input intervals* to this *segment tree node* and follow both paths down.
- If the *segment tree interval* is disjoint from the *query interval*, stop following the path down.

If the *query* is a single *point*, *segment tree* allows the retrieval of all *k input intervals* including the *query point* in $O(\log n + k)$ time, starting from the root we follow one search tree path (as any binary search tree) and report all *input intervals* attached to the traversed *segment tree nodes*.

Segment tree can be recursively generalized to d dimensions by constructing a one-dimensional *segment tree* based on the first coordinate of all *intervals*, then attaching to every node a *segment tree* of the remaining $d - 1$ dimensions built on all *intervals* in the subtree rooted at that node.

Because all levels are filled with nodes (except for the right places of the last level), *segment trees* are *complete trees* and they can be implemented using arrays (similarly to implementing *heaps*). Simple mathematical formulas are used to traverse from parent to children and vice versa.

Segment tree is a powerful framework that can be modified to be used for other purposes, such as answering *dynamic range queries* as *range sum queries* and *range minimum queries* including *point* and *range updates*.