Interval Trees & Segment Trees

April 21, 2013
In Computer Science, an interval tree is an ordered tree data structure to hold intervals. Specifically, it allows one to efficiently find all intervals that overlap with any given interval or point. It is often used for windowing queries, for instance, to find all roads on a computerized map inside a rectangular viewport, or to find all visible elements inside a three-dimensional scene. A similar data structure is the segment tree.

We in our work has decided to insert in an interval tree using the different data structure RED BLACK TREES, reason for doing this is that it's easy to maintain the auxiliary information using logN extra work per operation.

Underlying data structure:
- Red-black trees will store intervals, keyed on “LOW”

Additional information to store:
- Store the maximum endpoint in the subtree rooted at node

To maintain the information:
- Update max as traverse down during insert
- Recalculate max after delete with a traversal up the tree
- Update during rotations
- Develop the desired new operations
Note that:
\[ x \rightarrow \text{max} = \max \left\{ \begin{array}{l}
  x \rightarrow \text{high} \\
  x \rightarrow \text{left} \rightarrow \max \\
  x \rightarrow \text{right} \rightarrow \max
\end{array} \right. \]
Insert in an interval

- Same as Binary Search Tree
- Inserted via RED BLACK TREE
- Keyed on the lower element on an interval
- Take logN time with red black tree
- LogN best in theory
INSERT IN AN INTERVAL TREE

RUN TIME vs N
Observations in interval

<table>
<thead>
<tr>
<th>N</th>
<th>RUNTIME(ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>0.000957</td>
</tr>
<tr>
<td>10000</td>
<td>0.001068</td>
</tr>
<tr>
<td>100000</td>
<td>0.001219</td>
</tr>
<tr>
<td>1000000</td>
<td>0.111511</td>
</tr>
</tbody>
</table>
Search an interval

• Query-?
• Given a number, search ONE interval consisting of it.
• Given a number, search ALL intervals consisting of it.
• First search takes \( \log N \) time, best in theory.
Algorithm
Single Search

IntervalSearch(T, i)
{
    x = T->root;
    while (x != NULL && !overlap(i, x->interval))
        if (x->left != NULL && x->left->max >= i->low)
            x = x->left;
        else
            x = x->right;
    return x
}
Correctness of IntervalSearch()

- Case 1: if search goes right, ∃ overlap in the right subtree or no overlap in either subtree
  - If ∃ overlap in right subtree, we’re done
  - Otherwise:
    - x→left = NULL, or x→left→max < x→low
    - Thus, no overlap in left subtree!

```c
while (x != NULL && !overlap(i, x->interval))
    if (x->left != NULL && x->left->max ≥ i->low)
        x = x->left;
    else
        x = x->right;
return x;
```
• Case 2: if search goes left, ∃ overlap in the left subtree or no overlap in either subtree
  ■ If ∃ overlap in left subtree, we’re done
  ■ Otherwise:
    ○ i →low ≤ x →left →max, by branch condition
    ○ x →left →max = y →high for some y in left subtree
    ○ Since i and y don’t overlap and i →low ≤ y →high, i →high < y →low
    ○ Since tree is sorted by low’s, i →high < any low in right subtree
    ○ Thus, no overlap in right subtree

```c
while (x != NULL && !overlap(i, x->interval))
    if (x->left != NULL && x->left->max ≥ i->low)
        x = x->left;
    else
        x = x->right;
return x;
```
All Interval Search

- Single Search takes $\log N$ time, but gives only one intersecting interval.
- All Interval Search takes $\log(N+k)$ time, where $k$ is the total number of intersecting intervals, with different design of data structure.
- Highly optimised, best possible we can do.
All Interval Search Construction

IntervalSearchConst( tree T) {
    Sort 2n midpoints of all the intervals
    Store the intervals in T that intersect the mid of the sorted array;
    Intervals that are completely left to the mid, store them in left child;
    Store others in the right child;

    IntervalSearchConst(tree T->left)
    IntervalSearchConst(tree T->right)
}
All Interval Search

IntervalSearchAll(node, x)
{
    Store each interval in two sorted list-
    L[], sorted by increasing left end point
    R[], sorted by decreasing right end point
    Search list depending on which side of xmed the query is on:
        If \( x < x_{med} \) then search L, output all until you find a left endpoint > \( x \).
        If \( x_q \geq x_{med} \) then search R, output all until you find a right endpoint < \( x \).
}
Observations

<table>
<thead>
<tr>
<th>N</th>
<th>RUNTIME(ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>0.021</td>
</tr>
<tr>
<td>100</td>
<td>0.933</td>
</tr>
<tr>
<td>1000</td>
<td>0.014</td>
</tr>
<tr>
<td>1500</td>
<td>0.01525</td>
</tr>
</tbody>
</table>
Segment Trees

• A segment tree is a data structure for storing a set of intervals
• \( I = \{[x_1, x_1], [x_2, x_2], \ldots, [x_n, x_n]\} \) and can be used for solving problems e.g. concerning line segments.
• Let \( p_1, \ldots, p_m, m \leq 2n, \) be the ordered list of distinct endpoints of the intervals in \( I. \) The ordered sequence of endpoints \( p_1, \ldots, p_m \) partitions the real line into a set of atomic intervals \((−∞, p_1), [p_1, p_1], (p_1, p_2), [p_2, p_2], \ldots, (p_{n-1}, p_n), [p_n, p_n], (p_n, ∞)\)
• A segment tree is a balanced tree where each node corresponds to an interval. The leaves correspond to the atomic intervals according to left to right order. An internal node \( u \) corresponds to the union of the intervals corresponding to the leaves of the subtree rooted at \( u. \)
Algorithm
Insert

• Take NlogN Runtime

\[
\text{INSERT SEGMENT TREE}(v, [x, x])
\]
\[\text{if int}(v) \subseteq [x, x]\]
\[\text{then add } [x, x] \text{ to } I(v)\]
\[\text{else if int}(\text{lc}(v)) \cap [x, x] = 0\]
\[\text{then INSERT SEGMENT TREE} \text{(lc}(v), [x, x])\]
\[\text{If int}(\text{rc}(v)) \cap [x, x] = 0\]
\[\text{then INSERT SEGMENT TREE} \text{(rc}(v), [x, x])\]
# Observations

<table>
<thead>
<tr>
<th>N</th>
<th>RUNTIME (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>0.283</td>
</tr>
<tr>
<td>10000</td>
<td>2.011</td>
</tr>
<tr>
<td>100000</td>
<td>7.818</td>
</tr>
<tr>
<td>500000</td>
<td>3.8400</td>
</tr>
<tr>
<td>1000000</td>
<td>75.881</td>
</tr>
</tbody>
</table>
Algorithm
Search

search(int low, int high, seg_tree* temp)
{
    if (temp->start == low && temp->end == high)
        return temp->sum;
    else if (temp->left->start <= low && high <= temp->left->end)
        return search(low, high, temp->left);
    else if (temp->right->start <= low && high <= temp->right->end)
        return search(low, high, temp->right);
    else
        return search(low, temp->left->end, temp->left)
            + search(temp->left->end + 1, high, temp->right);
}

## Observations

<table>
<thead>
<tr>
<th>N</th>
<th>RUNTIME(ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>0.001223</td>
</tr>
<tr>
<td>10000</td>
<td>0.001306</td>
</tr>
<tr>
<td>100000</td>
<td>0.001350</td>
</tr>
<tr>
<td>500000</td>
<td>0.001400</td>
</tr>
<tr>
<td>1000000</td>
<td>0.001484</td>
</tr>
</tbody>
</table>
Algorithm
Update

update(seg_tree* temp, int index, int diff)
{
    temp->sum += diff;
    if (temp->left!=NULL && index>=temp->left->start && index <=temp->left->end)
        update(temp->left,index,diff);
    else if (temp->right!=NULL && index>=temp->right->start && index <=temp->right->end)
        update(temp->right,index,diff);
}
## Observations

<table>
<thead>
<tr>
<th>N</th>
<th>RUNTIME(ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>0.001354</td>
</tr>
<tr>
<td>10000</td>
<td>0.001413</td>
</tr>
<tr>
<td>100000</td>
<td>0.001484</td>
</tr>
<tr>
<td>500000</td>
<td>0.001500</td>
</tr>
<tr>
<td>1000000</td>
<td>0.001577</td>
</tr>
</tbody>
</table>
Query Used?

- Given an array, to find the sum between any two indices and update the array content
- Running over a loop -
  \[
  \text{SUM} \; \Theta(n) \quad \text{UPDATE} \; \Theta(1)
  \]
- Prefix Sum -
  \[
  \text{SUM} \; \Theta(1) \quad \text{UPDATE} \; \Theta(n)
  \]
  With Segment trees
  Sum=Update=\Theta(\log n)