# Everything Under The Sun

## A blog on CS concepts

# A simple approach to segment trees, part 2

*January 10, 2015* *by* *Kartik Kukreja*

This post is in continuation to a previous post on segment trees (https://kartikkukreja.wordpress.com/2014/11/09/a-simple-approach-to-segment-trees/). If you haven't read that post already, it would be a good idea to read that first.

Let me do a quick recap. The previous post described three use cases for segment trees (persistent/static, point updates and range updates) and explained the first two, leaving the last as the subject matter for this post. This post will describe how we can use range updates with segment trees, lazy propagation and various optimizations possible.

I tried to come up with a single template for segment trees which supported all the 3 use cases, had lazy propagation and all the optimizations that come with it but finally I decided it wasn't worth the effort for following reasons:

- The more general I made the template, the slower it ran. Most problems which require segment trees with lazy propagation have very strict time constraints on online judges and I had to run through hoops and do some serious gymnastics to get the solutions to pass.
- Supporting different flavors of lazy propagation and various optimizations places huge requirements on the SegmentTreeNode (described in previous post), requiring it to implement as many as 10 functions for any problem. I guess there comes a point when the cure becomes more harmful than the disease.
- Pretty much every new problem required me to generalize the template in some way or add more functions to SegmentTreeNode. This became unusable pretty soon.

For these reasons, I decided that different ideas in lazy propagation and various optimizations that come with it are hard to combine, to say the least, and probably not worth the effort. So I'll take a different approach. I'll start with a partial template and fill bits and pieces of it to demonstrate different ideas.

```
1  struct SegmentTreeNode {
2    int start, end; // this node is responsible for the segment [start...end]
3
4    // variables to store aggregate statistics and
5    // any other information required to merge these
6    // aggregate statistics to form parent nodes
7
```

```
8   void assignLeaf(InputType value) {
9     // InputType is the type of input array element
10    // Given the value of an input array element,
11    // build aggregate statistics for this leaf node
12  }
13
14  void merge(SegmentTreeNode& left, SegmentTreeNode& right) {
15    // merge the aggregate statistics of left and right
16    // children to form the aggregate statistics of
17    // their parent node
18  }
19
20  OutputType query() {
21    // OutputType is the type of the required aggregate statistic
22    // return the value of required aggregate statistic
23    // associated with this node
24  }
25 };
```

**view raw Partial Segment Tree Node.cpp** hosted with ♥ by **GitHub**

```
1  template<class InputType, class UpdateType, class OutputType>
2  class SegmentTree {
3    SegmentTreeNode* nodes;
4    int N;
5
6  public:
7    SegmentTree(InputType arr[], int N) {
8      this->N = N;
9      nodes = new SegmentTreeNode[getSegmentTreeSize(N)];
10     buildTree(arr, 1, 0, N-1);
11   }
12
13   ~SegmentTree() {
14     delete[] nodes;
15   }
16
17   // get the value associated with the segment [start...end]
18   OutputType query(int start, int end) {
19     SegmentTreeNode result = query(1, start, end);
20     return result.query();
21   }
22
23   // range update: update the range [start...end] by value
```

```
24    // Exactly what is meant by an update is determined by the
25    // problem statement and that logic is captured in segment tree node
26    void update(int start, int end, UpdateType value) {
27      update(1, start, end, value);
28    }
29
30  private:
31    void buildTree(InputType arr[], int stIndex, int start, int end) {
32      // nodes[stIndex] is responsible for the segment [start...end]
33      nodes[stIndex].start = start, nodes[stIndex].end = end;
34
35      if (start == end) {
36        // a leaf node is responsible for a segment containing only 1 element
37        nodes[stIndex].assignLeaf(arr[start]);
38        return;
39      }
40
41      int mid = (start + end) / 2,
42          leftChildIndex = 2 * stIndex,
43          rightChildIndex = leftChildIndex + 1;
44
45      buildTree(arr, leftChildIndex, start, mid);
46      buildTree(arr, rightChildIndex, mid + 1, end);
47      nodes[stIndex].merge(nodes[leftChildIndex], nodes[rightChildIndex]);
48    }
49
50    int getSegmentTreeSize(int N) {
51      int size = 1;
52      for (; size < N; size <<= 1);
53      return size << 1;
54    }
55
56    SegmentTreeNode query(int stIndex, int start, int end) {
57      // we'll fill this in later
58    }
59
60    void update(int stIndex, int start, int end, UpdateType value) {
61      // we'll fill this in later
62    }
63  };
```

**view raw Partial Segment Tree Template.cpp** hosted with ♥ by **GitHub**

We now only have to describe how to fill in update() and query() methods.

The previous post provided us a way to update a single element in O(log N) time. We can use it to achieve range update in O(N log N) by calling the point update function iteratively for each element in the given range. But we can do better! We can in fact do a range update in O(N). The range update function is very similar to the buildTree() function.

```
1  void update(int stIndex, int start, int end, UpdateType value) {
2    if (start == end) {
3      nodes[stIndex].applyUpdate(value);
4      return;
5    }
6
7    int mid = (nodes[stIndex].start + nodes[stIndex].end) / 2,
8        leftChildIndex = 2 * stIndex,
9        rightChildIndex = leftChildIndex + 1;
10
11   if (start > mid)
12     update(rightChildIndex, start, end, value);
13   else if (end <= mid)
14     update(leftChildIndex, start, end, value);
15   else {
16     update(leftChildIndex, start, mid, value);
17     update(rightChildIndex, mid+1, end, value);
18   }
19   nodes[stIndex].merge(nodes[leftChildIndex], nodes[rightChildIndex]);
20 }
```

**view raw Range update.cpp** hosted with ❤ by **GitHub**

This along with the query() function defined in the previous post allows us to do range updates in O(N) and range queries in O(log N). Note that this requires us to add a function `void applyUpdate(UpdateType value)` to our SegmentTreeNode.

Let's see it in use on a sample problem.

**Problem:**

*Given an array A of N floating point values, support two operations on any range A[a..b] (0<=a<=b<N):*

- *replace each A[i] (a<=i<=b) by sqrt(A[i])*
- *find sum A[a]+…+A[b]*

**Solution:** The SegmentTreeNode for this problem looks like this:

```
1  struct SegmentTreeNode {
2    int start, end; // this node is responsible for the segment [start...end]
3    double total;
4
5    void assignLeaf(double value) {
```

```
6     total = value;
7   }
8
9   void merge(SegmentTreeNode& left, SegmentTreeNode& right) {
10    total = left.total + right.total;
11  }
12
13  double query() {
14    return total;
15  }
16
17  // the value of the update is dummy in this case
18  void applyUpdate(bool value) {
19    total = sqrt(total);
20  }
21 };
```

**view raw Range update example node.cpp** hosted with ❤ by **GitHub**
The complete solution for this problem is available here (https://github.com/kartikkukreja/blog-codes/blob/master/src/segment-tree-range-update-sample-problem.cpp).

For most problems, however, we can't get away with an update function having linear complexity and must do better. This is where lazy propagation comes in. The basic idea of lazy propagation is to hold off propagating updates down the tree and propagate them only when absolutely necessary. Lazy propagation can be built into any of update() or query() functions but here I'll describe it only for update(). There are several reasons for why we might be able to hold off updates in internal nodes:

- Some updates are not really required and can be thrown away. Doing nothing is awesome!
- For some problems, updates can be applied to internal nodes (segments/ranges) as opposed to leaves (actual array elements).
- For some problems, updates can be accumulated in internal nodes until they cross a threshold and only afterwards must they be propagated.

Let's see some examples of these ideas on actual problems.

**Problem (GSS4 (http://www.spoj.com/problems/GSS4/)):**

*Given an array A of N integers, support two operations on any range A[a..b] (0<=a<=b<N):*

- *replace each A[i] (a<=i<=b) by floor(sqrt(A[i]))*
- *find sum A[a]+…+A[b]*

**Solution:**

There is only a small difference between this problem and the previous one and that is that all array elements are guaranteed to be integers, even after applying updates. This provides us an opportunity for optimization: There's only so many times you can take square root before a number reduces to 1 and once a number reduces to 1, it stays 1 forever. So, we can throw out updates for nodes which are 1. This problem demonstrates the first idea that we can sometimes throw away updates.

We can modify the update() function such that at each internal node, it decides whether to propagate an update or throw it out. The query() function remains unchanged.

```
1  void update(int stIndex, int start, int end, UpdateType value) {
2    if (nodes[stIndex].start == start && nodes[stIndex].end == end) {
3      lazyPropagatePendingUpdateToSubtree(stIndex, value);
4      return;
5    }
6
7    int mid = (nodes[stIndex].start + nodes[stIndex].end) >> 1,
8        leftChildIndex = stIndex << 1,
9        rightChildIndex = leftChildIndex + 1;
10
11   if (start > mid)
12     update(rightChildIndex, start, end, value);
13   else if (end <= mid)
14     update(leftChildIndex, start, end, value);
15   else {
16     update(leftChildIndex, start, mid, value);
17     update(rightChildIndex, mid+1, end, value);
18   }
19   nodes[stIndex].merge(nodes[leftChildIndex], nodes[rightChildIndex]);
20 }
21
22 void lazyPropagatePendingUpdateToSubtree(int stIndex, UpdateType value) {
23   nodes[stIndex].addUpdate(value);
24   if (!nodes[stIndex].isPropagationRequired())
25     return;
26
27   if (nodes[stIndex].start == nodes[stIndex].end) {
28     nodes[stIndex].applyPendingUpdate();
29     return;
30   }
31
32   UpdateType pendingUpdate = nodes[stIndex].getPendingUpdate();
33   nodes[stIndex].clearPendingUpdate();
34   int mid = (nodes[stIndex].start + nodes[stIndex].end) >> 1,
35       leftChildIndex = stIndex << 1,
36       rightChildIndex = leftChildIndex + 1;
37
38   lazyPropagatePendingUpdateToSubtree(leftChildIndex, pendingUpdate);
39   lazyPropagatePendingUpdateToSubtree(rightChildIndex, pendingUpdate);
40   nodes[stIndex].merge(nodes[leftChildIndex], nodes[rightChildIndex]);
```

```
41  }
```

**view raw GSS4 update().cpp** hosted with ❤ by **GitHub**

I've tried to make the update() function as general as I could so that it works for several similar problems, without requiring any change. However, this requires 5 new functions to be implemented in SegmentTreeNode, although they are all 1-liners. The SegmentTreeNode for this problem looks like this:

```
1   struct SegmentTreeNode {
2     int start, end; // this node is responsible for the segment [start...end]
3     ll total;
4     bool pendingUpdate;
5
6     SegmentTreeNode() : total(0), pendingUpdate(false) {}
7
8     void assignLeaf(ll value) {
9       total = value;
10    }
11
12    void merge(SegmentTreeNode& left, SegmentTreeNode& right) {
13      total = left.total + right.total;
14    }
15
16    ll query() {
17      return total;
18    }
19
20    // For this particular problem, propagation is not required
21    // if all elements in this segment are 1's
22    bool isPropagationRequired() {
23      return total > end-start+1;
24    }
25
26    void applyPendingUpdate() {
27      total = (ll) sqrt(total);
28      pendingUpdate = false;
29    }
30
31    // For this particular problem, the value of the update is dummy
32    // and is just an instruction to square root the leaf value
33    void addUpdate(bool value) {
34      pendingUpdate = true;
35    }
36
37    // returns a dummy value
```

```
38    bool getPendingUpdate() {
39      return true;
40    }
41
42    void clearPendingUpdate() {
43      pendingUpdate = false;
44    }
45  };
```

**view raw GSS4 SegmentTreeNode.cpp** hosted with ❤ by **GitHub**

The complete solution for this problem is available here (https://github.com/kartikkukreja/blog-codes/blob/master/src/spoj_GSS4.cpp). Note that I had to define the nodes[] array outside the SegmentTree template. The time limit on this problem is too strict to allow memory allocation/deallocation per test case.

**Problem (HORRIBLE (http://www.spoj.com/problems/HORRIBLE/)):**

*Given an array A of N integers, support two operations on any range A[a..b] (0<=a<=b<N):*

- *add a given value v to each A[i] (a<=i<=b)*
- *find sum A[a]+...+A[b]*

**Solution:**

This problem demonstrates the second idea: for some problems, updates can be applied to (or stored inside) internal nodes. Any node whose segment is completely covered by the update segment can just store the update value and not propagate it down the tree. The query() function, while coming back up the tree, picks up pending updates and applies them to the result.

```
1   SegmentTreeNode query(int stIndex, int start, int end) {
2     if (nodes[stIndex].start == start && nodes[stIndex].end == end) {
3       SegmentTreeNode result = nodes[stIndex];
4       if (result.hasPendingUpdate())
5         result.applyPendingUpdate();
6       return result;
7     }
8
9     int mid = (nodes[stIndex].start + nodes[stIndex].end) >> 1,
10        leftChildIndex = stIndex << 1,
11        rightChildIndex = leftChildIndex + 1;
12    SegmentTreeNode result;
13
14    if (start > mid)
15      result = query(rightChildIndex, start, end);
16    else if (end <= mid)
17      result = query(leftChildIndex, start, end);
18    else {
19      SegmentTreeNode leftResult = query(leftChildIndex, start, mid),
```

```
20                  rightResult = query(rightChildIndex, mid+1, end);
21    result.start = leftResult.start;
22    result.end = rightResult.end;
23    result.merge(leftResult, rightResult);
24   }
25
26   if (nodes[stIndex].hasPendingUpdate()) {
27    result.addUpdate(nodes[stIndex].getPendingUpdate());
28    result.applyPendingUpdate();
29   }
30   return result;
31  }
32
33  void update(int stIndex, int start, int end, UpdateType value) {
34   if (nodes[stIndex].start == start && nodes[stIndex].end == end) {
35    nodes[stIndex].addUpdate(value);
36    return;
37   }
38
39   int mid = (nodes[stIndex].start + nodes[stIndex].end) >> 1,
40    leftChildIndex = stIndex << 1,
41    rightChildIndex = leftChildIndex + 1;
42
43   if (start > mid)
44    update(rightChildIndex, start, end, value);
45   else if (end <= mid)
46    update(leftChildIndex, start, end, value);
47   else {
48    update(leftChildIndex, start, mid, value);
49    update(rightChildIndex, mid+1, end, value);
50   }
51   nodes[stIndex].merge(nodes[leftChildIndex], nodes[rightChildIndex]);
52  }
```

**view raw HORRIBLE query() and update().cpp** hosted with ❤ by **GitHub**

The SegmentTreeNode for this problem becomes:

```
1   struct SegmentTreeNode {
2    int start, end; // this node is responsible for the segment [start...end]
3    ll total, pendingUpdate;
4
5    SegmentTreeNode() : total(0), pendingUpdate(0) {}
6
7    void assignLeaf(ll value) {
```

```
8     total = value;
9   }
10
11  void merge(SegmentTreeNode& left, SegmentTreeNode& right) {
12    total = left.total + right.total;
13    if (left.pendingUpdate > 0)
14      total += left.pendingUpdate * (left.end - left.start + 1);
15    if (right.pendingUpdate > 0)
16      total += right.pendingUpdate * (right.end - right.start + 1);
17  }
18
19  ll query() {
20    return total;
21  }
22
23  bool hasPendingUpdate() {
24    return pendingUpdate != 0;
25  }
26
27  void applyPendingUpdate() {
28    total += (end - start + 1) * pendingUpdate;
29    pendingUpdate = 0;
30  }
31
32  void addUpdate(ll value) {
33    pendingUpdate += value;
34  }
35
36  ll getPendingUpdate() {
37    return pendingUpdate;
38  }
39 };
```

**view raw HORRIBLE SegmentTreeNode.cpp** hosted with ❤ by **GitHub**

Note how this change places special requirement on the merge() function. When we store an update inside an internal node, we ensure that queries to all other nodes (its ancestors and descendants) return correct results by:

- modifying the query() function such that it collects pending updates while going back up the tree (recursion unrolling). This ensures that all descendants of a node will see an update made to that node.
- modifying the merge() function such that update() function correctly updates all ancestor nodes of an updated node while going back up the tree (recursion unrolling).

The complete solution for this problem is available here (https://github.com/kartikkukreja/blog-codes/blob/master/src/spoj_HORRIBLE.cpp).

**Problem (FLIPCOIN (http://www.codechef.com/problems/FLIPCOIN/)):**

*Given an array A of N booleans, support two operations on any range A[a..b] (0<=a<=b<N):*

- *flip each A[i] (a<=i<=b)*
- *count how many of A[a],…,A[b] are true*

**Solution:**

This problem is exactly similar to the previous problem. Here also, we can store updates in internal nodes, modify the merge() function to correctly update ancestors and query() function to correctly update descendants. The SegmentTree template for the previous problem can be used exactly as is for this problem. The only change is to SegmentTreeNode:

```
1  struct SegmentTreeNode {
2    int start, end; // this node is responsible for the segment [start...end]
3    int count;
4    bool pendingUpdate;
5
6    SegmentTreeNode() : count(0), pendingUpdate(false) {}
7
8    void assignLeaf(bool value) {}
9
10   void merge(SegmentTreeNode& left, SegmentTreeNode& right) {
11     count = (left.pendingUpdate ? (left.end - left.start + 1 - left.count) : left.count)
12           + (right.pendingUpdate ? (right.end - right.start + 1 - right.count) : right.count);
13   }
14
15   int query() {
16     return count;
17   }
18
19   bool hasPendingUpdate() {
20     return pendingUpdate;
21   }
22
23   void applyPendingUpdate() {
24     count = (end - start + 1) - count;
25     pendingUpdate = false;
26   }
27
28   void addUpdate(bool value) {
29     pendingUpdate = !pendingUpdate;
30   }
31
```

```
32    bool getPendingUpdate() {
33      return true;
34    }
35  };
```

**view raw FLIPCOIN SegmentTreeNode.cpp** hosted with ♥ by **GitHub**

The complete solution for this problem is available here (https://github.com/kartikkukreja/blog-codes/blob/master/src/codechef_FLIPCOIN.cpp).

I hope this post provided a good introduction to range updates and lazy propagation in segment trees. There are some ideas that I've skipped, which are either very similar to what is discussed in this post (like lazy propagation in query() instead of in update()) or are rare (like holding off updates in internal nodes, until they cross a threshold, after which they should be lazily propagated). Both of these are natural extensions of what is described in the post. There may also exist some other ideas that I'm not yet aware of. Please share your thoughts, feedback and suggestions in comments.



*This entry was posted in Algorithms, Data Structures, Programming, Spoj and tagged FLIPCOIN, GSS4, HORRIBLE, lazy propagation, query, range updates, segment tree, update. Bookmark the permalink.*

# 22 thoughts on "A simple approach to segment trees, part 2"

1. Pingback: A simple approach to segment trees | Everything Under The Sun

2. madhur mishra says:
   *May 28, 2015 at 8:57 am*
   The tutorial is really awesome.

   Reply
   - kartik kukreja says:
     *May 28, 2015 at 3:32 pm*
     Thanks

     Reply
3. Lim Yu De says:
   *June 1, 2015 at 8:55 am*
   Really well explained. Easy to understand. Thanks for the tutorial!

   Reply
4. Karmic says:
   *June 3, 2015 at 1:04 pm*
   The explanation is excellent. Best i know on web .
   Please , if possible , give an example on lazy propagation on query and that of internal nodes with threshhold concept .

   Reply
   - Dipanker Singh says:
     *March 20, 2016 at 12:48 pm*
     yeah ! please !

     Reply
5. Mahatma Gandhi says:
   *July 4, 2015 at 7:09 pm*
   How can we change the value of a range [x,y] to v , in logarithmic time? Eg, 1 4 7 would mean changing the values of all elements in indices 1 to 4 to 7.

   Reply
   - kartik kukreja says:
     *July 4, 2015 at 7:27 pm*
     We can achieve this through lazy propagation. We'll lazily propagate the updates, stopping at the first segment that is completely covered by the update range and storing it there. Then, while querying if we go down that path, we can propagate the update downwards, essentially hiding the update time behind several query calls.

     Reply
6. sulabh kumar says:
   *July 10, 2015 at 4:50 pm*
   hey..i was trying "HORRIBLE" problem without lazy propagation but range update is not working properly.but update function from previous post is working fine.
   could you please take a look.
   http://ideone.com/K41G9r

   Reply
   - kartik kukreja says:

*July 11, 2015 at 5:02 pm*
Line 135 should be:
if (nodes[stIndex].start == nodes[stIndex].end)

Reply

7. Illusionist says:
*July 11, 2015 at 5:26 pm*
How can we multiply each element of a range [x,y] by v , in logarithmic time? Eg, 1 4 7 would mean multiplying the values of all elements in indices 1 to 4 by 7.

Reply

- Illusionist says:
*July 12, 2015 at 1:04 am*
I got it !! Thanks anyways !!

Reply

- sulabh kumar says:
*July 12, 2015 at 12:14 pm*
thank you for takin the time…:)

Reply

8. Koushik says:
*July 25, 2015 at 11:35 pm*
Nice post… Very Useful…

Reply

9. Ido Hadanny says:
*September 7, 2015 at 10:51 am*
Hey Kartik, don't you think that in FLIPCOIN, it's nicer that the value would be a tuple with (#heads, #tails)? or (#heads, #total)? this way, the merge function would be self-contained.
Note that in your code, you break the encapsulation of "merge" and outside it you handle the total number of coins:

result.start = leftResult.start;
result.end = rightResult.end;

WDYS?

Reply

- kartik kukreja says:
*September 7, 2015 at 12:35 pm*
Hey, in your earlier comment, you said that storing segment start and end indices in SegmentTreeNode is better for encapsulation and code reuse. In this problem, there's added benefit to storing them in the node because we needed them to calculate the statistics.

I didn't understand a few things about your question. Are you saying that the argument of assignLeaf() and addUpdate() should remain a boolean value but we should store the number of heads & tails in each node? Isn't what is done almost the same? count stores the number of heads and we use (end-start+1) to calculate the total number of coins.

Reply

10. Shubham Kashyap says:
    *September 18, 2015 at 11:14 am*
    Sir I don't understand the use of start and end in SegmentTreeNode. Like in last post we are not doing anything like this?

    Reply
    - kartik kukreja says:
      *September 19, 2015 at 9:07 pm*
      When we store start and end indices in SegmentTreeNode, we don't have to pass those as arguments to query() and update() methods. Basically these are two ways to accomplish the same thing:
      1. Either you pass the start and end indices of the current interval to query() and update() methods, which looks more messy.
      2. Or store the start and end indices in the node itself and then you only have to pass the node index to these methods. It has added benefits if a node has to use the start and end index values for query or update (for an example, look at the solution of the problem HORRIBLE).

      Reply
11. kaku09 says:
    *March 20, 2016 at 12:41 pm*
    can you please give a link to the question in which we have to use the third idea ?
    i.e. in which updates can be accumulated in internal nodes until they cross a threshold and only afterwards must they be propagated.

    Reply
    - Kartik Kukreja says:
      *March 20, 2016 at 10:01 pm*
      I'm sure there will be many problems that utilize this concept. A sample problem that I was able to think of: Given an integer array, support two operations: adding an integer to every element in a segment and querying how many elements are larger than X (a constant) in a segment. For this problem, we could store two values in each node: maximum value of any element in this segment and the sum of pending updates. Until the max + updates reaches X, we don't have to propagate updates.

      Reply
12. Anonymous says:
    *January 28, 2017 at 1:39 pm*
    what do u do if there are two kinds of updates like sqrt and add v to all in range l to r

    Reply
    - Kartik Kukreja says:
      *January 29, 2017 at 11:23 pm*
      I think it would depend on the problem but in general, you will store some values for both updates in each node. In this example, how do you apply sqrt? It's not an aggregate operation. Just sqrt all elements in a range?

      Reply