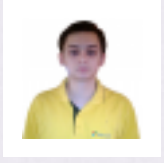
Everything Under The Sun

A blog on CS concepts



A simple approach to segment trees

November 9, 2014 by *kartik kukreja*

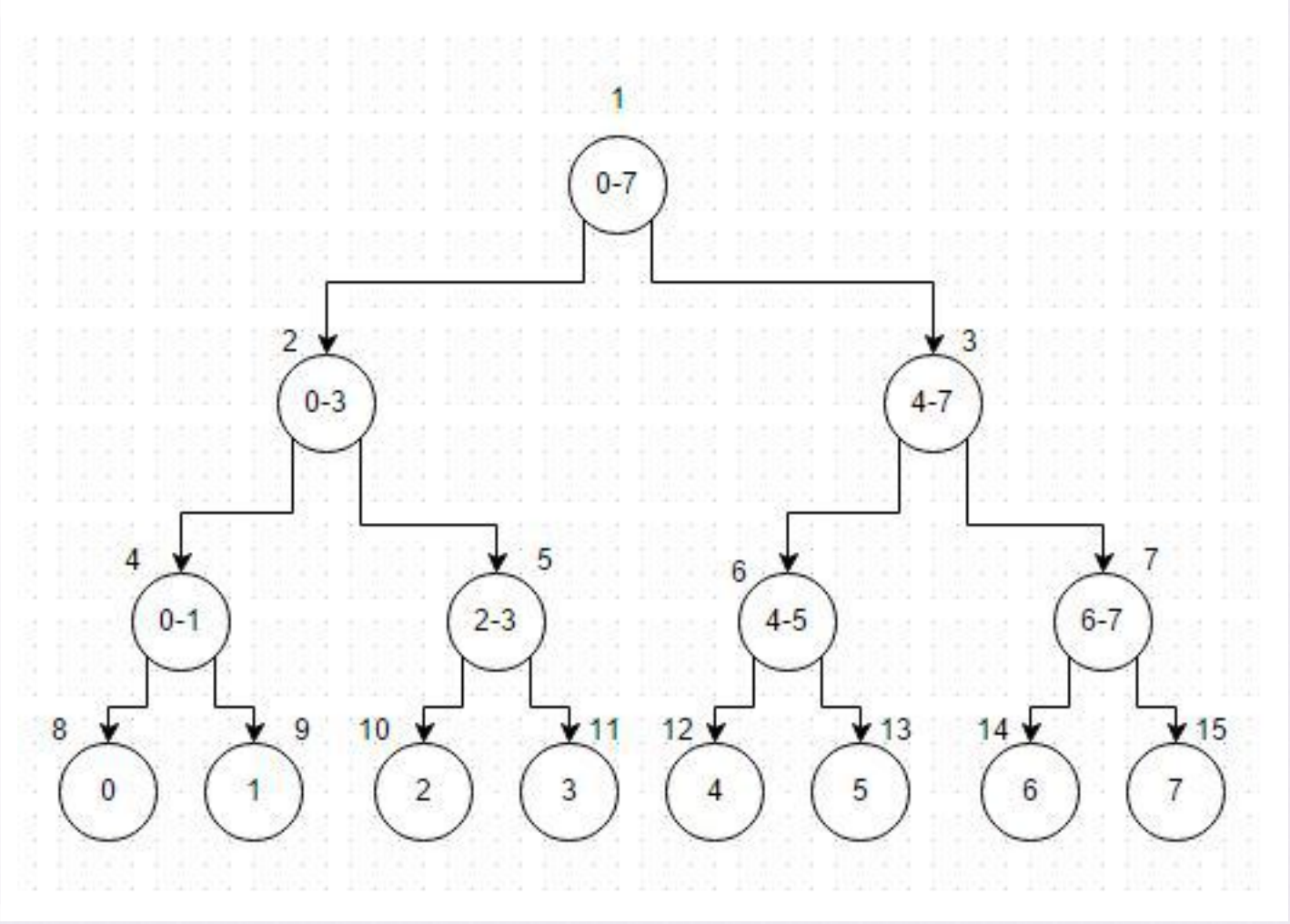
A segment tree is a tree data structure that allows aggregation queries and updates over array intervals in logarithmic time. As I see it, there are three major use cases for segment trees:

1. **Static or persistent segment trees:** This is probably the most common use case. We preprocess an array of N elements to construct a segment tree in O(N). Now, we can query aggregates over any arbitrary range/segment of the array in O(log N).
2. **Segment tree with point updates:** This allows us to update array values, one at a time in O(log N), while still maintaining the segment tree structure. Queries over any arbitrary range still occurs in O(log N).
3. **Segment tree with range updates:** This allows us to update a range of array elements at once in O(N) in the worst case, however problem specific optimizations and lazy propagation typically give huge improvements. Queries over any arbitrary range still occurs in O(log N).

In this post, I'll cover the first two use cases because they go together. Given a static segment tree, it is very easy to add point update capability to it. I'll leave the third use case as the subject matter of a future blog post. I intend this post to be a practical introduction to segment trees, rather than a theoretical description, so it will focus on how we can divide a segment tree into its components, the working of each component and how we can separate the problem specific logic from the underlying data structure. We'll build a template for a segment tree and then apply it to several problems to understand how problem specific logic can be cleanly separated from the template.

Structure of a segment tree

Let's understand what a segment tree looks like. Each node in a segment tree stores aggregate statistics for some range/segment of an array. The leaf nodes stores aggregate statistics for individual array elements. Although a segment tree is a tree, it is stored in an array similar to a heap. If the input array had 2^n elements (i.e., the number of elements were a power of 2), then the segment tree over it would look something like this:



ARCHIVES

January 2015 (1)
December 2014 (1)
November 2014 (2)
June 2014 (1)
March 2014 (1)
December 2013 (3)
November 2013 (2)
October 2013 (4)
September 2013 (2)
August 2013 (4)
July 2013 (2)
June 2013 (9)
May 2013 (11)
April 2013 (9)
March 2013 (5)

CATEGORIES

Algorithms (37)
Artificial Intelligence (6)
Coursera (9)
Data Structures (6)
Hackerrank (6)
MOOC (9)
Programming (2)

Each node here shows the segment of the input array for which it is responsible. The number outside a node indicates its index in the segment tree array. Clearly, if the array size N were a power of 2, then the segment tree would have 2*N-1 nodes. It is simpler to store the first node at index 1 in the segment tree array in order to simplify the process of finding indices of left and right children (a node at index i has left and right children at 2*i and 2*i+1 respectively). Thus, for an input array of size N, an array of size 2*N would be required to store the segment tree.

In practice, however, N is not usually a power of 2, so we have to find the power of 2 immediately greater than N, let's call it x, and allocate an array of size 2*x to store the segment tree. The following procedure calculates the size of array required to store a segment tree for an input array size N:

```
1  int getSegmentTreeSize(int N) {
2          int size = 1;
3          for (; size < N; size <<= 1);
4          return size << 1;
5  }
```

Size of a segment tree hosted with ♥ by **GitHub** **view raw**

We'll try to separate the implementation of the underlying data structure from the problem specific logic. For this purpose, let us define a structure for a segment tree node:

```
1  struct SegmentTreeNode {
2      // variables to store aggregate statistics and
```

Spoj (3)
Uncategorized (4)

BLOG STATS

96,353 pageviews

```
    // any other information required to merge these
    // aggregate statistics to form parent nodes

    void assignLeaf(T value) {
        // T is the type of input array element
        // Given the value of an input array element,
        // build aggregate statistics for this leaf node
    }

    void merge(SegmentTreeNode& left, SegmentTreeNode& right) {
        // merge the aggregate statistics of left and right
        // children to form the aggregate statistics of
        // their parent node
    }

    V getValue() {
        // V is the type of the required aggregate statistic
        // return the value of required aggregate statistic
        // associated with this node
    }
};
```

Segment tree node hosted with ♥ by **GitHub** **view raw**

Building a segment tree

We can build a segment tree recursively in a depth first manner, starting at the root node (representative of the whole input array), working our way towards the leaves (representatives of individual input array elements). Once both children of a node have returned, we can merge their aggregate statistics to form their parent node.

```
void buildTree(T arr[], int stIndex, int lo, int hi) {
        if (lo == hi) {
                nodes[stIndex].assignLeaf(arr[lo]);
                return;
        }

        int left = 2 * stIndex, right = left + 1, mid = (lo + hi) / 2;
        buildTree(arr, left, lo, mid);
        buildTree(arr, right, mid + 1, hi);
        nodes[stIndex].merge(nodes[left], nodes[right]);
}
```

Building a segment tree hosted with ♥ by **GitHub** **view raw**

Here I've assumed that the type of input array elements is T. stIndex represents the index of current segment tree node in the segment tree array, lo and hi indicate the range/segment of input array this node is responsible for. We build the whole segment tree with a single call to buildTree(arr, 1, 0, N-1), where N is the size of input array arr. Clearly, the time complexity of this procedure is O(N), assuming that assignLeaf() and merge() operations work in O(1).

Querying the segment tree

Suppose we want to query the aggregate statistic associated with the segment [lo,hi], we can do this recursively as follows:

```
// V is the type of the required aggregate statistic
V getValue(int lo, int hi) {
        SegmentTreeNode result = getValue(1, 0, N-1, lo, hi);
        return result.getValue();
```

```
}

// nodes[stIndex] is responsible for the segment [left, right]
// and we want to query for the segment [lo, hi]
SegmentTreeNode getValue(int stIndex, int left, int right, int lo, int
        if (left == lo && right == hi)
                return nodes[stIndex];

        int mid = (left + right) / 2;
        if (lo > mid)
                return getValue(2*stIndex+1, mid+1, right, lo, hi);
        if (hi <= mid)
                return getValue(2*stIndex, left, mid, lo, hi);

        SegmentTreeNode leftResult = getValue(2*stIndex, left, mid, lo,
        SegmentTreeNode rightResult = getValue(2*stIndex+1, mid+1, righ
        SegmentTreeNode result;
        result.merge(leftResult, rightResult);
        return result;
}
```

Querying the segment tree hosted with ♥ by **GitHub** **view raw**

This procedure is similar to the one used for building the segment tree, except that we cut off recursion when we reach a desired segment. The complexity of this procedure is O(log N).

Updating the segment tree

The above two procedures, building the segment tree and querying it, are sufficient for the first use case: a static segment tree. It so happens that the second use case:

point updates, doesn't require many changes. In fact, we don't have to change the problem specific logic at all. No changes in the structure SegmentTreeNode are required.

We just need to add in a procedure for updating the segment tree. It is very similar to the buildTree() procedure, the only difference being that it follows only one path down the tree (the one that leads to the leaf node being updated) and comes back up, recursively updating parent nodes along this same path.

```
// We want to update the value associated with index in the input array
void update(int index, T value) {
        update(1, 0, N-1, index, value);
}

// nodes[stIndex] is responsible for segment [lo, hi]
void update(int stIndex, int lo, int hi, int index, T value) {
        if (lo == hi) {
                nodes[stIndex].assignLeaf(value);
                return;
        }

        int left = 2 * stIndex, right = left + 1, mid = (lo + hi) / 2;
        if (index <= mid)
                update(left, lo, mid, index, value);
        else
                update(right, mid+1, hi, index, value);

        nodes[stIndex].merge(nodes[left], nodes[right]);
}
```

Updating the segment tree hosted with ♥ by **GitHub** **view raw**

Clearly, the complexity of this operation is O(log N), assuming that assignLeaf() and merge() work in O(1).

Segment Tree template

Let's put all this together to complete the template for a segment tree.

```
// T is the type of input array elements
// V is the type of required aggregate statistic
template<class T, class V>
class SegmentTree {
        SegmentTreeNode* nodes;
        int N;

public:
        SegmentTree(T arr[], int N) {
                this->N = N;
                nodes = new SegmentTreeNode[getSegmentTreeSize(N)];
                buildTree(arr, 1, 0, N-1);
        }

        ~SegmentTree() {
                delete[] nodes;
        }

        V getValue(int lo, int hi) {
                SegmentTreeNode result = getValue(1, 0, N-1, lo, hi);
                return result.getValue();
        }

        void update(int index, T value) {
                update(1, 0, N-1, index, value);
```

```
26          }
27
28  private:
29          void buildTree(T arr[], int stIndex, int lo, int hi) {
30                  if (lo == hi) {
31                          nodes[stIndex].assignLeaf(arr[lo]);
32                          return;
33                  }
34
35                  int left = 2 * stIndex, right = left + 1, mid = (lo + h
36                  buildTree(arr, left, lo, mid);
37                  buildTree(arr, right, mid + 1, hi);
38                  nodes[stIndex].merge(nodes[left], nodes[right]);
39          }
40
41          SegmentTreeNode getValue(int stIndex, int left, int right, int
42                  if (left == lo && right == hi)
43                          return nodes[stIndex];
44
45                  int mid = (left + right) / 2;
46                  if (lo > mid)
47                          return getValue(2*stIndex+1, mid+1, right, lo,
48                  if (hi <= mid)
49                          return getValue(2*stIndex, left, mid, lo, hi);
50
51                  SegmentTreeNode leftResult = getValue(2*stIndex, left,
52                  SegmentTreeNode rightResult = getValue(2*stIndex+1, mid
53                  SegmentTreeNode result;
54                  result.merge(leftResult, rightResult);
55                  return result;
56          }
57
58          int getSegmentTreeSize(int N) {
59                  int size = 1;
60                  for (; size < N; size <<= 1);
```

```
61                  return size << 1;
62          }
63
64          void update(int stIndex, int lo, int hi, int index, T value) {
65                  if (lo == hi) {
66                          nodes[stIndex].assignLeaf(value);
67                          return;
68                  }
69
70                  int left = 2 * stIndex, right = left + 1, mid = (lo + h
71                  if (index <= mid)
72                          update(left, lo, mid, index, value);
73                  else
74                          update(right, mid+1, hi, index, value);
75
76                  nodes[stIndex].merge(nodes[left], nodes[right]);
77          }
78  };
```

Segment tree template hosted with ♥ by **GitHub** **view raw**

We shall now see how this template can be used to solve different problems, without requiring a change in the tree implementation, and how the structure SegmentTreeNode is implemented differently for different problems.

The **first problem** we'll look at it is GSS1. This problem asks for a solution to maximum subarray problem for each range of an array. My objective here is not to explain how to solve this problem, rather to demonstrate how easily it can be implemented with the above template at hand.

As it turns out, we need to store 4 values in each segment tree node to be able to

merge child nodes to form a solution to their parent's node:

1. Maximum sum of a subarray, starting at the leftmost index of this range
2. Maximum sum of a subarray, ending at the rightmost index of this range
3. Maximum sum of any subarray in this range
4. Sum of all elements in this range

The SegmentTreeNode for this problem looks as follows:

```
struct SegmentTreeNode {
        int prefixMaxSum, suffixMaxSum, maxSum, sum;

        void assignLeaf(int value) {
                prefixMaxSum = suffixMaxSum = maxSum = sum = value;
        }

        void merge(SegmentTreeNode& left, SegmentTreeNode& right) {
                sum = left.sum + right.sum;
                prefixMaxSum = max(left.prefixMaxSum, left.sum + right.
                suffixMaxSum = max(right.suffixMaxSum, right.sum + left
                maxSum = max(prefixMaxSum, max(suffixMaxSum, max(left.m
        }

        int getValue() {
                return maxSum;
        }
};
```

GSS1 segment tree node hosted with ♥ by **GitHub** **view raw**

The complete solution for this problem can be viewed here.

The **second problem** we'll look at is GSS3, which is very similar to GSS1 with the only difference being that it also asks for updates to array elements, while still maintaining the structure for getting maximum subarray sum. Now, we can understand the advantage of separating problem specific logic from the segment tree implementation. This problem requires no changes to the template and even uses the same SegmentTreeNode as used for GSS1. The complete solution for this problem can be viewed here.

The **third problem**: BRCKTS, we'll look at is very different from the first two but the differences are only superficial since we'll be able to solve it using the same structure. This problem gives a string containing parenthesis (open and closed), requires making updates to individual parenthesis (changing an open parenthesis to closed or vice versa), and checking if the whole string represents a correct parenthesization.

As it turns out, we need only 2 things in each segment tree node:

1. The number of unmatched open parenthesis in this range
2. The number of unmatched closed parenthesis in this range

The SegmentTreeNode for this problem looks as follows:

```
struct SegmentTreeNode {
        int unmatchedOpenParans, unmatchedClosedParans;

        void assignLeaf(char paranthesis) {
```

```
5                   if (paranthesis == '(')
6                           unmatchedOpenParans = 1, unmatchedClosedParans
7                   else
8                           unmatchedOpenParans = 0, unmatchedClosedParans
9           }
10
11          void merge(SegmentTreeNode& left, SegmentTreeNode& right) {
12                  int newMatches = min(left.unmatchedOpenParans, right.ur
13                  unmatchedOpenParans = right.unmatchedOpenParans + left.
14                  unmatchedClosedParans = left.unmatchedClosedParans + ri
15          }
16
17          bool getValue() {
18                  return unmatchedOpenParans == 0 && unmatchedClosedParar
19          }
20  };
```

BRCKTS segment tree node hosted with ♥ by **GitHub** **view raw**

The complete solution for this problem can be viewed here.

The **final problem** we'll look at in this post is KGSS. This problem asks for the maximum pair sum in each subarray and also requires updates to individual array elements. As it turns out, we only need to store 2 things in each segment tree node:

1. The maximum value in this range
2. The second maximum value in this range

The SegmentTreeNode for this problem looks as follows:

```
struct SegmentTreeNode {
        int maxNum, secondMaxNum;

        void assignLeaf(int num) {
                maxNum = num;
                secondMaxNum = -1;
        }

        void merge(SegmentTreeNode& left, SegmentTreeNode& right) {
                maxNum = max(left.maxNum, right.maxNum);
                secondMaxNum = min(max(left.maxNum, right.secondMaxNum)
        }

        int getValue() {
                return maxNum + secondMaxNum;
        }
};
```

KGSS segment tree node hosted with ♥ by **GitHub** **view raw**

The complete solution for this problem can be viewed here.

I hope this post presented a gentle introduction to segment trees and I look forward to feedback for possible improvements and suggestions for a future post on segment trees with lazy propagation.

Continue to Part 2...

Related

A simple approach to segment trees, part 2
In "Algorithms"

Range updates with BIT / Fenwick Tree
In "Data Structures"

Kruskal's Minimum Spanning Tree Algorithm
In "Algorithms"

This entry was posted in Algorithms, Data Structures, Spoj and tagged aggregate statistics, BRCKTS Spoj, C++ implementation, GSS1 Spoj, GSS3 Spoj, KGSS Spoj, maximum subarray problem, query, segment tree, template, update. Bookmark the permalink.

← Interesting problem, multiple solutions

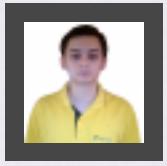
Test oracle: A useful tool for competitive programmers →

10 thoughts on "A simple approach to segment trees"

Rajat says: Reply

December 20, 2014 at 5:05 am

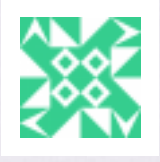
Good work! Keep it up. What I think is that you should use a bit simple language and demonstrate at least one problem solving completely.Rest is great.



kartik kukreja says: Reply

December 20, 2014 at 4:01 pm

Solutions to 4 problems are described in the post and complete code for them are provided.



Sudharsansai says: Reply

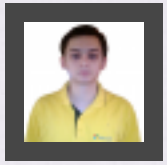
December 20, 2014 at 3:34 pm

Really a nice post...A helping one for a beginner like me

Rajat says: Reply

December 22, 2014 at 6:44 am

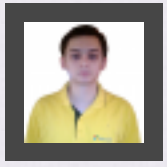
Sir,when will there be post for lazy propagation.



kartik kukreja says: Reply

December 22, 2014 at 10:07 am

I'm a little busy these days but I'll work on that post. It should be out some time in the near future.



kartik kukreja says: Reply

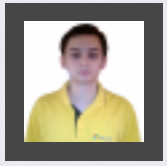
January 10, 2015 at 5:37 pm

The second part of this post, explaining lazy propagation, is out: https://kartikkukreja.wordpress.com/2015/01/10/a-simple-approach-to-segment-trees-part-2/

Anonymous says: Reply

December 30, 2014 at 6:56 pm

Kartik, your blog posts are very well-written and helpful. Blessings and keep them coming!!



kartik kukreja says: Reply

December 30, 2014 at 8:55 pm

Thank you

sethuiyer says: Reply

January 6, 2015 at 12:14 pm

Thanks a lot for providing such a nice explanation and for an awesome template 🙂

Pingback: A simple approach to segment trees, part 2 | Everything Under The Sun

Leave a Reply

Enter your comment here...