How is the memory of the array of segment tree 2 * 2 ^(ceil(log(n))) - 1?

The link: http://www.geeksforgeeks.org/segment-tree-set-1-sum-of-given-range/. This is the quoted text:

> We start with a segment arr[0 . . . n-1]. and every time we divide the current segment into two halves(if it has not yet become a segment of length 1), and then call the same procedure on both halves, and for each such segment we store the sum in corresponding node. All levels of the constructed segment tree will be completely filled except the last level. Also, the tree will be a Full Binary Tree because we always divide segments in two halves at every level. Since the constructed tree is always full binary tree with n leaves, there will be n-1 internal nodes. So total number of nodes will be 2n – 1. Height of the segment tree will be ceil[log(n)]. Since the tree is represented using array and relation between parent and child indexes must be maintained, size of memory allocated for segment tree will be 2*2 ^(ceil(log(n))) - 1.

How is the memory allocated(last line of the above para) that much? How are the parent and child indexes stored in the code if it is correct? Please give reasoning behind this. If this is false then what is the correct value?

arrays memory data-structures tree segment-tree

asked Feb 12 '15 at 6:21

dauntless
43 1 5

3 Answers

What is happening here is, if you have an array of n elements, then the segment tree will have a leaf node for each of these n entries. Thus, we have (n) leaf nodes, and also (n-1) internal nodes.

Total number of nodes= n + (n-1) = 2n-1 Now, we know its a full binary tree and thus the height is: ceil(Log2(n)) +1

Total no. of nodes = 2^0 + 2^1 + 2^2 + … + 2^ceil(Log2(n)) // which is a geometric progression where 2^i denotes, the number of nodes at level i.

Formula of summation G.P. = a * (r^size - 1)/(r-1) where a=2^0

Total no. of nodes = 1*(2^(ceil(Log2(n))+1) -1)/(2-1)

= 2* [2^ceil(Log2(n))] -1 (you need space in the array for each of the internal as well as leaf nodes which are this count in number), thus it is the array of size.

= O(4 * n) approx..

You may also think this way, is the segment tree is this:

```
     10
    /  \
   3    7
  /\    /\
 1  2  3  4
```

If the above is you segment tree, then you array of segment tree will be: 10,3,7,1,2,3,4 i.e. 0th element will store the sum of 1st and 2nd entries, 1st entry will store the sum of 3 and 4th and

2nd will store the sum of 5th and 6th entry!!

Also, the better explanation is: if the array size **n** is a power of 2, then we have exactly **n-1** internal nodes, summing up to **2n-1** total nodes. But not always, we have **n** as the power of 2, so we basically need the smallest power of **n** which is greater then **n**. That means this,

```
int s=1;
for(; s<n; s<<=1);
```

You may see my same answer here

edited Jan 26 at 21:51

yuxiang
32 5

answered Feb 13 '15 at 14:53

user3004790
300 2 7

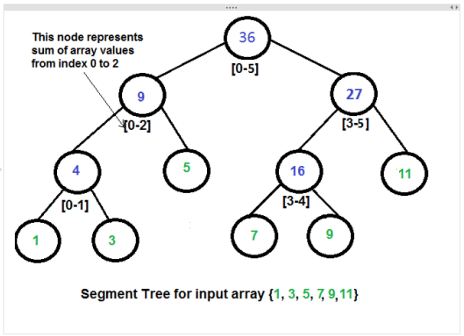
Isn't there a contradiction? As in first it has been mentioned that the total no. of nodes is 2*(n-1).Next it is said that the total no. of nodes is 2* 2 ceil(Log2(n)) -1(I understood the explanation behind that). But aren't we allocating extra memory by considering 2* 2 ceil(Log2(n)) -1 when n is not a power of 2. Is it right to assume that a segment tree requires 2* n-1 array size for all n since 2* n-1 is the total no. of nodes for a full binary tree and that is what is required in a segment tree. Is that right? If not what is the purpose behind allocating 2* 2 ceil(Log2(n)) -1. – dauntless Feb 15 '15 at 5:03

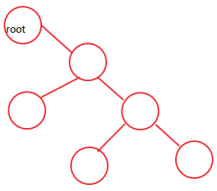
in the last line i think you have a typo. it should be "so we basically need the smallest power of 2 (not n) which is greater the n" – shshnk Aug 17 '16 at 14:17

Oddly enough, I was reading from the same source as the question when I came upon this. I'll try and answer my best.

Let's start with a basic difference in trees representations (in **context** only):

1. The almost "Worst Case" scenario. This one is **not completely balanced** and not really fun to traverse. Why? Because, with different inputs, different trees might be generated and hence time taken to traverse is not very predictable.

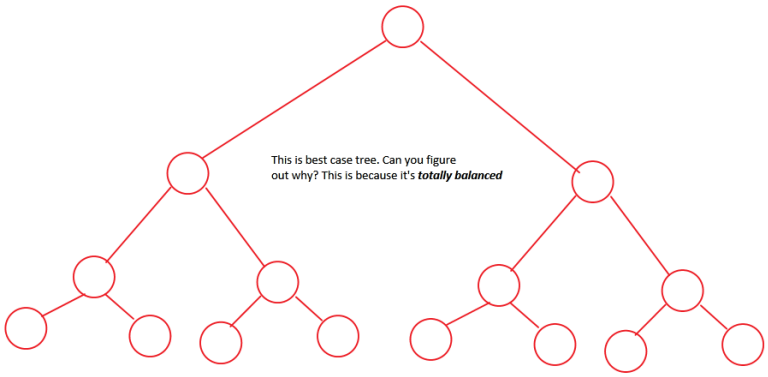




A *worst case* tree in terms of traversal.

This is "almost" a *worst case* tree.

2. Our "Best Case" scenario. This one is **totally balanced or complete** and will take a predictable amount of time to traverse, always. Moreover, this tree is also better *"hacked"*.



Now let's get back to our question. [Refer to the first image] We know that for every **n-input** array (The numbers in *green*), there will be **n-1 internal nodes** (The numbers in *blue*). So a maximum of **2n-1** node space must be allocated.

But the code here does something on the contrary. Why and how?

1. **What you expect:** You expect that the memory allocated for **2n-1** nodes should be sufficient. In other words, this should be done:

   ```
   int *st = new int[2*n - 1];
   ```

   Assuming the rest of the code works well, this is isn't a very good idea. That's because it creates our unbalanced tree, much like in our first case. Such a tree is not easy to traverse nor easy to apply to problem-solving.

2. **What really happens:** We add/pad extra memory with `null` or `0` values. We do this:

   ```
   int x = (int)(ceil(log2(n))); //Height of segment tree
   int max_size = 2*(int)pow(2, x) - 1; //Maximum size of segment tree
   int *st = new int[max_size];
   ```

   That is we allocate enough space to generate a balanced complete tree. Such a tree is easy to traverse (using some special modifications) and can be applied to problems directly.

How did we allocate enough memory for **case 2**? Here's how:

- We know there are at least three components in our balanced Segment Tree:
  1. **n** numbers from our input array.
  2. **n-1** internal nodes which are mandatorily required.
  3. The extra space we need to allocate for our **padding**.
- We also know that a balanced tree with **k** leaves will have:

  $$Number\ of\ leaves\ in\ target\ tree = k$$

  $$Height = \lceil log_2(k) \rceil$$

  $$Nodes = 2 * 2^{\lceil log_2(k) \rceil} - 1$$

- Combining the two we get the desired outcome:

  ```
  int x = (int)(ceil(log2(n))); //Height of segment tree
  int max_size = 2*(int)pow(2, x) - 1; //Maximum size of segment tree
  int *st = new int[max_size];
  ```

Trivia! Raising 2 to the power of `x` above, ensures that we get the nearest ceiling integer which is:

1. Greater than or equal to `n` (Number of elements in our input array).
2. Is perfectly and repeatedly divisible by 2, to get a **completely balanced** *2-ary (binary)* tree.

answered Feb 21 '15 at 18:16

Quirk
712 7 20

check out leetcode.com/problems/range-sum-query-mutable/solution/… and search for `1. Build segment tree.` it doesn't use padding. It only uses 2*n array to represent the binary tree. (it somehow always transformed it into a complete tree) And it seems correct. But it's not explained anywhere. – Weishi Zeng Sep 21 at 20:49

Let the size of input array is n.
All the input array elements will be leaf nodes in segment tree so the **number of leaf nodes = n**
Since the Segment tree is a complete tree so the Hight of Segment Tree **h = ⌈ $Log_2 n$ ⌉ + 1**
Maximum number of nodes in a binary tree of height 'h' is $2^h - 1$

So Number of nodes in a segment tree = $2^{⌈ Log_2 n ⌉ + 1} - 1$
Equals to $2 * 2^{⌈ Log_2 n ⌉} - 1$

answered Aug 9 at 6:22

vijay yadav
41 3