Data Structures

Oliver-Matis Lill

June 6, 2017

Session Structure

- First I will hold a short lecture on the subject
- Rest of the time is spent on problem solving
- Try to finish the standard problems at home if necessary

Topics

- Session subjects
  1. Fenwick Tree
  2. Segment Tree
- Advanced subjects for self-learning:
  1. Lazy Propagation on Segment Tree
  2. Persistent Segment Tree

Problem nr. 1

- We have an array of up to 10^5 elements:

4	3	6	7	5	2	1	9	7	8

- We have to perform the following operations on the array:
  1. Increase the value of some element by x
  2. Give some element a new value y
  3. Output the sum of elements in segment $[l, r]$
- Up to 10^5 operations

Fenwick Tree

Idea

- Let's look at the binary form of indices
- Let's create a new array f, where $f_i = a_{i-s+1} + \ldots + a_i$ and s is the value of the rightmost 1-bit of i
- For example, if $i = 12_{10} = 1100_2$, then $s = 100_2 = 4_{10}$ and $f_{12} = a_9 + a_{10} + a_{11} + a_{12}$
- Rightmost 1-bit of i can be acquired with the expression $i\& - i$

Fenwick Tree

Example

f[i]	-	4	7	6	20	5	7	1	37	7	15
a[i]	-	4	3	6	7	5	2	1	9	7	8
i	0	1	2	3	4	5	6	7	8	9	10
i&-i	0	1	2	1	4	1	2	1	8	1	2

Fenwick Tree Operations

- Fenwick tree allows you to output the sum of segment $[1, i]$ in $O(\log n)$ time
- To compute the sum, add f_i to the sum and subtract from i its rightmost bit. Repeat until $i = 0$
- For example if $i = 26_{10} = 11010_2$, then the traversed indices are $11010_2, 11000_2, 10000_2$ and the sum is $f_{26} + f_{24} + f_{16}$

- Additionally you can increase the value of element i by x in $O(\log n)$ time
- To do that, increase f_i by x and add to i its rightmost bit. Repeat until i is larger than the size of the array
- For example, if $i = 45_{10} = 101101_2$ then the traversed indices are $101101_2, 101110_2, 110000_2, 1000000_2$ and the updated fenwick tree elements are $f_{45}, f_{46}, f_{48}, f_{64}$

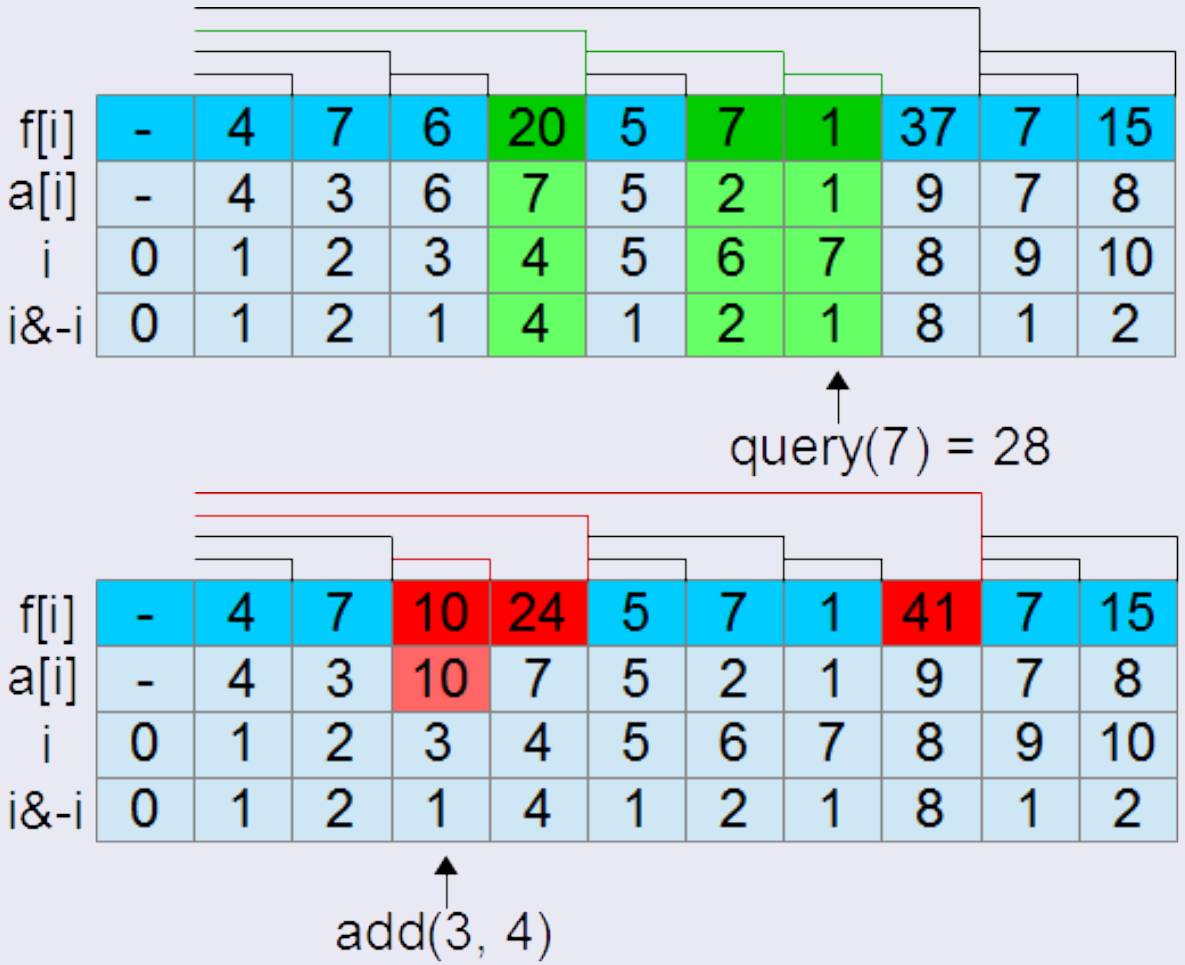
Fenwick Tree Code

Example

```
int get(int i) {
    int sum = 0;
    while(i > 0) {
        sum += f[i];
        i -= i&-i;
    }
    return sum;
}
void add(int i, int x) {
    while(i < f.size() ) {
        f[i] += x;
        i += i&-i;
    }
}
```

Fenwick Tree Operations

Example



Problem nr. 2

- We have up to 10^5 array elements and operations
- The operations are:
  1. Increase the value of some element by x
  2. Assign a new value y to some element
  3. Output the **maximum** value of elements in segment $[l, r]$
- Would Fenwick tree work here?

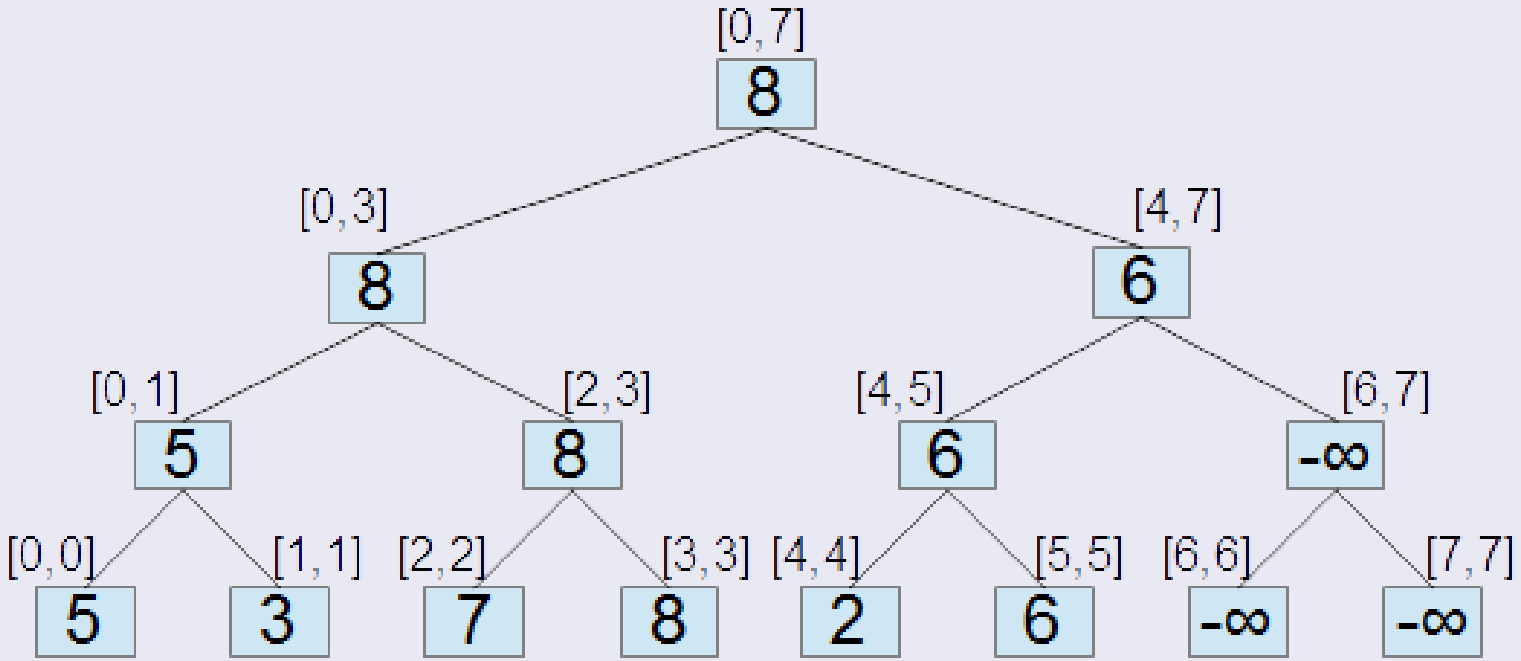
Segment Tree

Idea

- Use a perfect binary tree. Store array elements in the leaves
- Non-leaf nodes hold the maximum/sum/etc of their children
- As a result the value of some node is the maximum of the leaves of its subtree, which corresponds to some initial array segment

Segment Tree

Example



Segment Tree Data

- We could store the segment tree nodes in the following fashion:

```
struct Node {
    Node* leftChild, rightChild; //Children of this node
    int l, r; //Segment covered by this node
    int value; //The max of the covered segment
};
```

- For non-leaf nodes the following equations must always hold

$$\begin{aligned} l &= \text{leftChild->l} \\ r &= \text{rightChild->r} \\ \text{value} &= \max(\text{leftChild->value}, \text{rightChild->value}) \end{aligned}$$

Segment Tree Set Operation

- Notice how the nodes on each level cover disjoint segments, for example for the previously shown segment tree the segments in levels are:

  level 0 [0:7]
  level 1 [0:3], [4:7]
  level 2 [0:1], [2:3], [4:5], [6:7]
  level 3 [0:0], [1:1], [2:2], [3:3], [4:4], [5:5], [6:6], [7:7]

- When we update some array element, only a single node on each level is affected
- Since there are $O(\log n)$ levels we have to update at most $O(\log n)$ nodes

Segment Tree Set Operation

- This gives us a simple algorithm to perform the set operation. It works in the following way:
  1. Recursively traverse from root to the leaf to be updated
  2. Perform the set operation on the leaf
  3. Backtrack up through the recursion, updating the nodes on the path

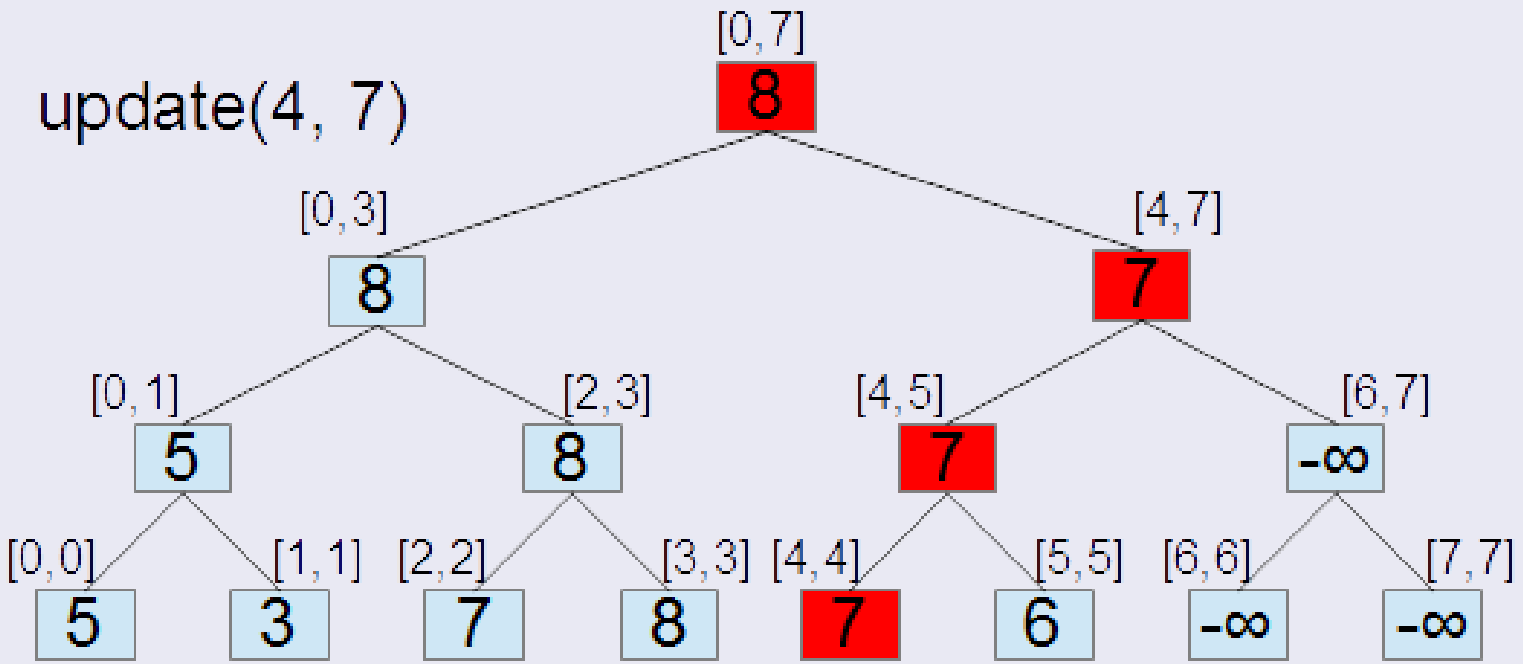
Segment Tree Set Operation Code

Example

```
void set(Node* cur, int i, int y) {
    //If we are at leaf, update it and finish
    if(cur->isLeaf() ) {
        cur->value = y;
        return;
    }
    //Recursively traverse towards the leaf
    if(i <= cur->leftChild->r)
        add(cur->leftChild, i, y);
    else
        add(cur->rightChild, i, y);
    //Finally update nodes as we backtrack
    cur->value = max(cur->leftChild->value, cur->rightChild->value);
}
```

Segment Tree Set Operation Example

Example



Segment Tree Get Operation

- The get operation can also be performed with a recursive algorithm
- The main difference, and complexity is that we have to operate on a range rather than a single element

Segment Tree Get Operation Code

Example

```
int get(Node* cur, int l, int r) {
    //If node is completely inside the queried segment, return
    if(l <= cur->l && cur->r <= r)
        return cur->value;

    int result = -INFINITY;
    //Recurse to children that intersect with the query segment
    if(l <= cur->leftChild->r)
        result = max(result, get(cur->leftChild, l, r) );
    if(cur->rightChild->l <= r)
        result = max(result, get(cur->rightChild, l, r) );

    return result;
}
```

Segment Tree Get Operation Time Complexity

Proof of Complexity

- Analyzing the time complexity is more complicated than with set operation
- Look at the first node where it recurses to both children, let's call that node "split"
- Note that for its children the following equations must hold:

$$\begin{aligned} \text{split->leftChild->r} &\leq r \\ l &\leq \text{split->rightChild->l} \end{aligned}$$

Segment Tree Get Operation Time Complexity

Proof of Complexity

- When we recurse to some node "cur" in the subtree of "split->leftChild", then we have two possiblilities:
  1. $$l \leq \text{cur->leftChild->r}$$

     Here "cur->rightChild" is completely inside $[l, r]$ and therefore returns immediately
  2. $$\text{cur->leftChild->r} < l$$

     Here "cur->leftChild" is completely outside $[l, r]$ and we won't recurse into it
- Note how in both cases, there is at most one node that we recurse into that doesn't return immediately

Proof of Complexity

- When we recurse to some node "cur" in the subtree of "split->rightChild", then we similarly have two possibilities:
  1. $$\text{cur->rightChild->l} \leq r$$

     Here "cur->leftChild" is completely inside [l, r] and returns immediately
  2. $$r < \text{cur->rightChild->l}$$

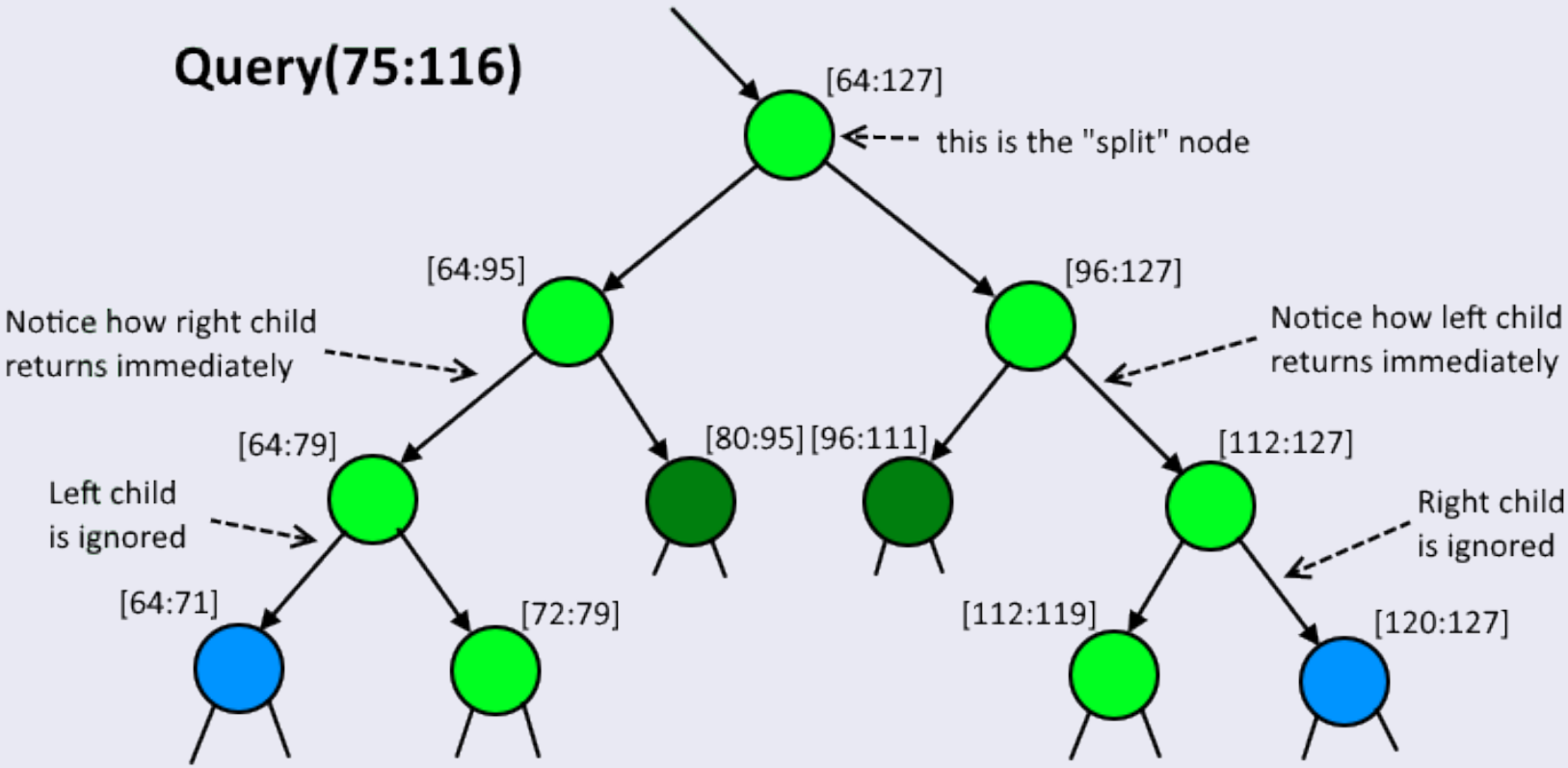
     Here "cur->rightChild" is completely outside [l, r] and won't be recursed into
- Once again, in both cases there is at most one child we can recurse into that won't return immediately

Segment Tree Get Operation Time Complexity

Proof of Complexity

- At each level up to the "split" node we go through only one node
- At each level after the "split" node we traverse at most two nodes that don't return immediately
  1. One in the subtree of "split->leftChild"
  2. One in the subtree of "split->rightChild"
- Since there are $O(\log n)$ levels, the time complexity will be $O(\log n)$

Proof of Complexity Example



Segment Tree Get Operation

Example