Data Structures, Advanced Concepts

Oliver-Matis Lill

March 21, 2017

Problem nr. 1

- You have up to 10^5 array elements and operations
- Operations are the following:
  1. Increase the value of elements in segment $[l, r]$ by d
  2. Give every element in segment $[l, r]$ a new value x
  3. Output the sum of elements in segment $[l, r]$
- How to facilitate these operations?

Lazy Propagation

Idea

- If the range of a node is completely inside the range of the operation, then it's easy to update its value
- Let's update only the topmost covered nodes
- Let's mark the children of those nodes for updating. Let's perform the actual update on them when they are traversed later by some other call
- To facilitate that, we need two functions:
  1. Marking function: marks for each operation, what nodes need to be updated
  2. Propagation function: applies the operation marked on a given node

Lazy Propagation

- Marking function is activated on the root and it marks what nodes this operation needs to be applied to
- Function consists of the following steps:
  1. If the node is completely inside the segment of the operation, then mark this node for update, perform propagation on it and finish
  2. If left child intersects with the segment of the operation, recurse into it
  3. If right child intersects with the segment of the operation, recurse into it
  4. Update the value of the current node based on the new values of its children
- Algorithm behaves similiarly to the querying function

Lazy Propagation Code

Example

```
void mark(Node* cur, int l, int r, Marking mark) {
    if(cur->isInside(l, r) ) {
        cur->addMark(mark);
        cur->propagate();
        return;
    }

    if(cur->leftChild->intersects(l, r) )
        mark(cur->leftChild, l, r, mark);

    if(cur->rightChild->intersects(l, r) )
        mark(cur->rightChild, l, r, mark);

    cur->value = cur->leftChild->getValue() +
                 cur->rightChild->getValue();
}
```

Lazy Propagation

- Propagation function is activated always when a node is traversed or its value is queried
- Its algorithm is the following:
  1. If a set operation is marked on this node, remove any markings from its children and mark them for the same set operation
  2. Apply the set operation and remove its marking
  3. If an addition operation is marked on this node, send this marking to its children. This step doesn't remove their previous markings
  4. Apply the addition operation and remove its marking
- Time complexity is obviously $O(1)$
- This function makes sure that the nodes have a correct value only when we need it, preventing a lot of extra work

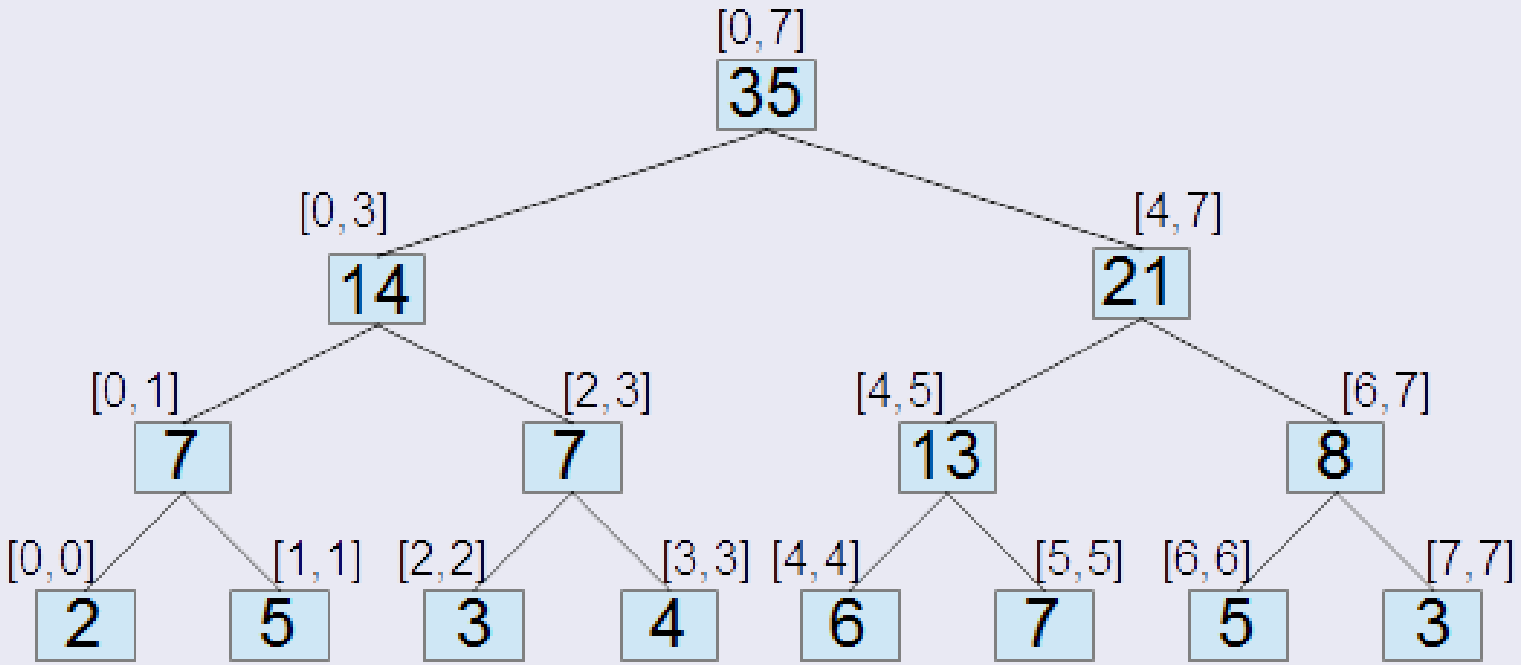
Lazy Propagation Code

Example

```
void Node::propagate() {
    if(this->hasSetMark() ) {
        this->value = this->segmentLength * this->setMark();
        this->leftChild->addMark(this->setMark);
        this->rightChild->addMark(this->setMark);
        this->removeSetMark();
    }
    if(this->hasAdditionMark() ) {
        this->value += this->segmentLength * this->additionMark();
        this->leftChild->addMark(this->additionMark);
        this->rightChild->addMark(this->additionMark);
        this->removeAdditionMark();
    }
}
```

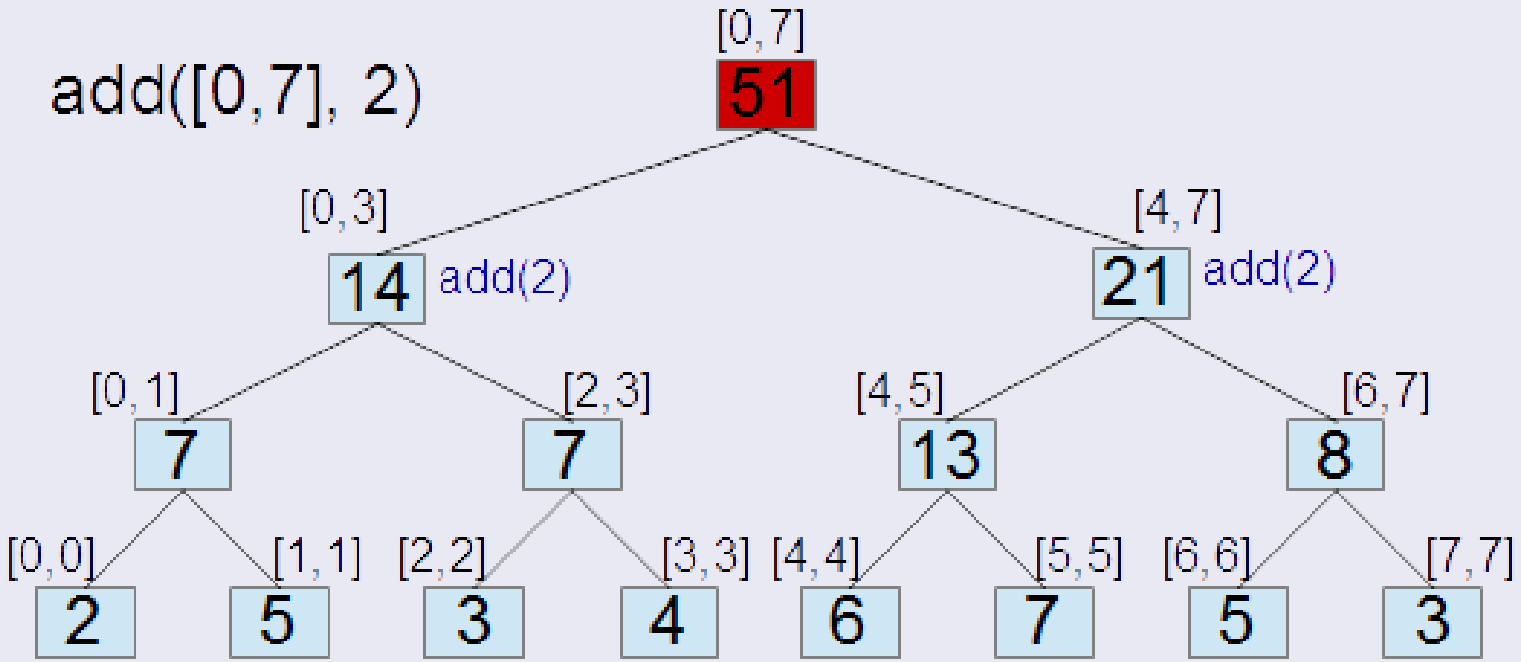
Lazy Propagation

Example



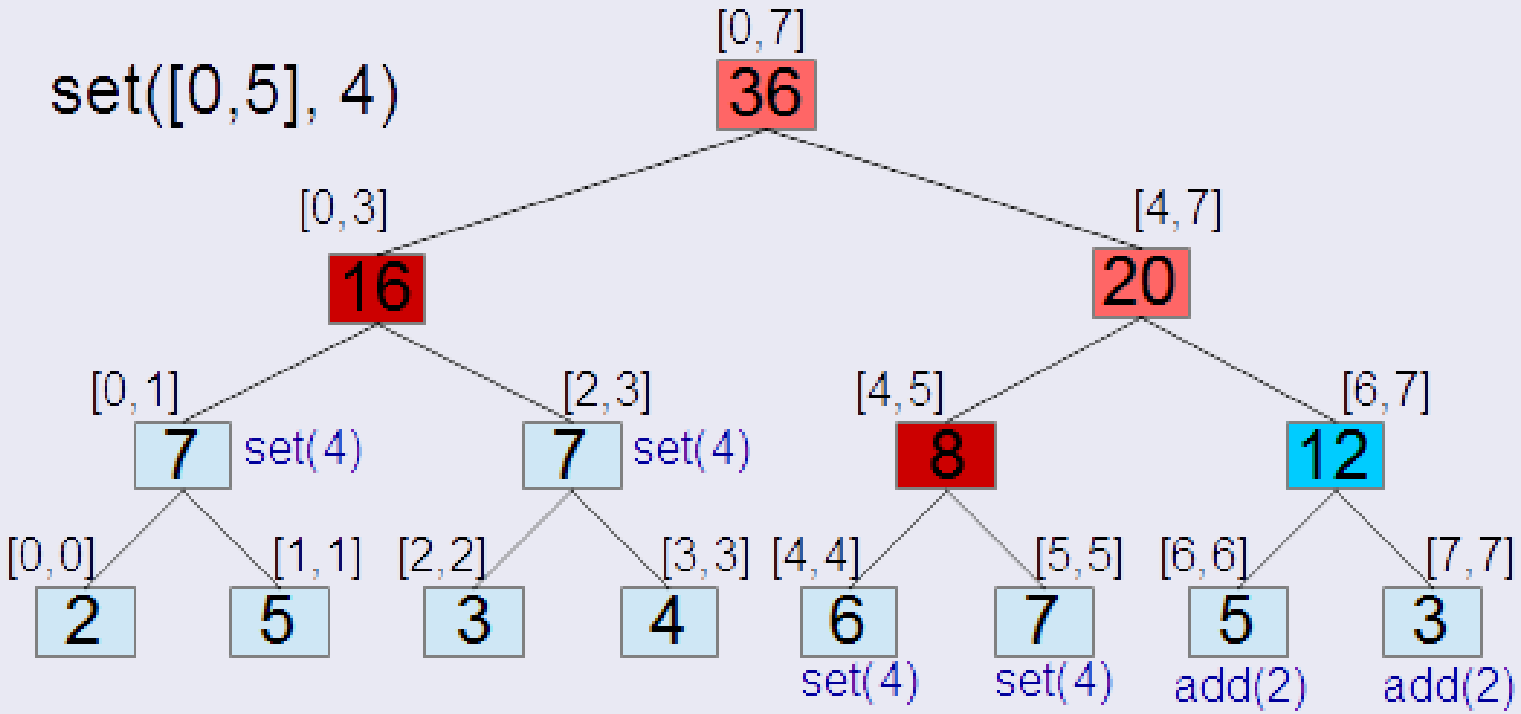
Lazy Propagation

Example

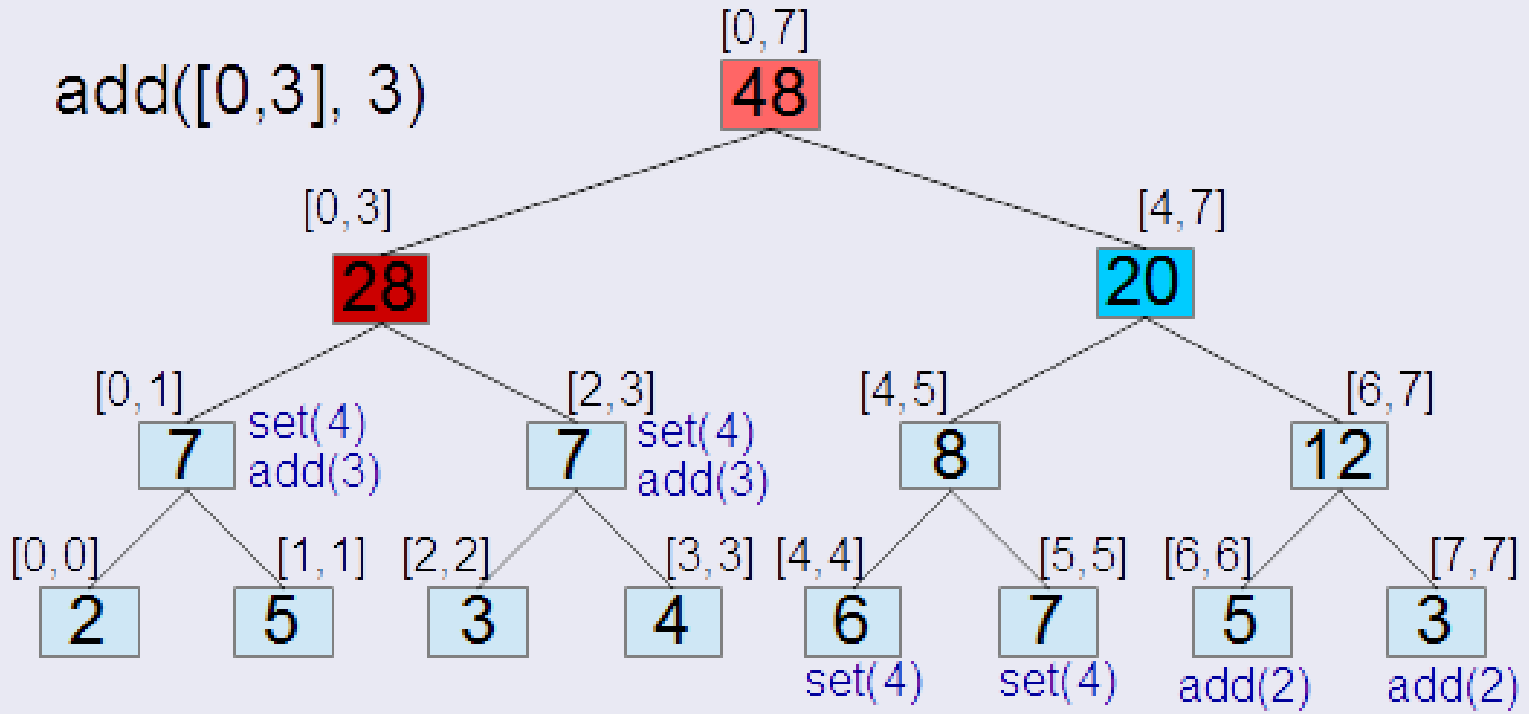


Lazy Propagation

Example

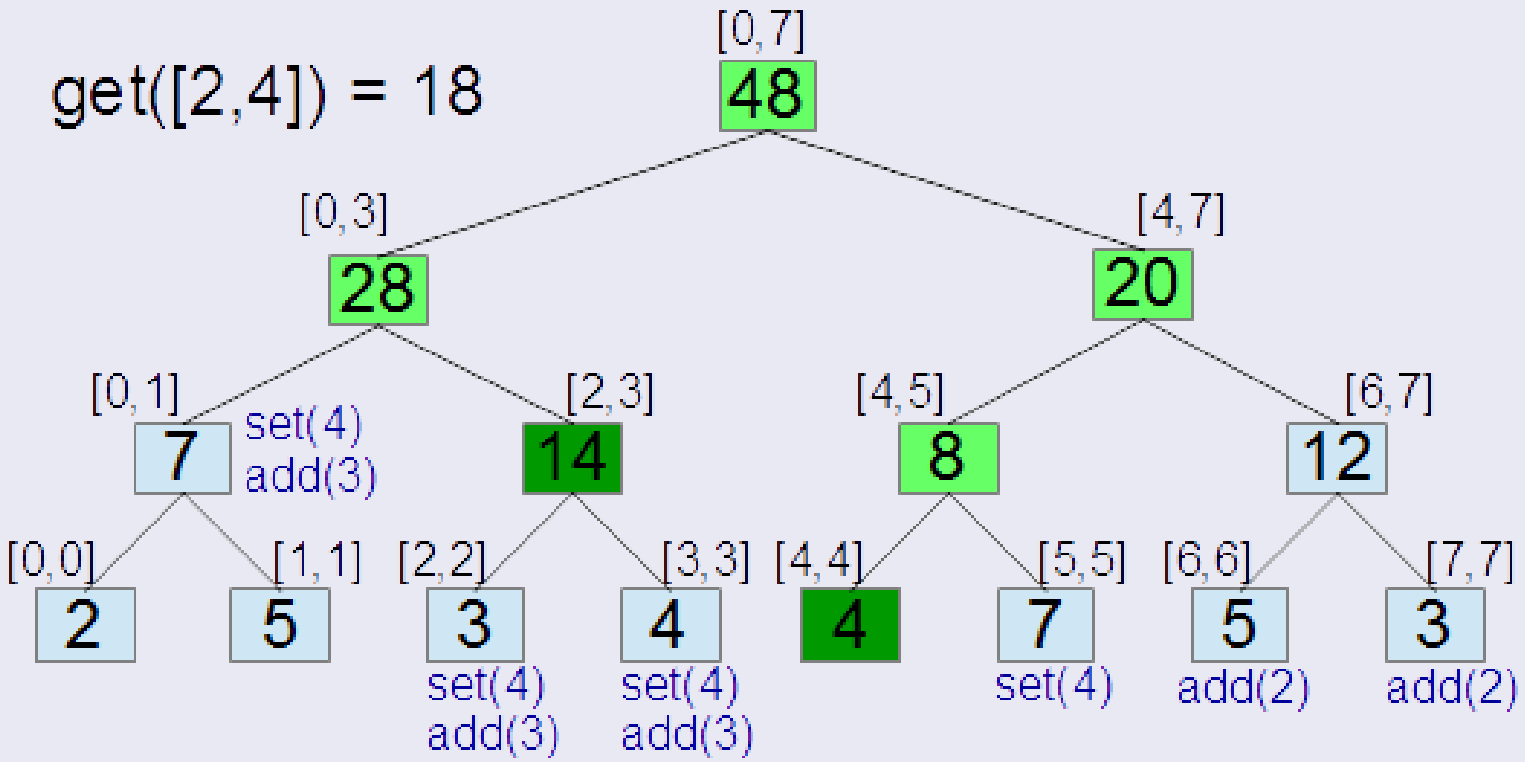


Example


add([0,3], 3)
[0,7]
48
[0,3]
28
[4,7]
20
[0,1]
7
set(4)
add(3)
[2,3]
7
set(4)
add(3)
[4,5]
8
[6,7]
12
[0,0]
2
[1,1]
5
[2,2]
3
[3,3]
4
[4,4]
6
set(4)
[5,5]
7
set(4)
[6,6]
5
add(2)
[7,7]
3
add(2)

Lazy Propagation

Example



Lazy Propagation Caveats

- Make sure that the propagations have been correctly performed, it's easy to make a mistake there
- Propagation needs to be performed in the traversal of the query function as well
- Propagation should be performed when asking for node values. That can be facilitated with this function:

```
int Node::getValue() {
    this->propagate();
    return this->value;
}
```

Problem nr. 2

- We have some initial array with $N \leq 10^5$ elements
- At each of the $T \leq 10^5$ timepoints $1, \ldots, T$, some element i gets updated to a new value x
- We have $Q \leq 10^5$ queries each of which asks for the sum of array elements at some segment $[l, r]$ in some past timepoint t
- Queries are online and asked after the updates

Persistent Segment Tree

Idea

- The queries would be easy to answer if we had a snapshot of the segment tree at each timepoint
- When updating some element, at most $O(\log n)$ nodes in the segment tree get changed: the nodes along the path from root to the updated leaf
- For each timepoint, instead of creating a copy of the entire segment tree, copy only nodes on the path to be updated and update them
- The nodes on the copied path can have children from previous timepoints

Persistent Segment Tree

- Addition will be similiar to the basic segment tree one. The main difference is that it needs to create new nodes on the updated path
- Query operation will remain the same, it just needs to be called from the correct root

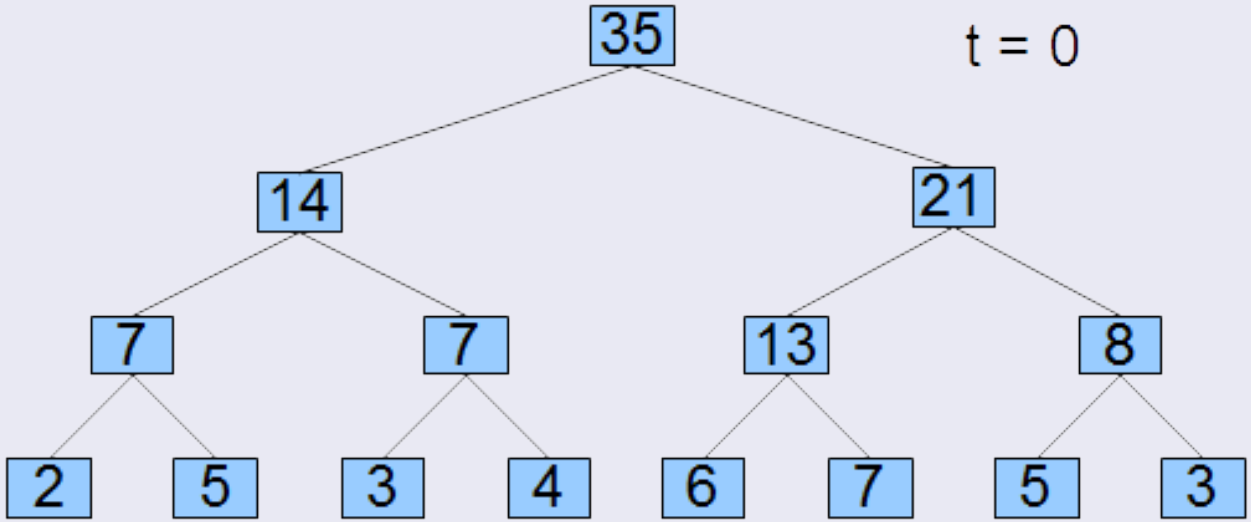
Persistent Segment Tree Code

Example

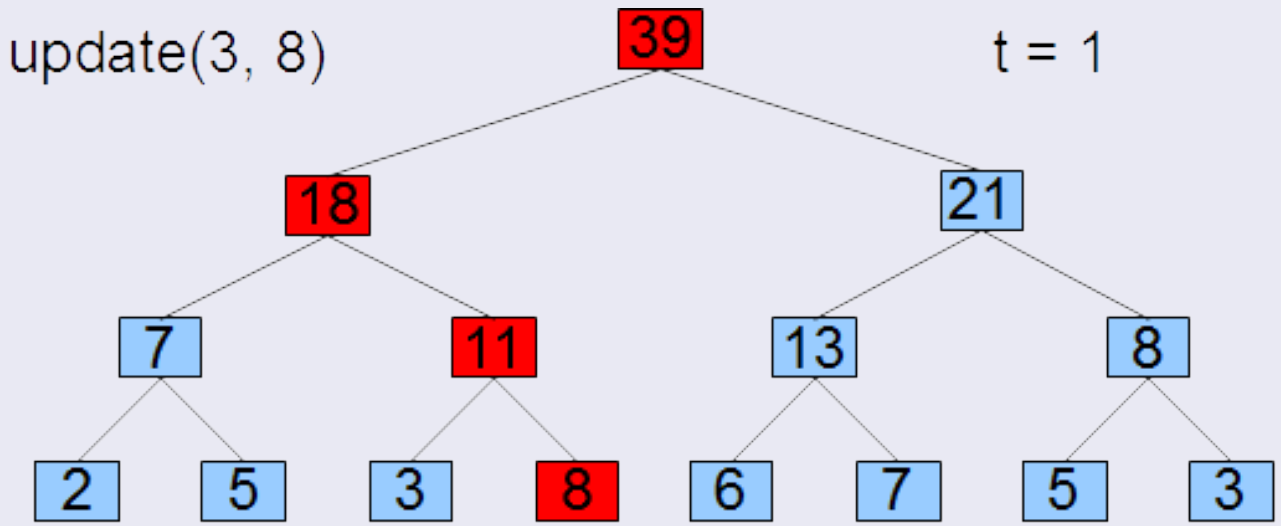
```
Node* add(Node* cur, int i, int x) {
    Node* next = new Node;
    *next = *cur;
    if(next->isLeaf() ) {
        next->value += x;
        return next;
    }
    if(next->leftChild->contains(i) )
        next->leftChild = add(next->leftChild, i, x);
    if(next->rightChild->contains(i) )
        next->rightChild = add(next->rightChild, i, x);
    return next;
}
```

Persistent Segment Tree

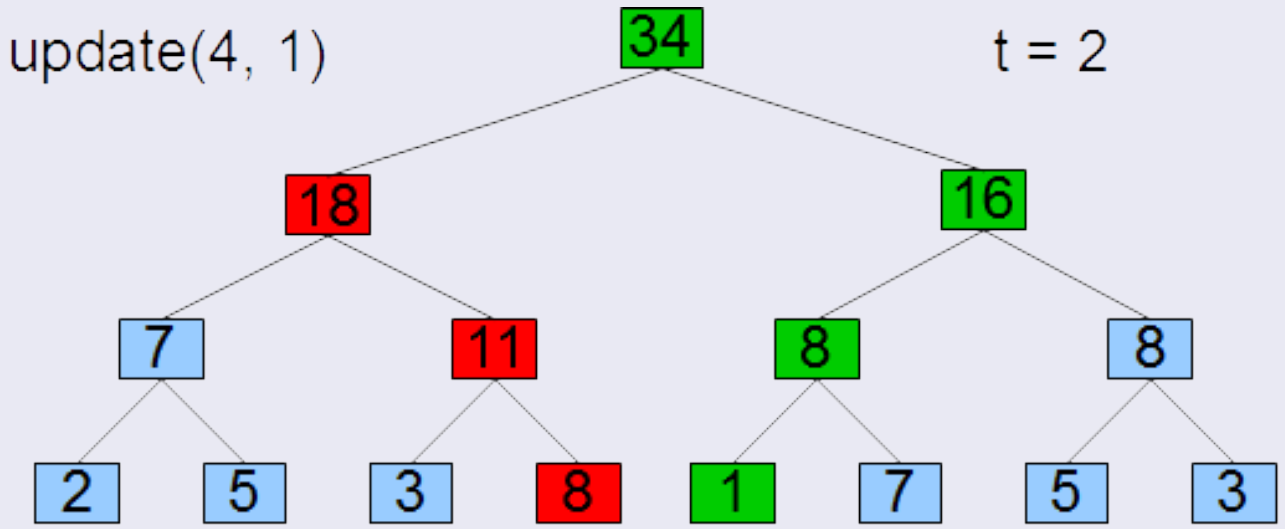
Example



Example

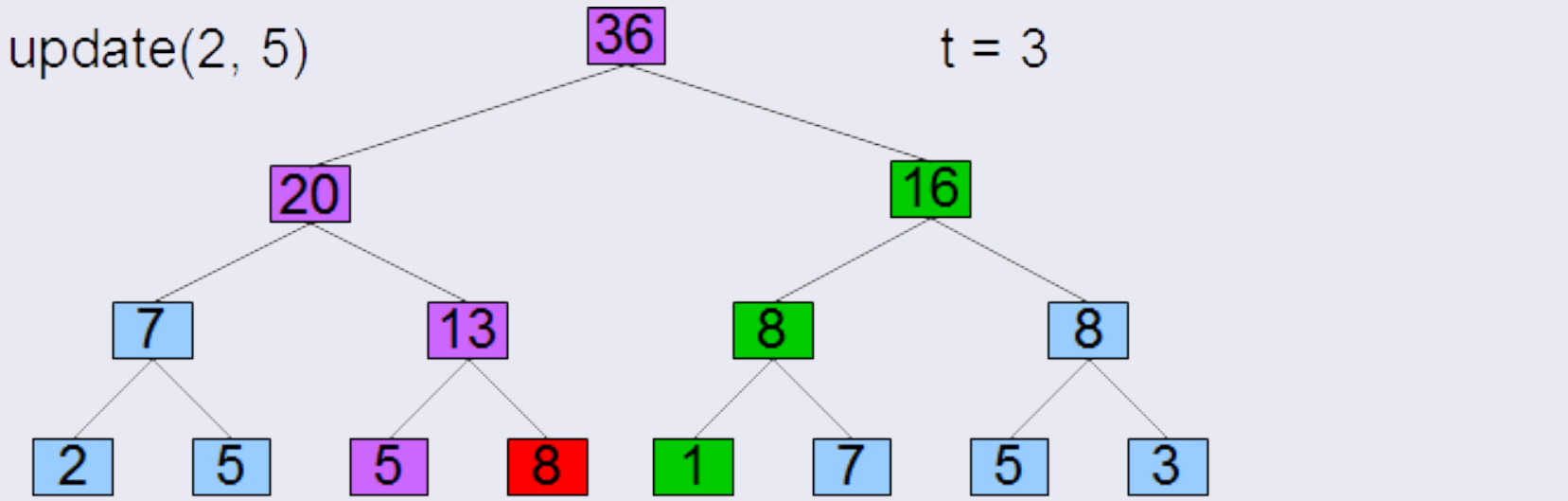


Example



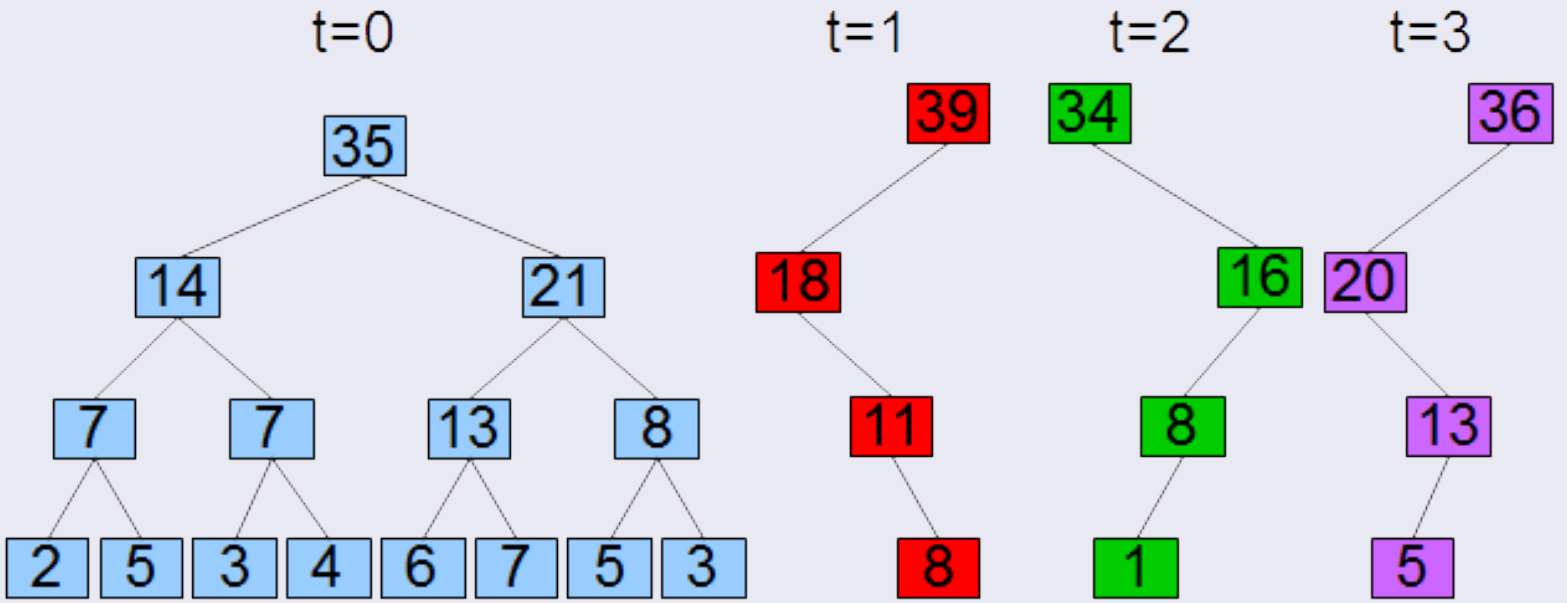
Persistent Segment Tree

Example



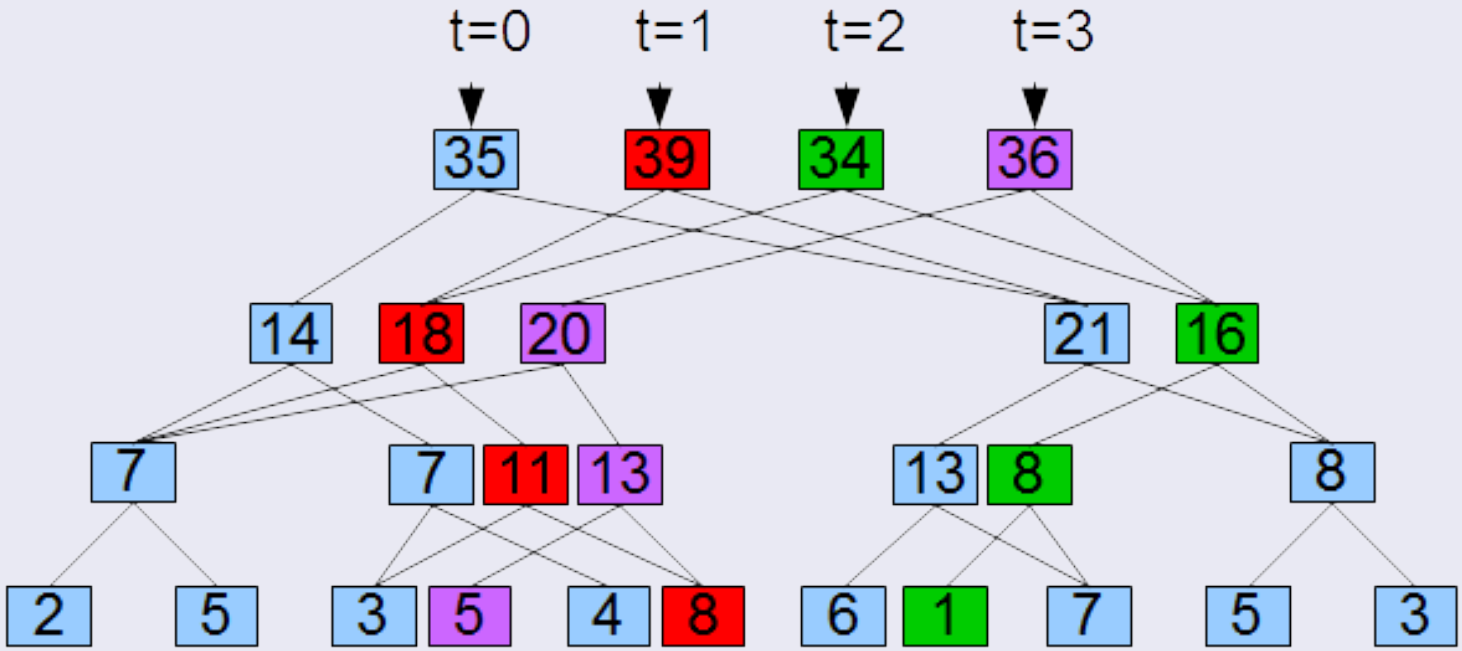
Example

The data stored for each timepoint looks like the following:



Example

And the entire persistent tree with all parent-child relations is the following:



Persistent Segment Tree

- The total storage complexity is $O(N + T \log N)$
- The time complexity for answering each query is $O(\log N)$
- Some problems:
  1. `https://www.codechef.com/problems/SEGSUMQ`
  2. `https://www.hackerrank.com/contests/w22/challenges/sequential-prefix-function`