Segment Trees

Timon Gehr

February 14, 2017

Swiss Olympiad in Informatics

Arrays

List of n elements $[x_0, \ldots, x_{n-1}]$ with operations.

Updates

$$[x_0, \ldots, x_i, \ldots, x_{n-1}] \mapsto [x_0, \ldots, \underline{x_i'}, \ldots, x_{n-1}]$$

$$[1, 3, 4, 2, 6, \underbrace{0}_{i=5}, 6, 1, 0, 4] \mapsto [1, 3, 4, 2, 6, \underline{5}, 6, 1, 0, 4]$$

Queries

$$[x_0, \ldots, x_i, \ldots, x_{n-1}] \mapsto x_i$$

$$[1, 3, \underbrace{4}_{i=2}, 2, 6, 5, 6, 1, 0, 4] \mapsto 4$$

Running time: $\mathcal{O}(1)$.

We want to support more general operations.

Operations on Segments

Updates

$$[x_0, \ldots, \underbrace{x_l, \ldots, x_r}_{x \mapsto f(x)}, \ldots, x_{n-1}] \mapsto [x_0, \ldots, \underline{f(x_l), \ldots, f(x_r)}, \ldots, x_{n-1}]$$

$$[1, 3, \underbrace{4, 2, 6, 5}_{x \mapsto x+2}, 6, 1, 0, 4] \mapsto [1, 3, \underline{6, 4, 8, 7}, 6, 1, 0, 4]$$

$$[1, 3, 6, 4, \underbrace{8, 7, 6, 1}_{x \mapsto 3}, 0, 4] \mapsto [1, 3, 6, 4, \underline{3, 3, 3, 3}, 0, 4]$$

Queries

$$[x_0, \ldots, \underbrace{x_l, \ldots, x_r}_{\circ}, \ldots, x_{n-1}] \mapsto x_l \circ \cdots \circ x_r$$

$$[1, 3, \underbrace{6, 4, 3, 3, 3, 3, 0, 4}_{+}] \mapsto 6 + 4 + 3 + 3 + 3 + 0 + 4 = 23$$

$$[1, \underbrace{3, 6, 4, 3, 3}_{\max}, 3, 3, 0, 4] \mapsto \max(\{3, 6, 4, 3, 3\}) = 6$$

Associativity

Queries

$$[x_0, \dots, \underbrace{x_l, \dots, x_r}_{\circ}, \dots, x_{n-1}] \mapsto x_l \circ \dots \circ x_r$$

Why is $x_l \circ x_{l+1} \circ x_{l+2} \circ \dots \circ x_{r-1} \circ x_r$ well-defined?
$\Rightarrow$ We assume that $(x \circ y) \circ z = x \circ y \circ z = x \circ (y \circ z)$.

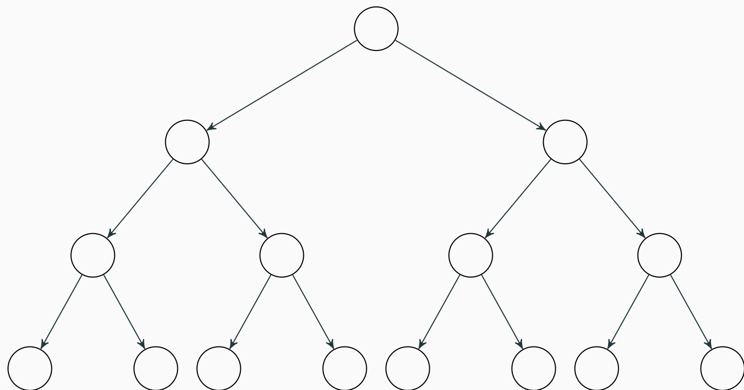
Examples

$+, \cdot, \min, \max, \gcd, \operatorname{lcm}$, AND, OR, XOR, string concatenation, matrix multiplication, function composition

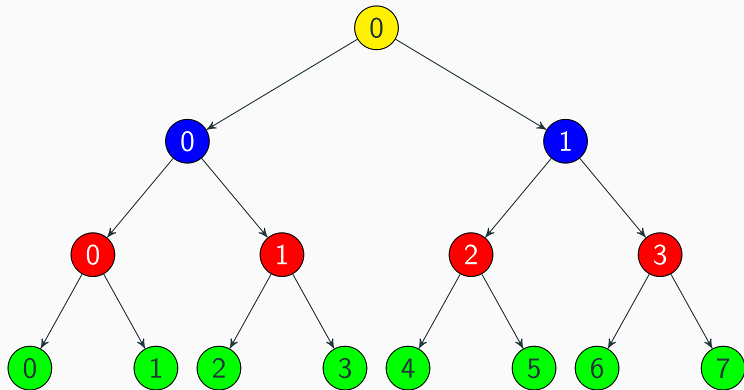
Non-examples

pow, $-$, $/$, average, floating-point addition, NAND, NOR, $==$, ...

Binary Trees



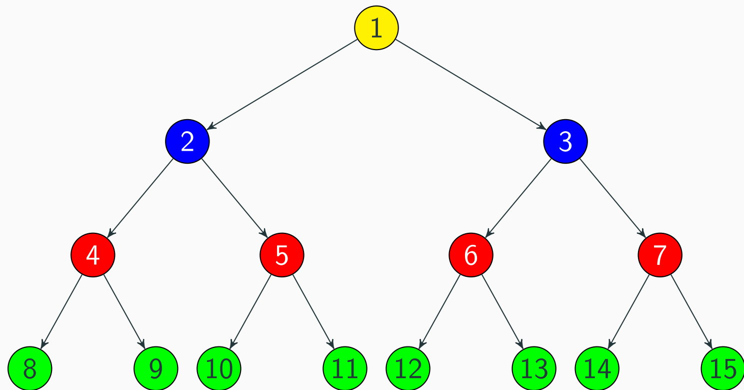
Binary Trees – Layer Structure



For node i:

- children $2 \cdot i$ and $2 \cdot i + 1$ (in next layer)
- parent $\lfloor i/2 \rfloor$

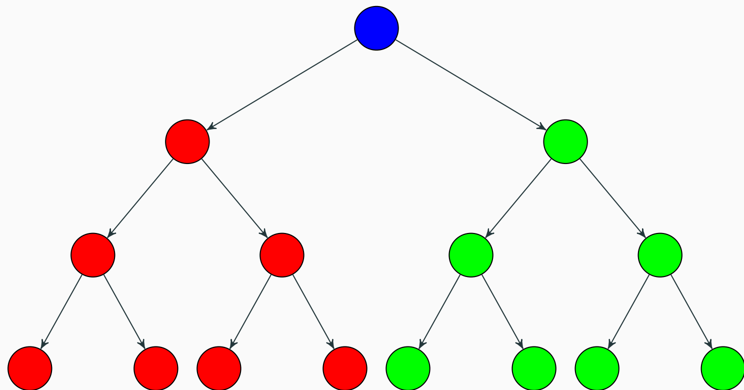
Binary Trees – Layer Structure, Unique Indices



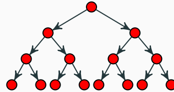
For node i:

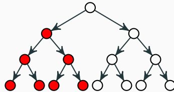
- children $2 \cdot i$ and $2 \cdot i + 1$
- parent $\lfloor i/2 \rfloor$

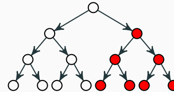
Binary Trees – Recursive Structure

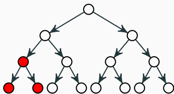


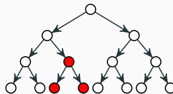
Binary Trees – Subtrees

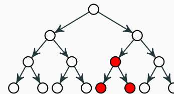


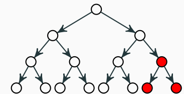


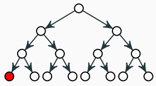


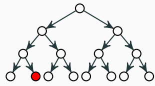


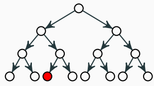


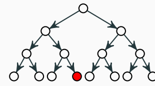


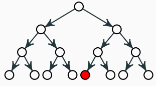


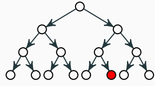


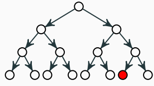


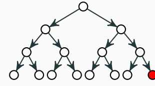






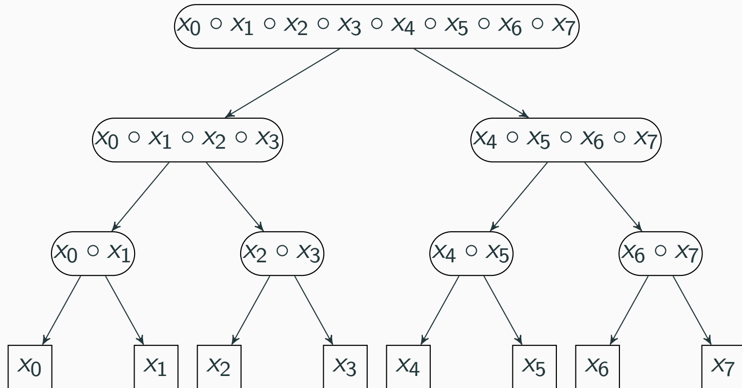




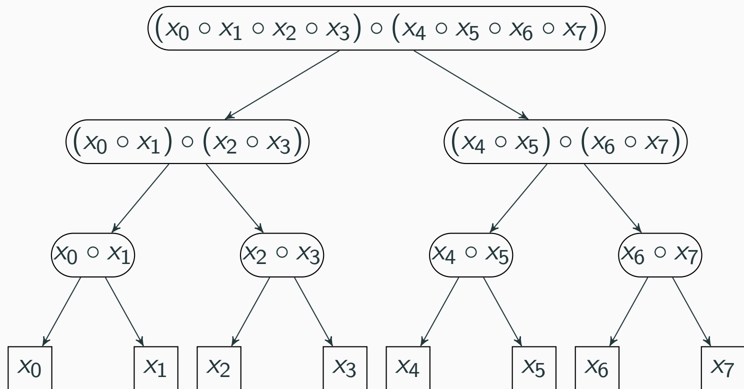


Segment Trees

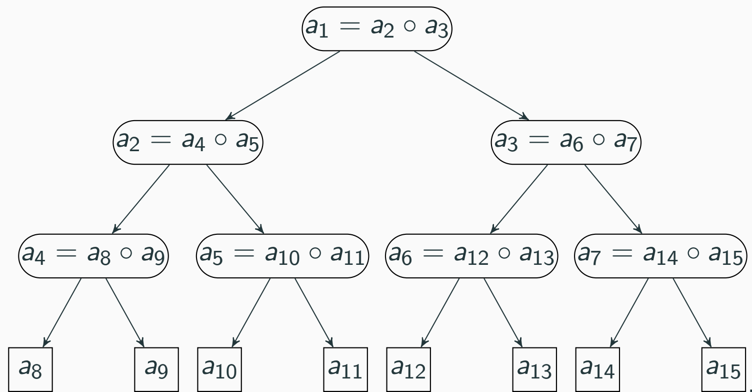
Idea: Lowest layer in binary tree represents the array, parent summarizes the segment in its subtree.



Building a Segment Tree

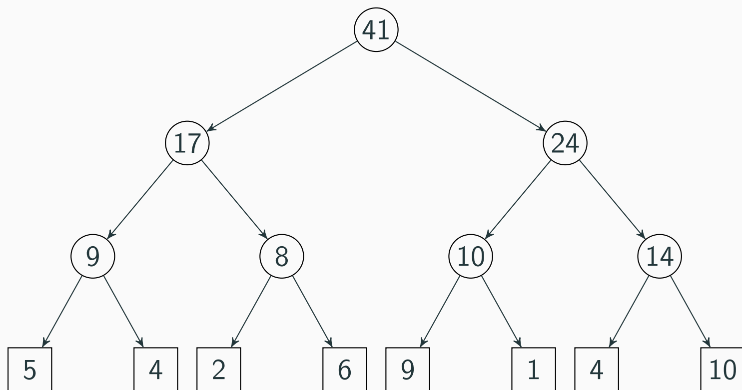


Building a Segment Tree

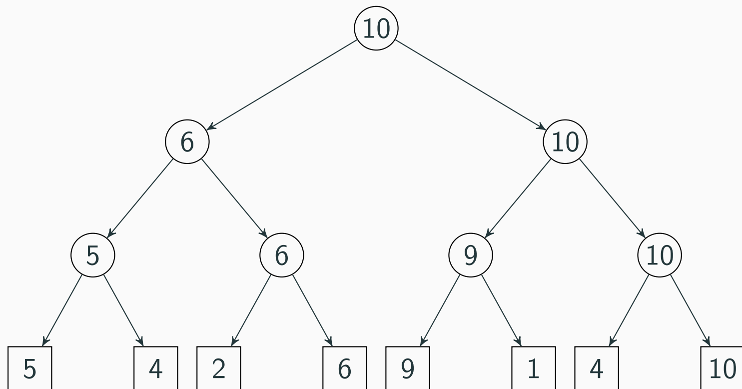


where $[a_8, \ldots, a_{15}] = [x_0, \ldots x_7]$.

Example: Addition

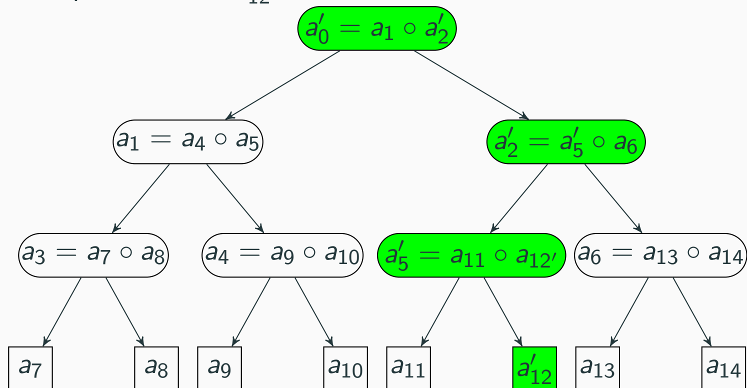


Example: Maximum



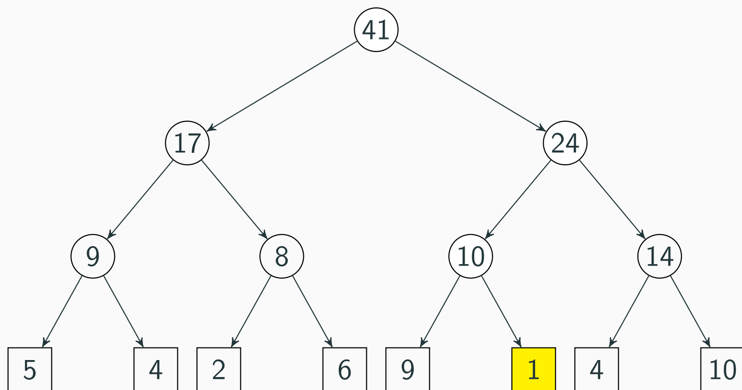
Simple Updates

We update $a_{12} \rightarrow a'_{12}$.

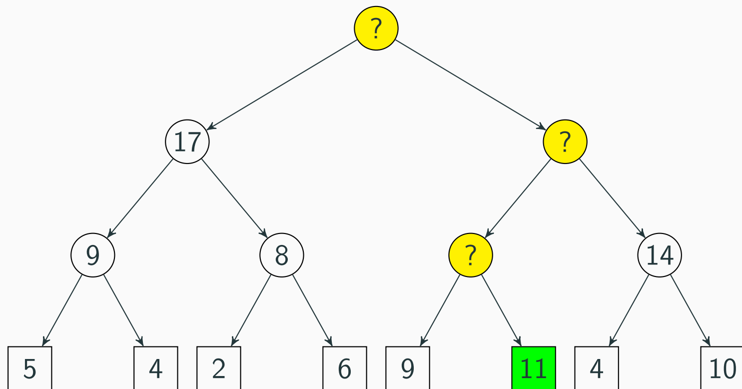


In the segment tree, only one node per layer needs to change.

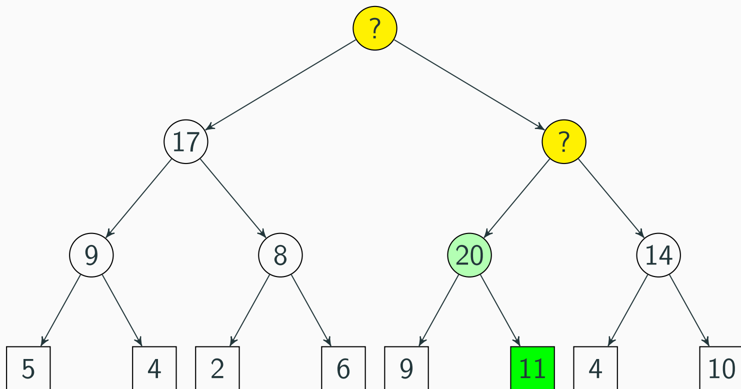
Example: Addition



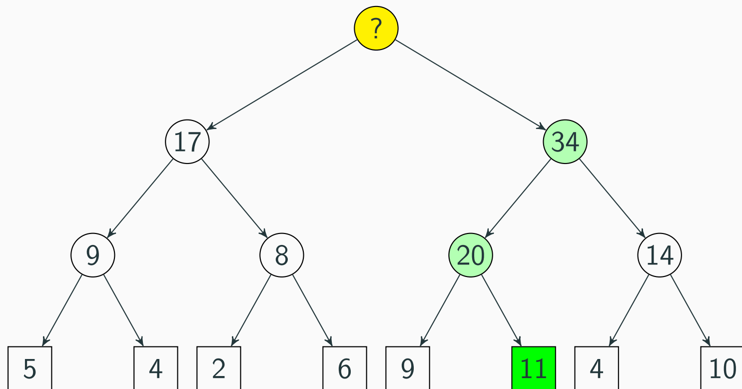
Example: Addition



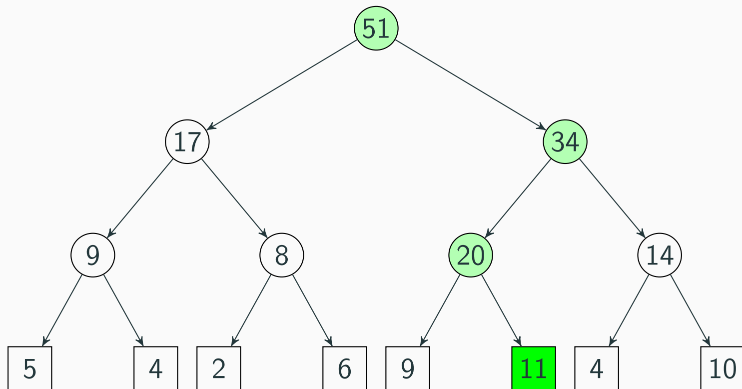
Example: Addition



Example: Addition



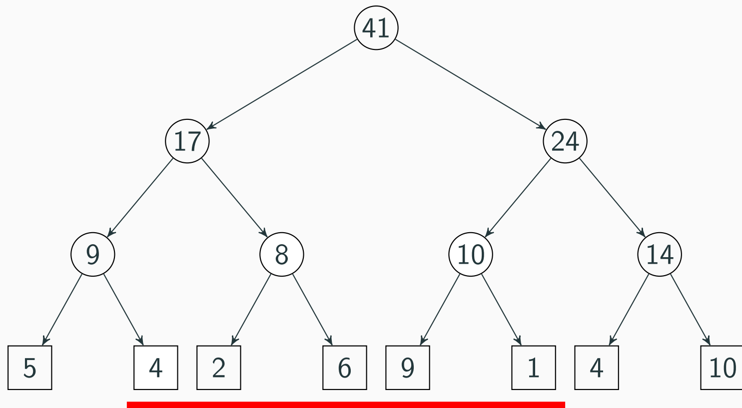
Example: Addition



Segment Queries

We have results for a few selected segments.

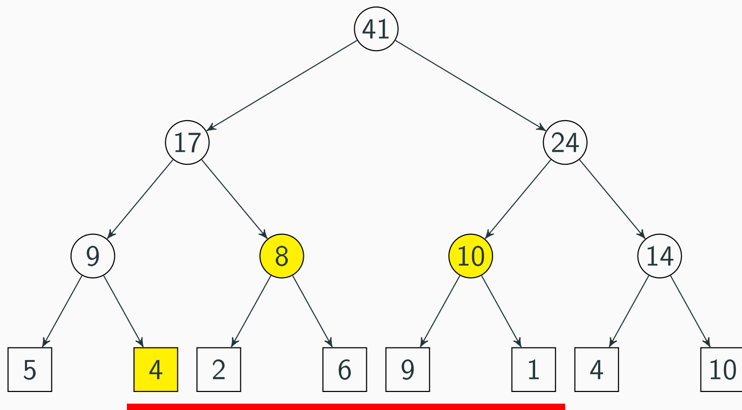
How to query arbitrary segments?



Segment Queries

We have results for a few selected segments.

How to query arbitrary segments?

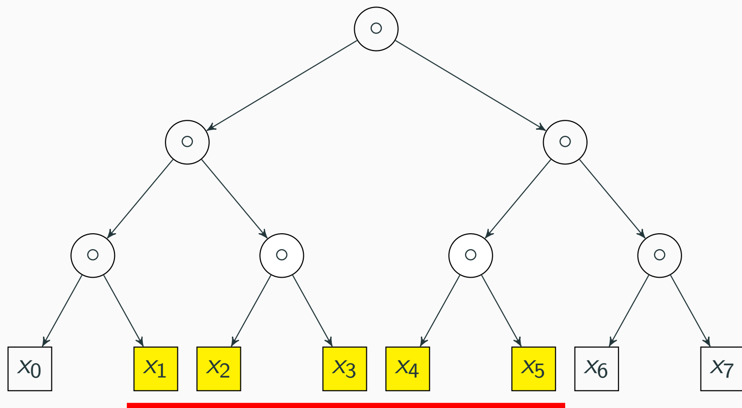


Combine small number of nodes that span the segment together.

$4 + 8 + 10 = 22$.

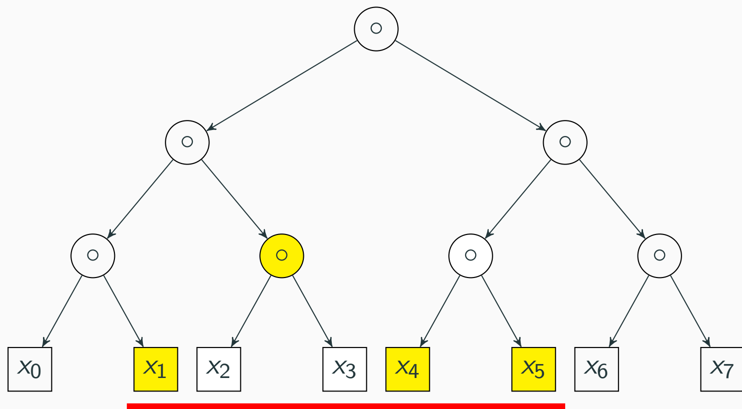
The Canonical Cover, Example 1

Idea: replace siblings by their parent.



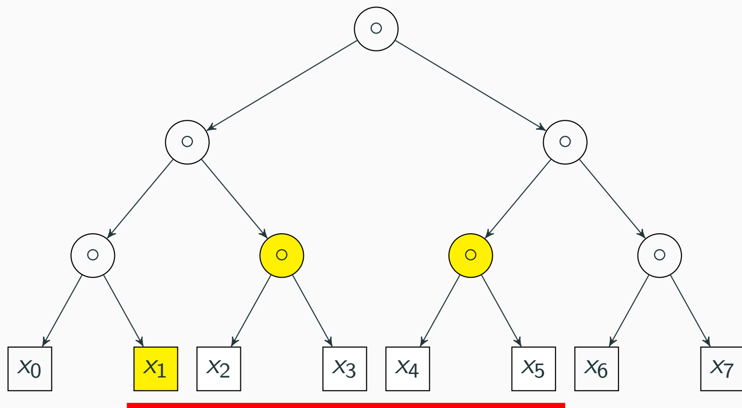
The Canonical Cover, Example 1

Idea: replace siblings by their parent.



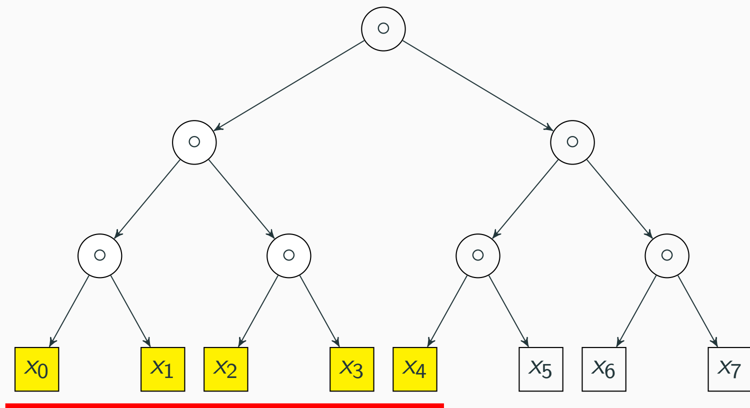
The Canonical Cover, Example 1

Idea: replace siblings by their parent.



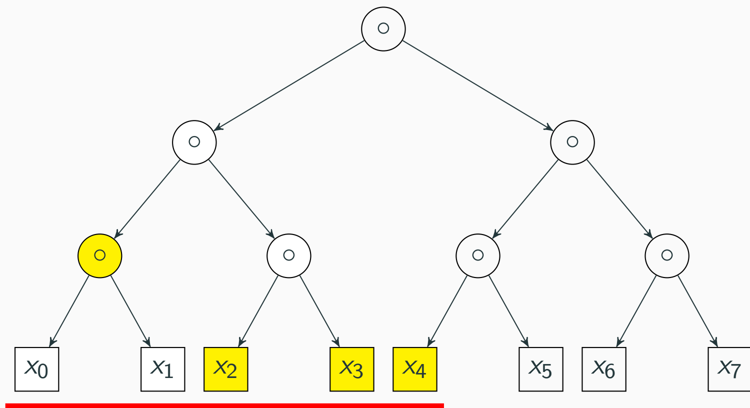
The Canonical Cover, Example 2

Idea: replace siblings by their parent.



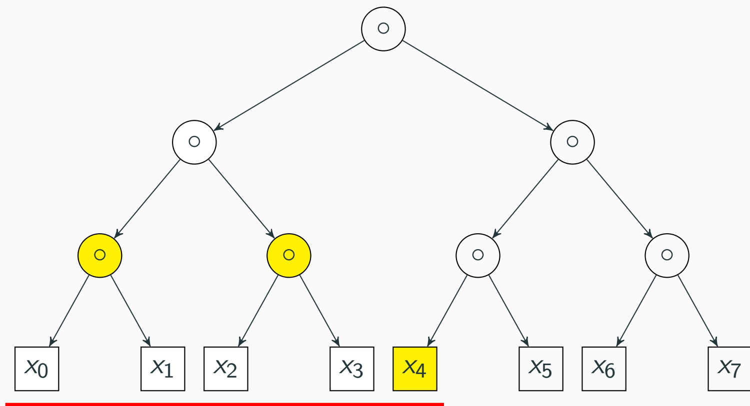
The Canonical Cover, Example 2

Idea: replace siblings by their parent.



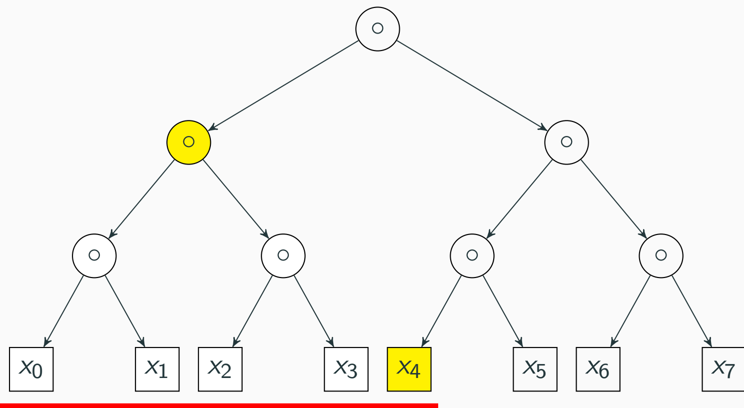
The Canonical Cover, Example 2

Idea: replace siblings by their parent.



The Canonical Cover, Example 2

Idea: replace siblings by their parent.

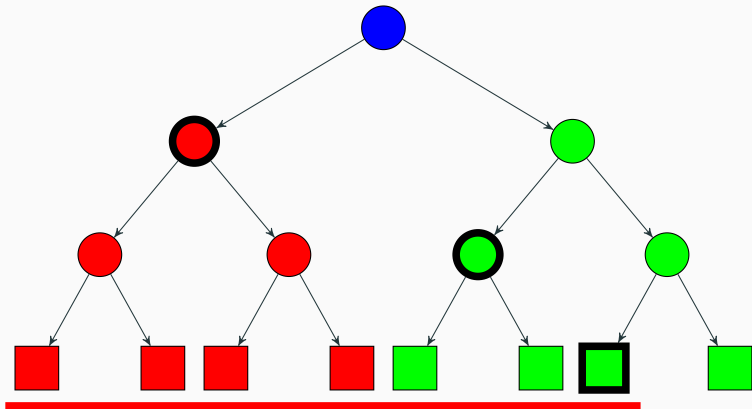


The Canonical Cover is Small

For any segment:

Canonical cover contains at most two nodes per level.

Recursive Computation of Canonical Cover

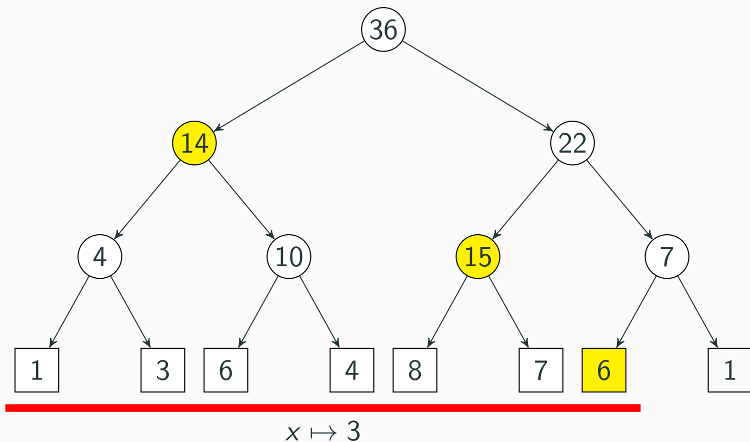


Idea:

- Root in the canonical cover iff the segment spans all leaves.
- Otherwise, take the union of the results for both subtrees.

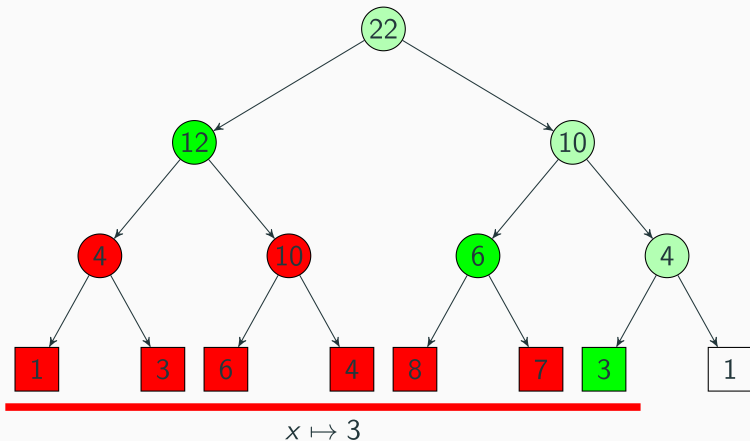
Segment Updates

Naive idea: Update elements of canonical cover.



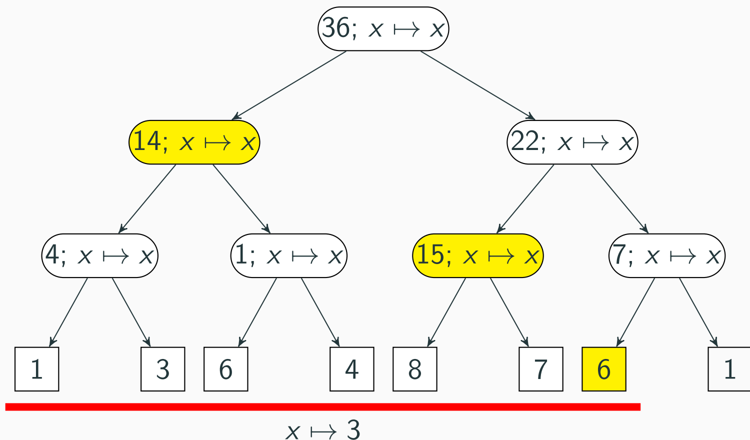
Segment Updates

Naive idea: Update elements of canonical cover.



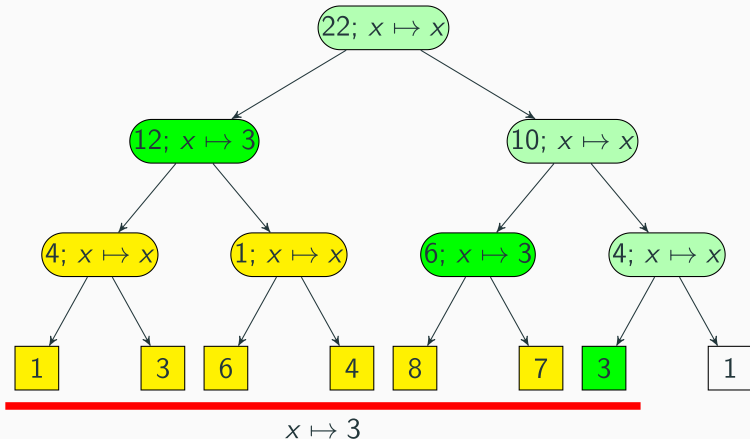
Segment Updates with Lazy Propagation

Idea: Store at each (internal) node an operation that needs to be applied to the child nodes in order to restore the segment tree.



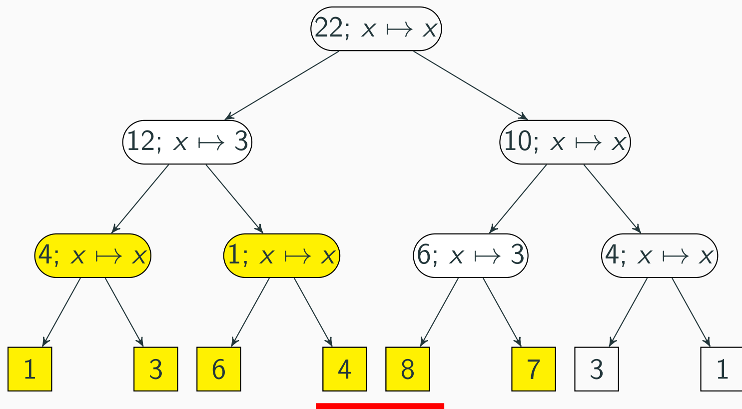
Segment Updates with Lazy Propagation

Idea: Store at each (internal) node an operation that needs to be applied to the child nodes in order to restore the segment tree.



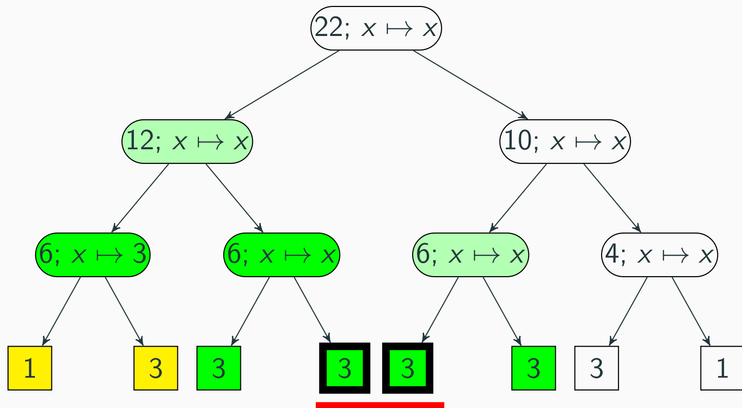
Lazy Propagation during Canonical Cover Computation

Ensure the elements of the computed canonical cover have no pending updates: *Propagate* before recursive calls.



Lazy Propagation during Canonical Cover Computation

Ensure the elements of the computed canonical cover have no pending updates: *Propagate* before recursive calls.



Implementation: Representation

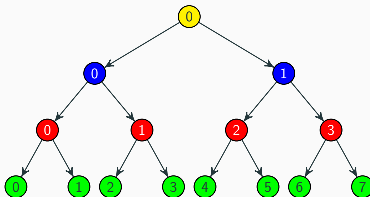
Local node data

```
struct Data{
  Value value;
  Operation lazy;
  int left,right; // left and right borders
  void apply(Operation op);
};
```

(To save space, `left` and `right` can be computed on the fly.)

Implementation: Representation

```
struct SegTree{
  vector<vector<Data>> tree;
};
```

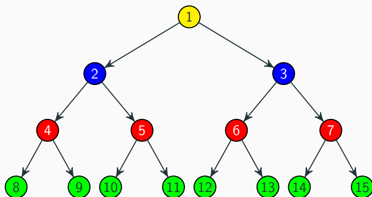


For node i:

- children $2 \cdot i$ and $2 \cdot i + 1$ (in next layer)
- parent $\lfloor i/2 \rfloor$

Implementation: Representation

```
struct SegTree{
  vector<Data> tree;
};
```

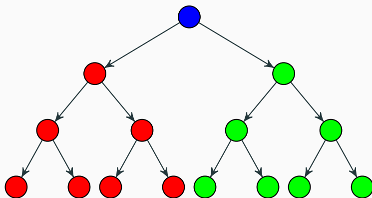


For node i:

- children $2 \cdot i$ and $2 \cdot i + 1$
- parent $\lfloor i/2 \rfloor$

Implementation: Representation

```
struct SegTree{
  Data data;
  SegTree *l,*r; // subtrees
};
```



- Can be initialized lazily (automatic coordinate compression)
- Easily extended to support persistence.

Drawback: Less efficient.

Implementation: Data

We support sum queries, and additive and absolute updates.

```
struct Data{
  int value,left,right;
  pair<bool,int> lazy;
  bool needPropagate()
  { return lazy.first || lazy.second != 0; }
  void apply(pair<bool,int> op){
    if(op.first){ // with reset
      value = op.second*(right-left+1);
      lazy = op;
    }else{
      value += op.second*(right-left+1);
      lazy.second += op.second;
} } };
```

Implementation: Initialization

```
int combine(int a,int b){ return a + b; }
struct SegTree{ // ...
  vector<Data> tree;
  SegTree(vector<int> &data){
    int n=1;
    while(n<2*data.size()) n*=2;
    tree.resize(n);
    for(int i=0;i<n/2;i++){
      tree[n/2+i].value=i<data.size()?data[i]:0;
      tree[n/2+i].left=tree[n/2+i].right=i;
    }
    for(int i=n/2-1;i>=1;i--){
      tree[i].value=combine(tree[2*i].value,
                            tree[2*i+1].value);
      tree[i].left=tree[2*i].left;
      tree[i].right=tree[2*i+1].right;
} } }
```

Implementation: Propagation

```
struct SegTree{
  // ...
  void propagate(int i){
    if(!tree[i].needPropagate()) return;
    for(int k:{0,1}){
      update(2*i+k,tree[i].left,
                   tree[i].right,
                   tree[i].lazy);
    }
    tree[i].lazy={false,0}; // nop
  }
  // ...
}
```

Implementation: Query

```
struct SegTree{ // ...
  int query(int i,int l,int r){
    if(l <= tree[i].left && tree[i].right <= r){
      return tree[i].value;
    }
    if(tree[i].right < l || r < tree[i].left){
      return 0;
    }
    propagate(i);
    return combine(query(2*i,l,r),
                   query(2*i+1,l,r));
  } // ...
}
```

Implementation: Update

```
struct SegTree{// ...
  void update(int i,int l,int r,pair<bool,int> op){
    if(l<=tree[i].left && tree[i].right<=r){
      tree[i].apply(op);
      return;
    }
    if(tree[i].right < l || r < tree[i].left){
      return;
    }
    propagate(i);
    for(int k:{0,1}){
      update(2*i+k,l,r,op);
    }
    tree[i].value=combine(tree[2*i].value,
                          tree[2*i+1].value);
} }
```