Segment Tree

Arnar Bjarni Arnarson

September 21, 2025

School of Computer Science
Reykjavík University

Range queries

- We have an array A of size n.

Range queries

- We have an array A of size n.
- Given i, j, we want to answer:

Range queries

- We have an array A of size n.
- Given i, j, we want to answer:
  - $\max(A[i], A[i+1], \ldots, A[j-1], A[j])$

Range queries

- We have an array A of size n.
- Given i, j, we want to answer:
  - $\max(A[i], A[i+1], \ldots, A[j-1], A[j])$
  - $\min(A[i], A[i+1], \ldots, A[j-1], A[j])$

Range queries

- We have an array A of size n.
- Given i, j, we want to answer:
  - $\max(A[i], A[i+1], \ldots, A[j-1], A[j])$
  - $\min(A[i], A[i+1], \ldots, A[j-1], A[j])$
  - $\mathrm{sum}(A[i], A[i+1], \ldots, A[j-1], A[j])$

Range queries

- We have an array A of size n.
- Given i, j, we want to answer:
  - $\max(A[i], A[i+1], \ldots, A[j-1], A[j])$
  - $\min(A[i], A[i+1], \ldots, A[j-1], A[j])$
  - $\operatorname{sum}(A[i], A[i+1], \ldots, A[j-1], A[j])$
- We want to answer these queries efficiently, or in other words, without looking through all elements.

Range queries

- We have an array A of size n.
- Given i, j, we want to answer:
    - $\max(A[i], A[i+1], \ldots, A[j-1], A[j])$
    - $\min(A[i], A[i+1], \ldots, A[j-1], A[j])$
    - $\text{sum}(A[i], A[i+1], \ldots, A[j-1], A[j])$
- We want to answer these queries efficiently, or in other words, without looking through all elements.
- Sometimes we also want to update elements.

Segment Tree

- We create a perfect binary tree where the leaves are the elements of the array.

Segment Tree

- We create a perfect binary tree where the leaves are the elements of the array.
- Then each internal vertex is the sum of the values below it.

Segment Tree

- We create a perfect binary tree where the leaves are the elements of the array.
- Then each internal vertex is the sum of the values below it.
- Then we have $\mathcal{O}(n)$ nodes and each query can be pieced together from $\mathcal{O}(\log(n))$ node values.

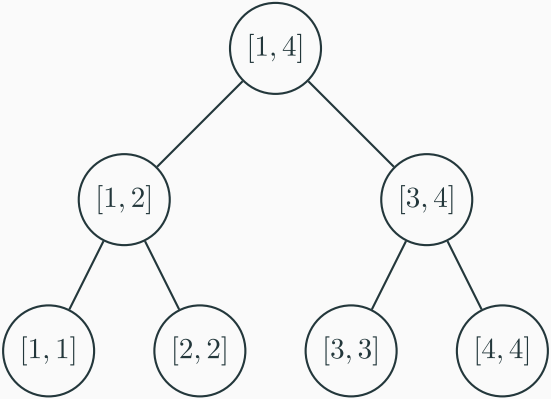
Segment Tree

- We create a perfect binary tree where the leaves are the elements of the array.
- Then each internal vertex is the sum of the values below it.
- Then we have $\mathcal{O}(n)$ nodes and each query can be pieced together from $\mathcal{O}(\log(n))$ node values.
- We travel down the tree looking for the left and right end points, adding intervals that are completely inside our query range.

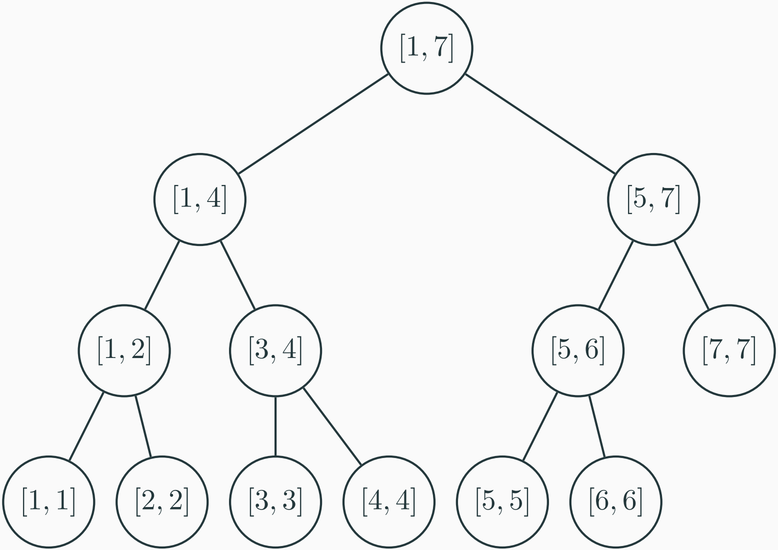
Segment Tree

- We create a perfect binary tree where the leaves are the elements of the array.
- Then each internal vertex is the sum of the values below it.
- Then we have $\mathcal{O}(n)$ nodes and each query can be pieced together from $\mathcal{O}(\log(n))$ node values.
- We travel down the tree looking for the left and right end points, adding intervals that are completely inside our query range.
- When we update a value we only need to update the parents of that node up to the root, at most $\mathcal{O}(\log(n))$ nodes.

Drawn Segment Tree, $n = 4$



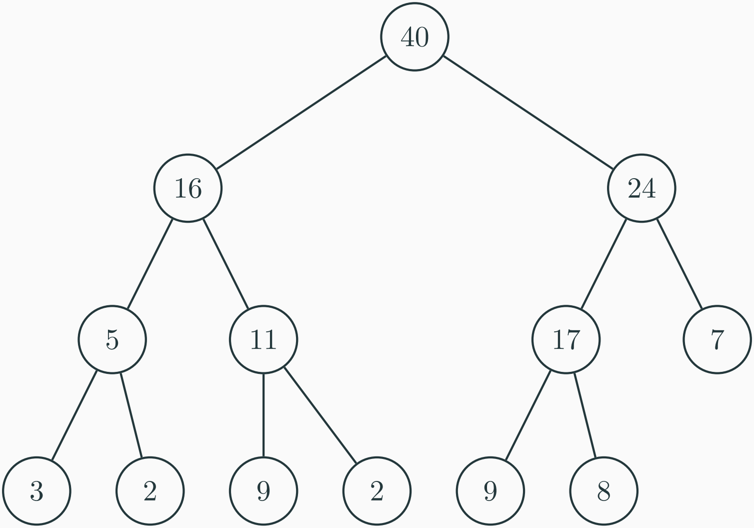
Drawn Segment Tree, $n = 7$



Segment Tree - Code

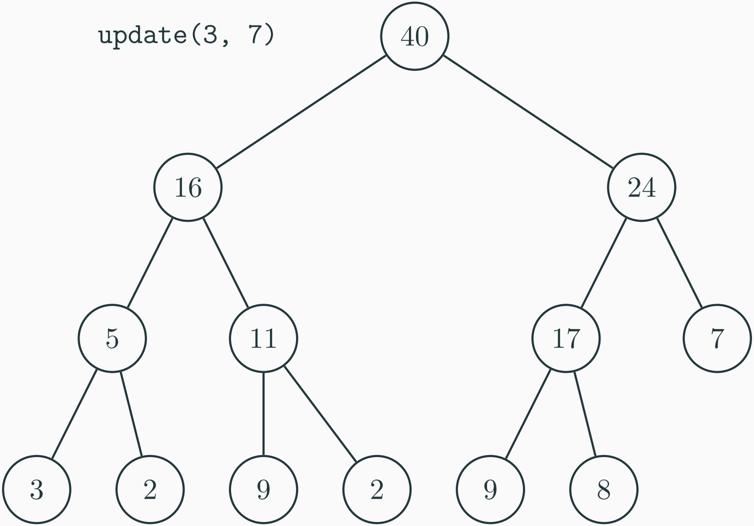
```cpp
struct segment_tree {
    segment_tree *left, *right;
    int from, to, value;
    segment_tree(int from, int to)
        : from(from), to(to), left(NULL), right(NULL), value(0) { }
};

segment_tree* build(const vector<int> &arr, int l, int r) {
    if (l > r) return NULL;
    segment_tree *res = new segment_tree(l, r);
    if (l == r) {
        res->value = arr[l];
    } else {
        int m = (l + r) / 2;
        res->left = build(arr, l, m);
        res->right = build(arr, m + 1, r);
        if (res->left != NULL) res->value += res->left->value;
        if (res->right != NULL) res->value += res->right->value;
    }
    return res;
}
```

Updates

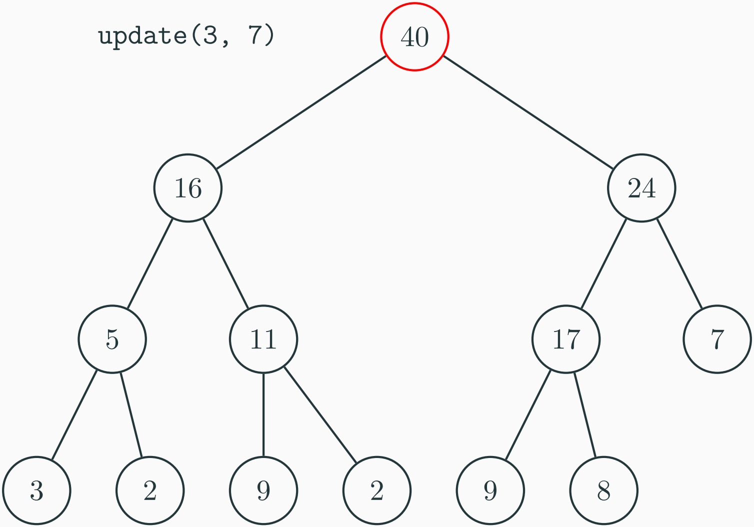


Updates

`update(3, 7)`

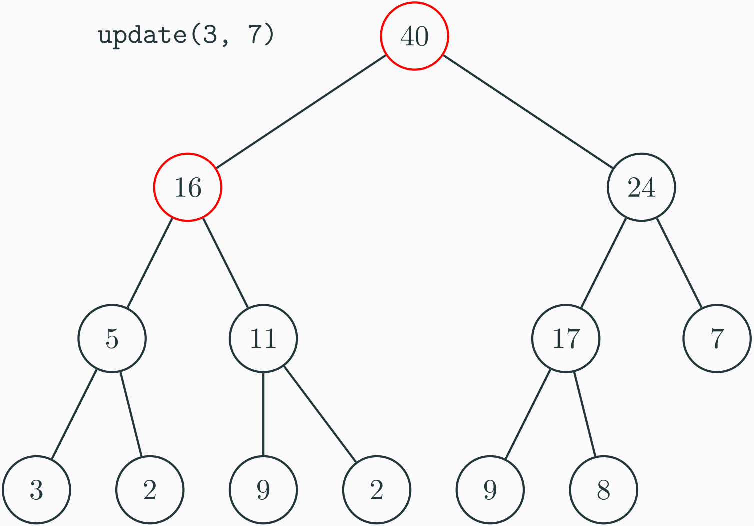


Updates

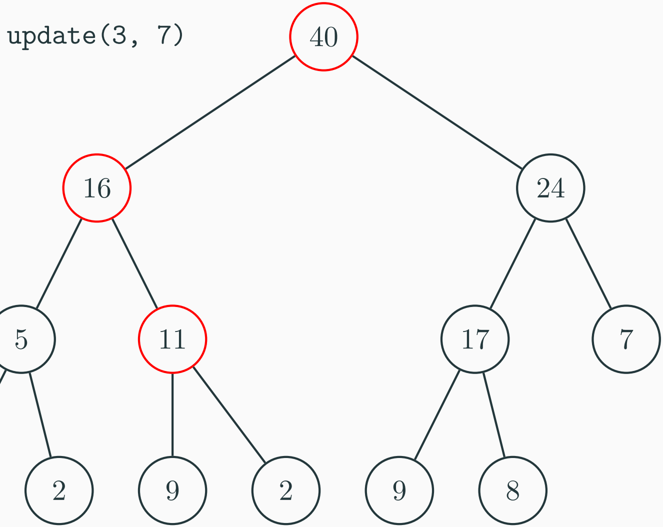


Updates

`update(3, 7)`

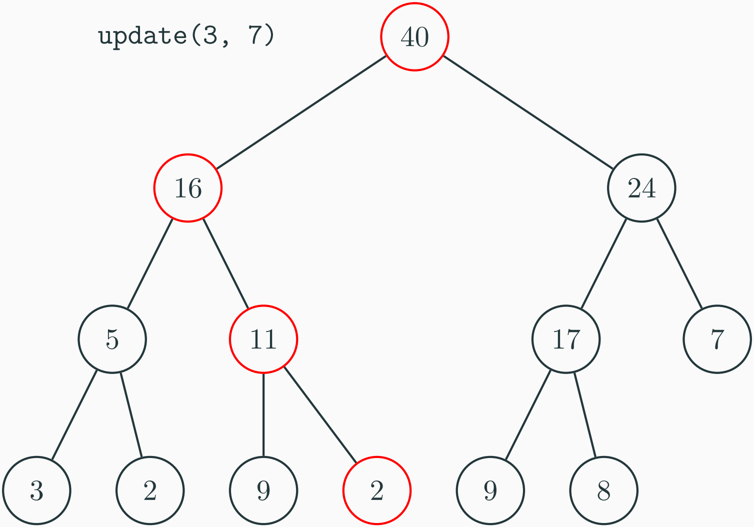


Updates

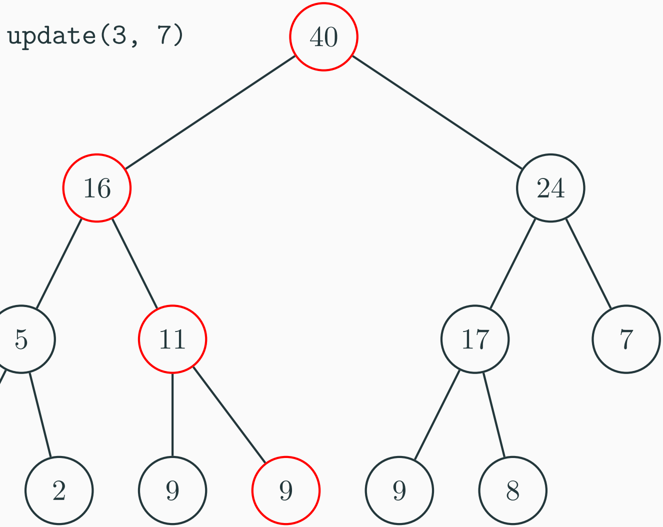


Updates

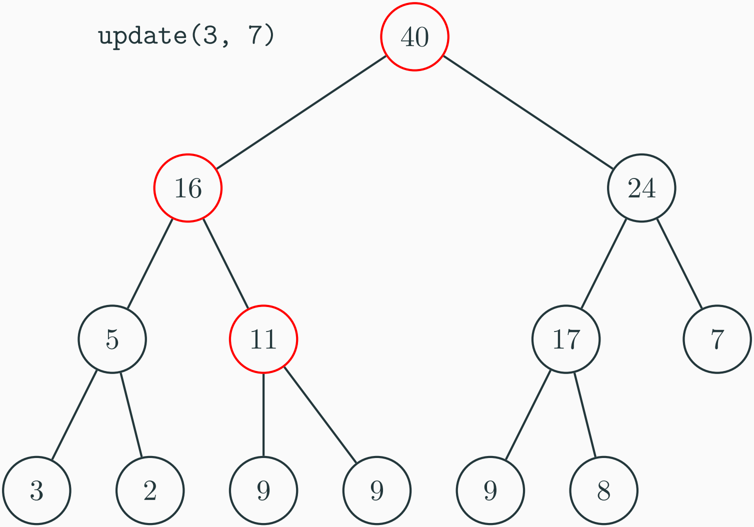
`update(3, 7)`



Updates

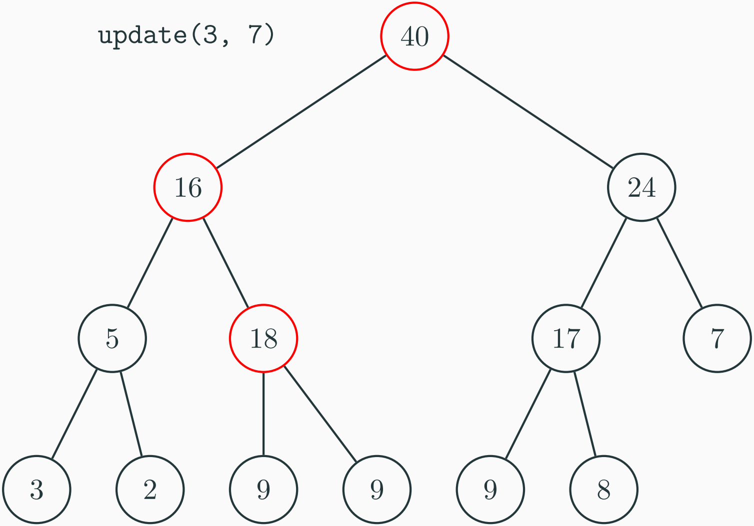


Updates

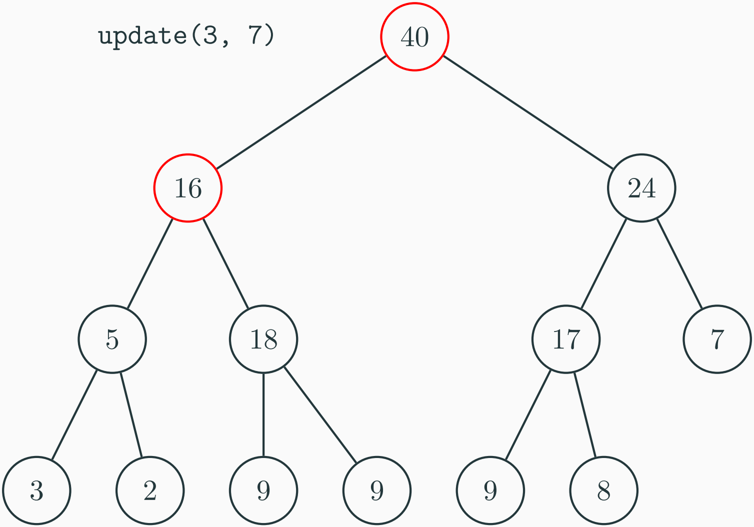


Updates

`update(3, 7)`

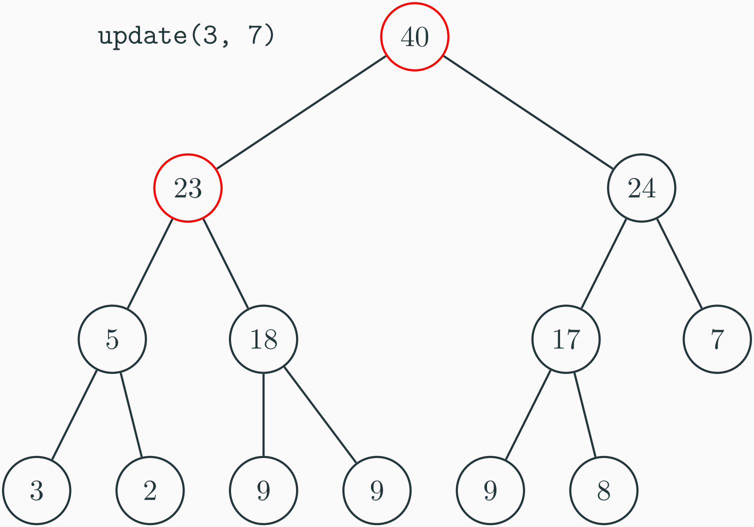


Updates

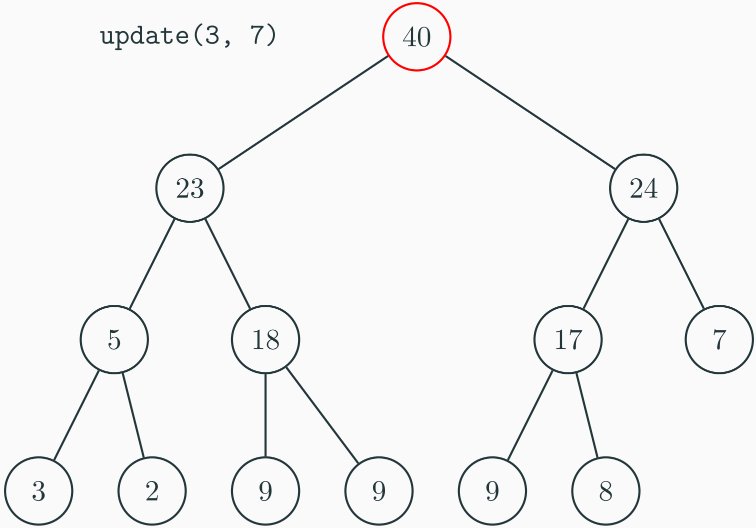


Updates

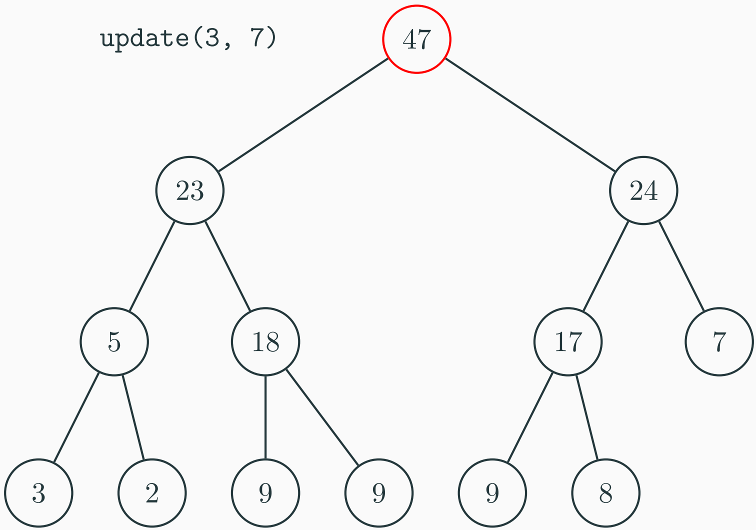
`update(3, 7)`



Updates

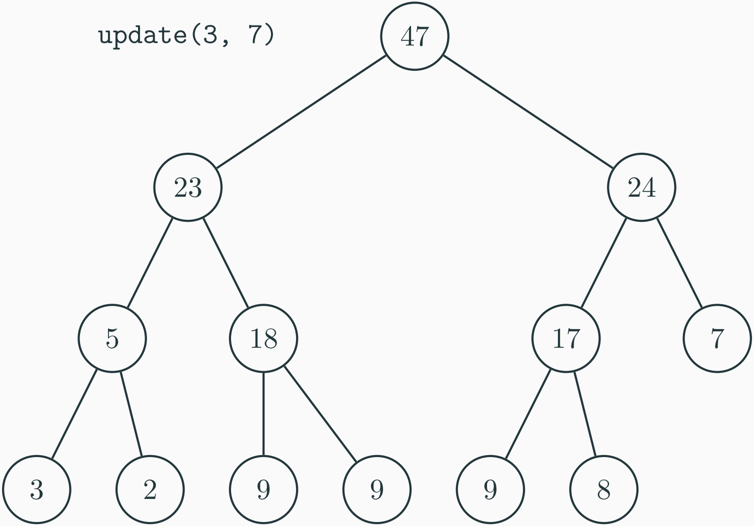


Updates

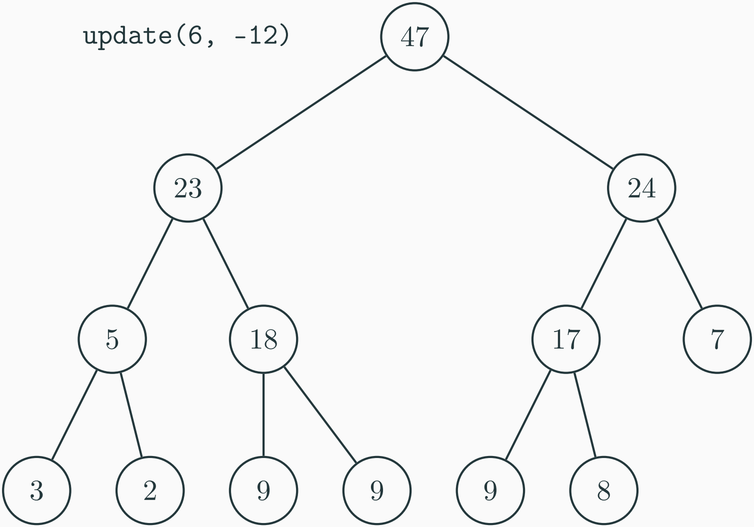


Updates

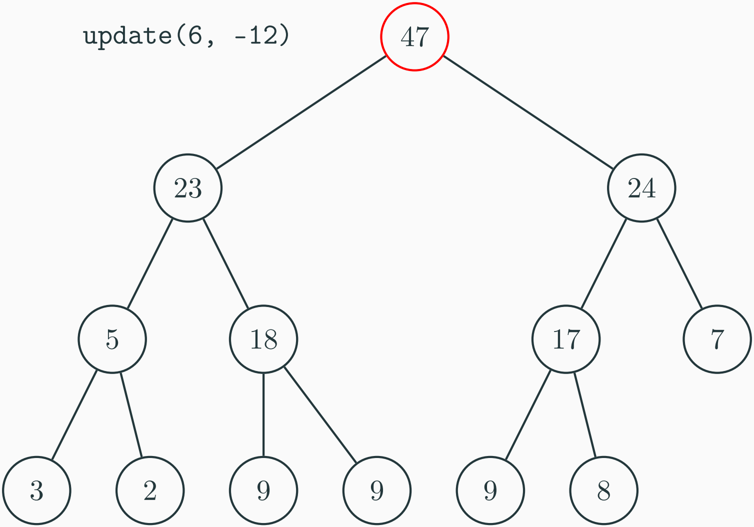
`update(3, 7)`



Updates

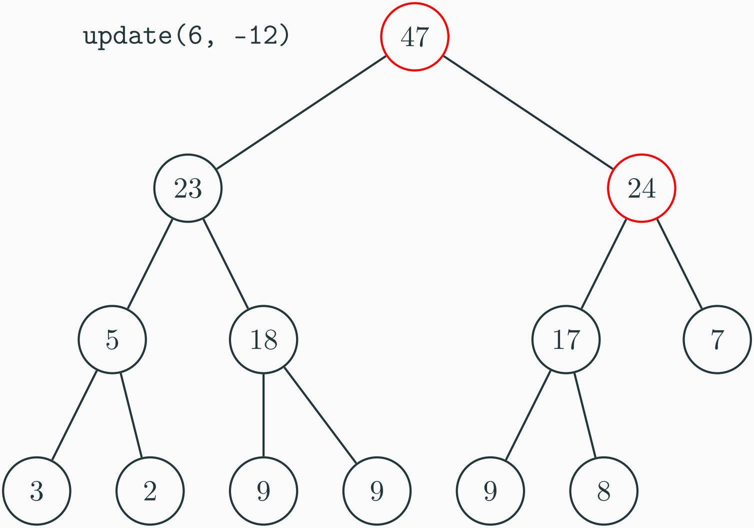


Updates



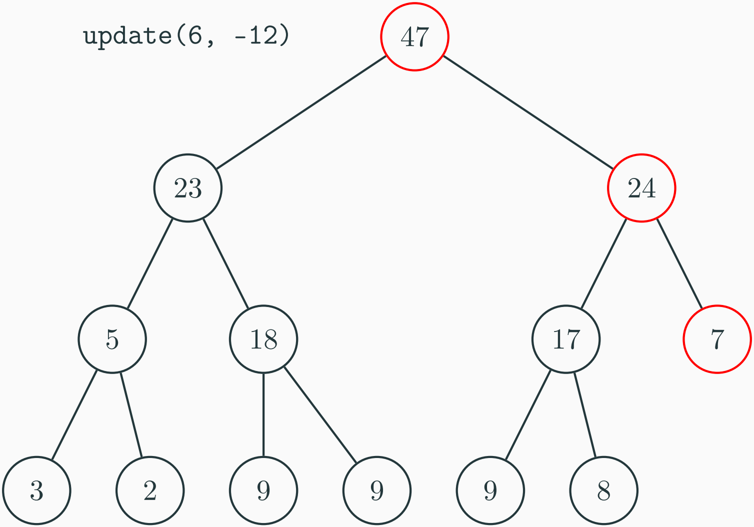
Updates

`update(6, -12)`



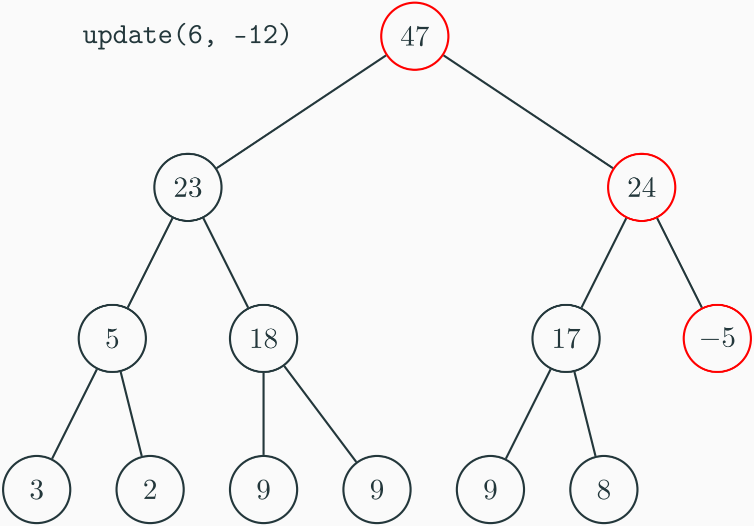
Updates

`update(6, -12)`



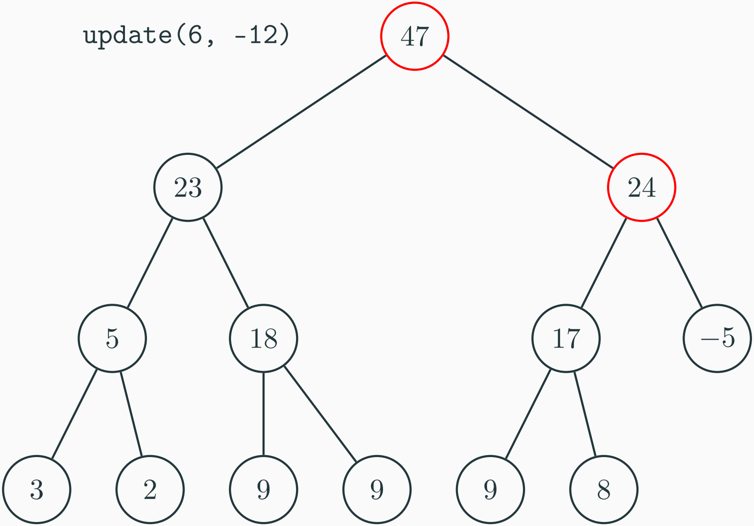
Updates

`update(6, -12)`

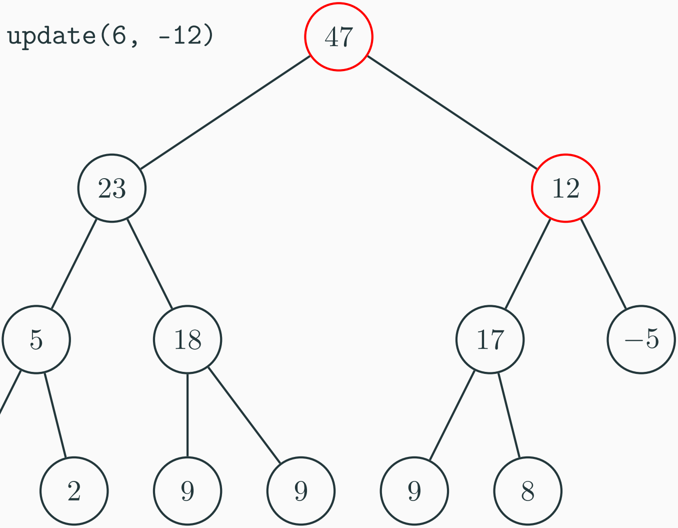


Updates

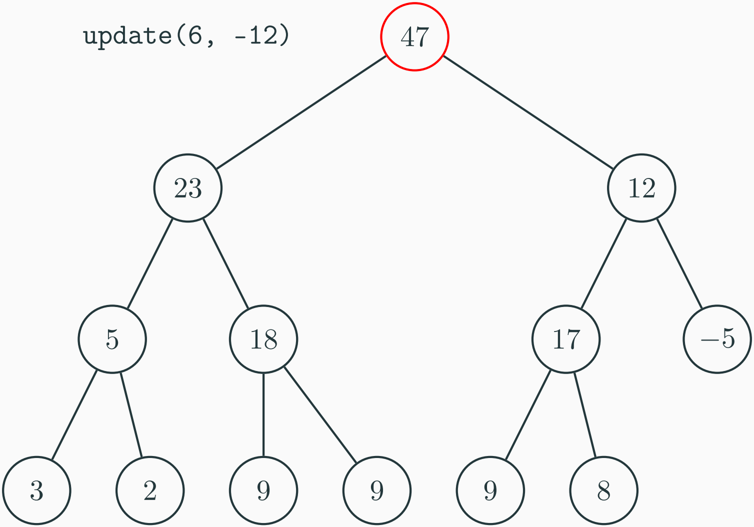
`update(6, -12)`



Updates

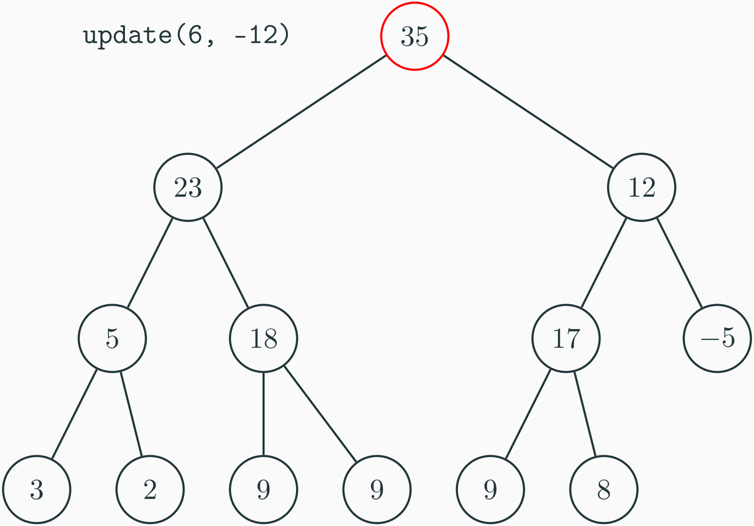


Updates

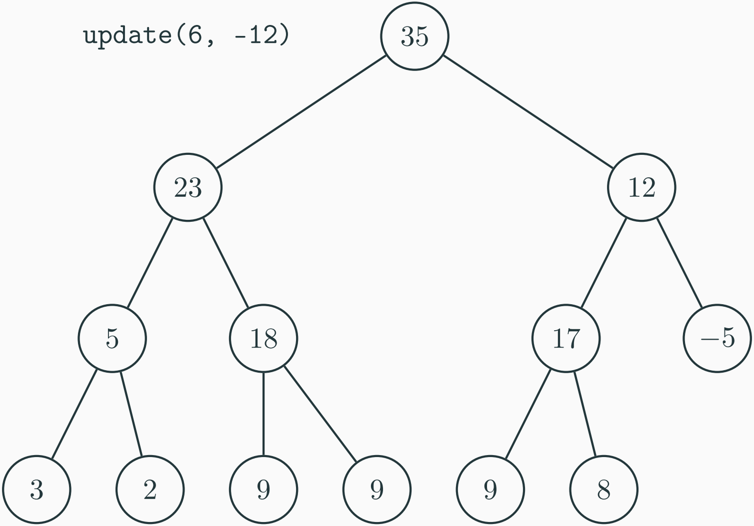


Updates

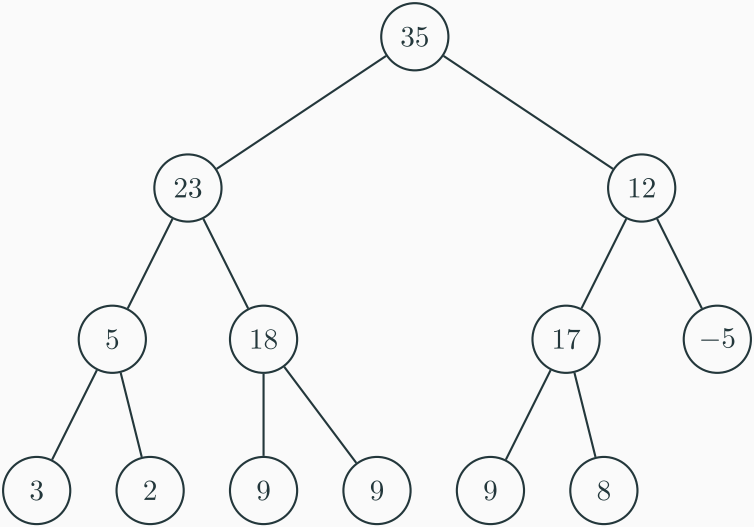
`update(6, -12)`



Updates



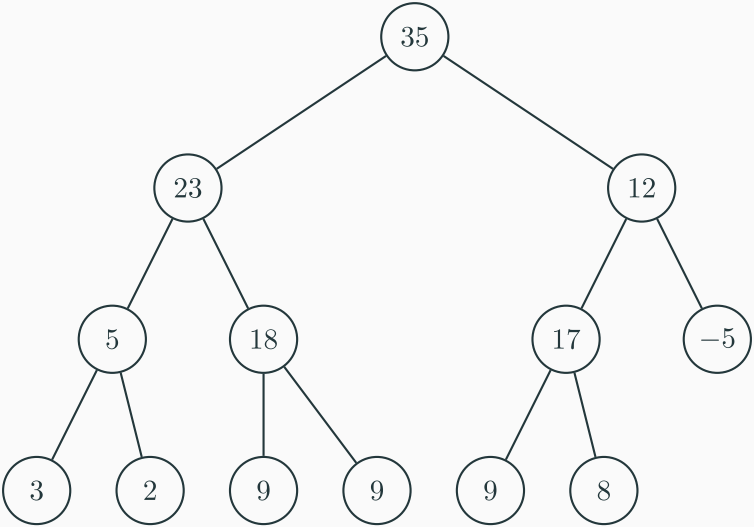
Updates



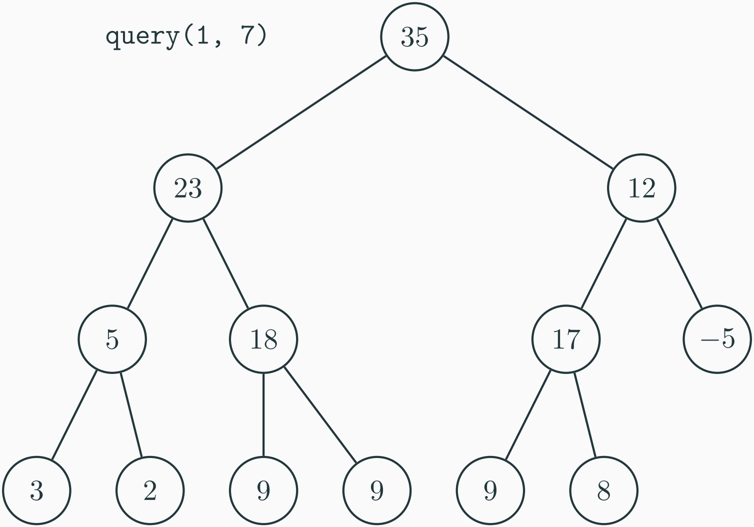
Updating a Segment Tree - Code

```
int update(segment_tree *tree, int i, int val) {
    if (tree == NULL) return 0;
    if (tree->to < i) return tree->value;
    if (i < tree->from) return tree->value;
    if (tree->from == tree->to && tree->from == i) {
        tree->value = val;
    } else {
        tree->value = update(tree->left, i, val) + update(tree->right, i, val);
    }
    return tree->value;
}
```

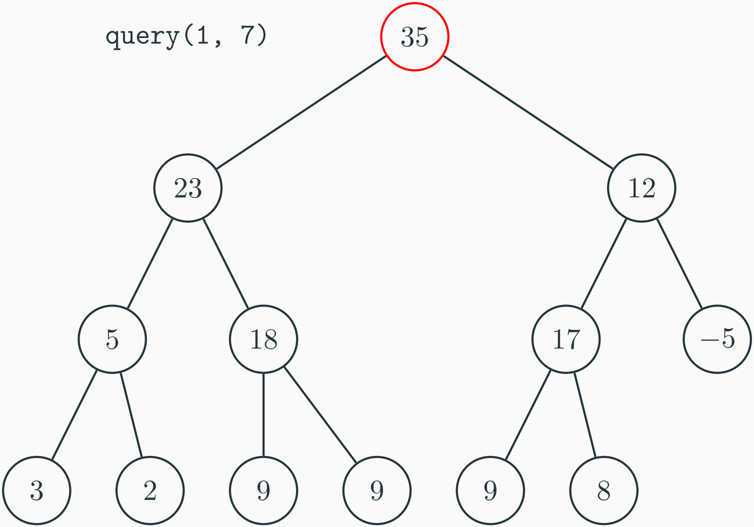
Querying



Querying

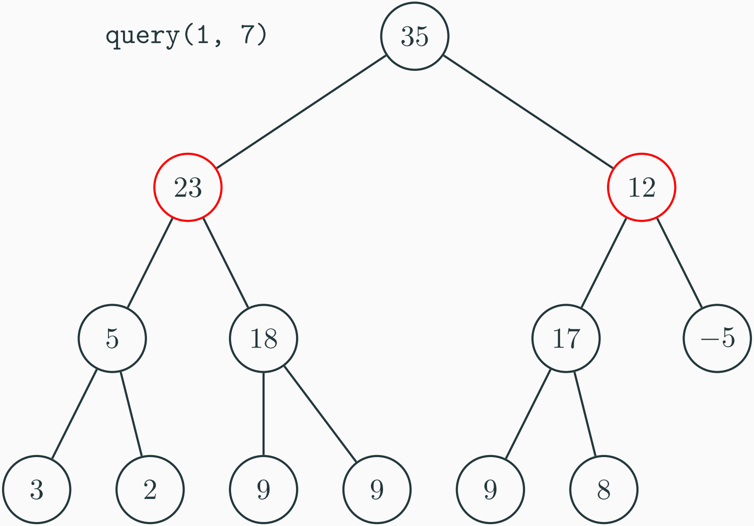


Querying

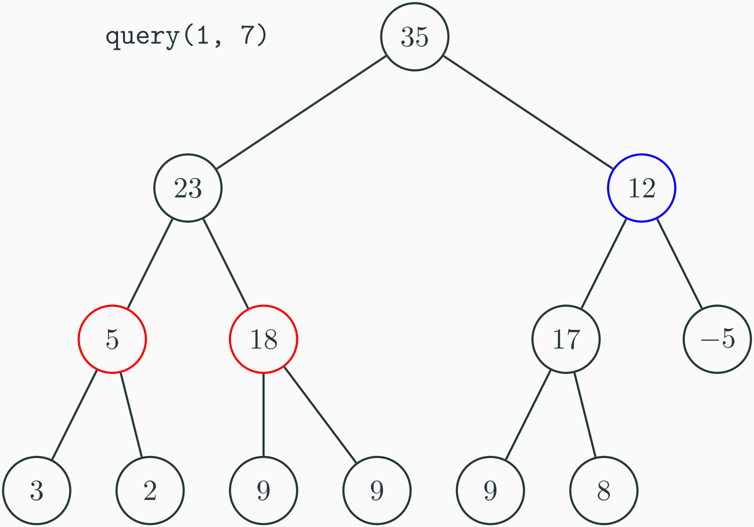


Querying

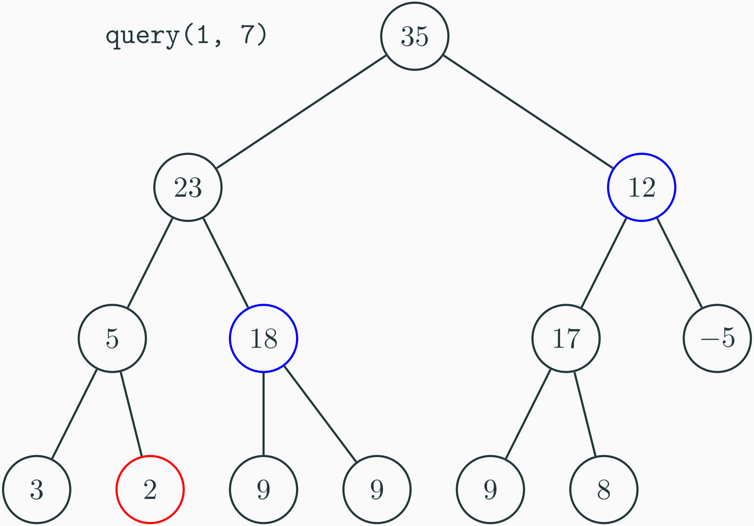
`query(1, 7)`



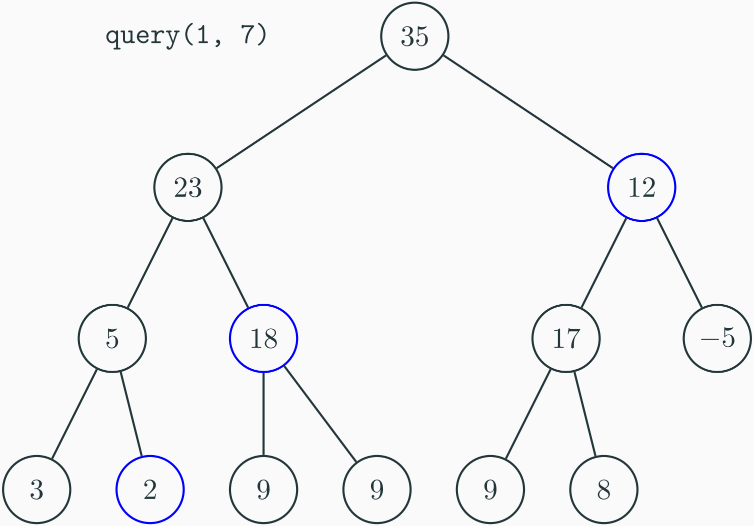
Querying



Querying

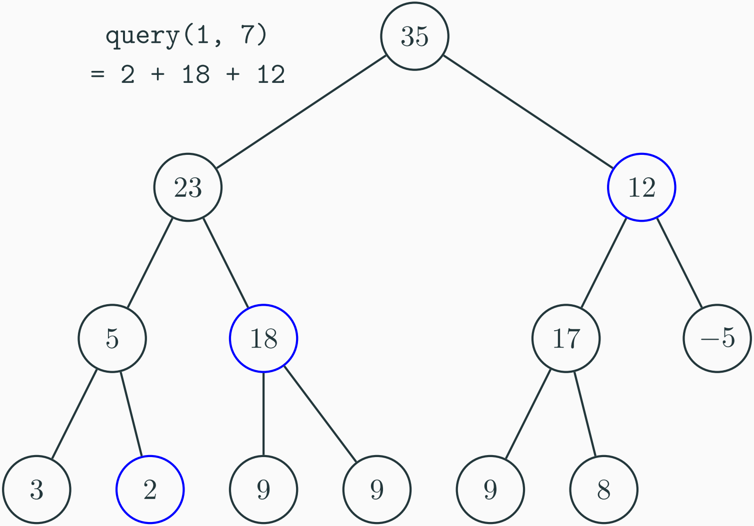


Querying

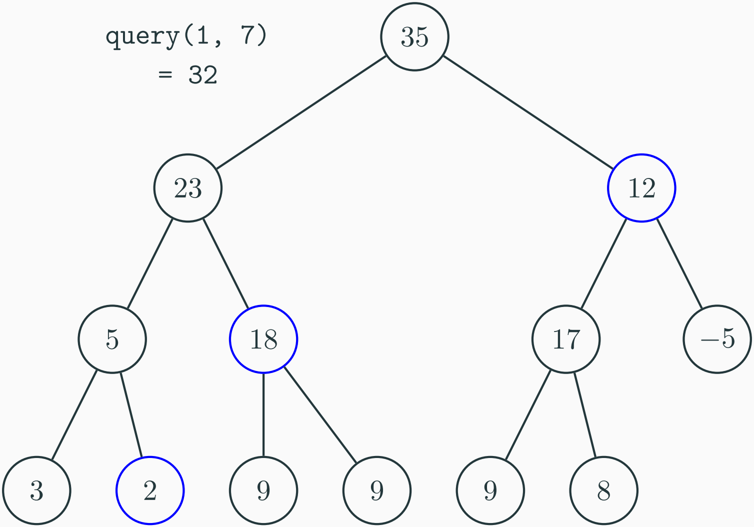


Querying

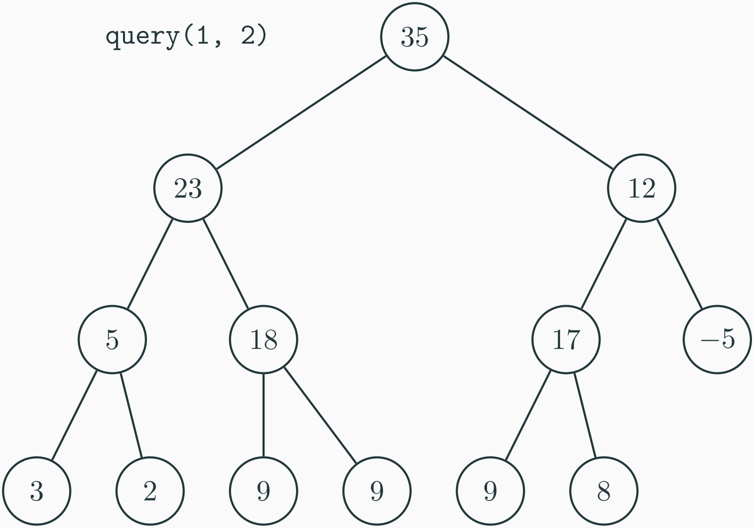
```
query(1, 7)
= 2 + 18 + 12
```



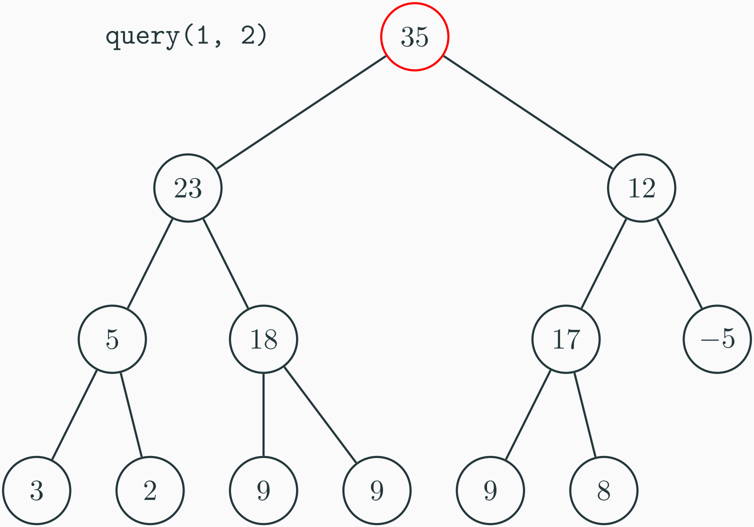
Querying



Querying

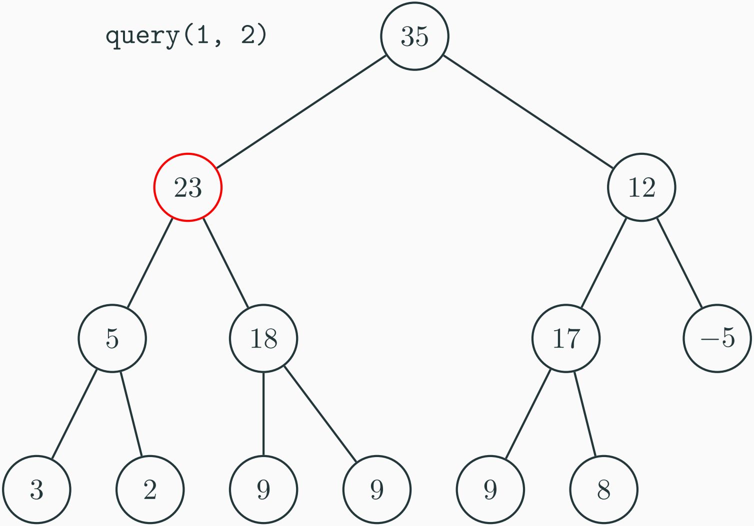


Querying

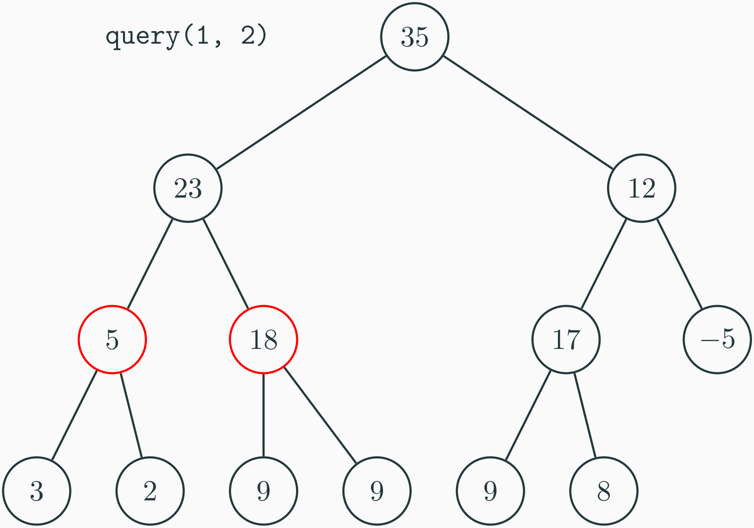


Querying

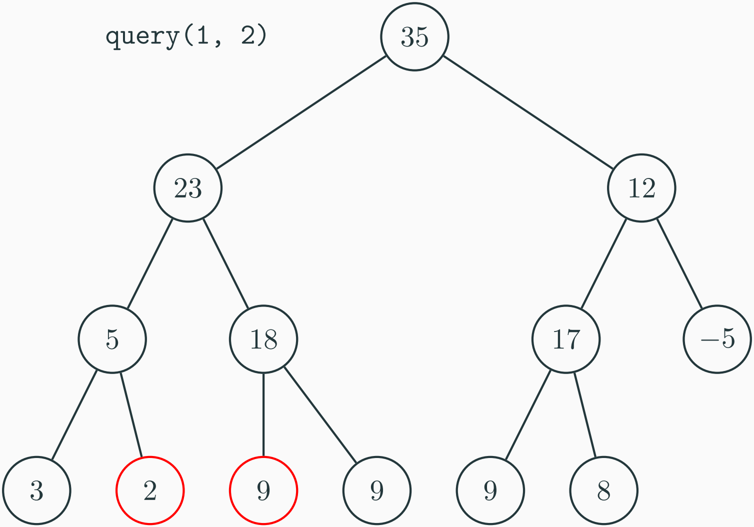
`query(1, 2)`



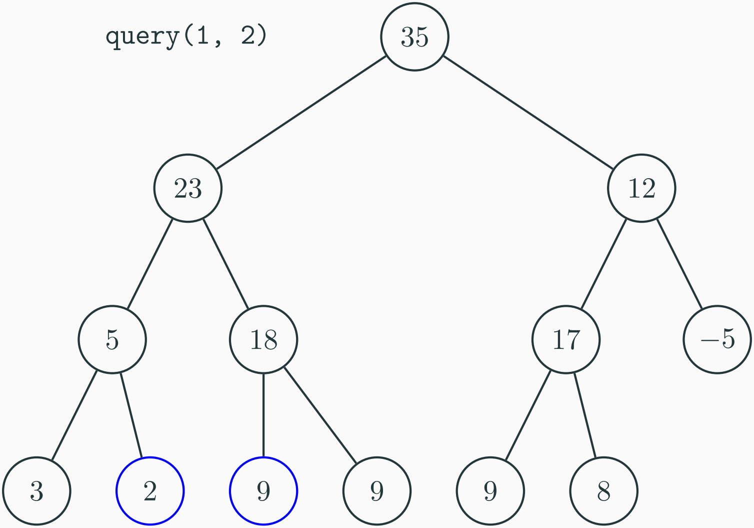
Querying



Querying

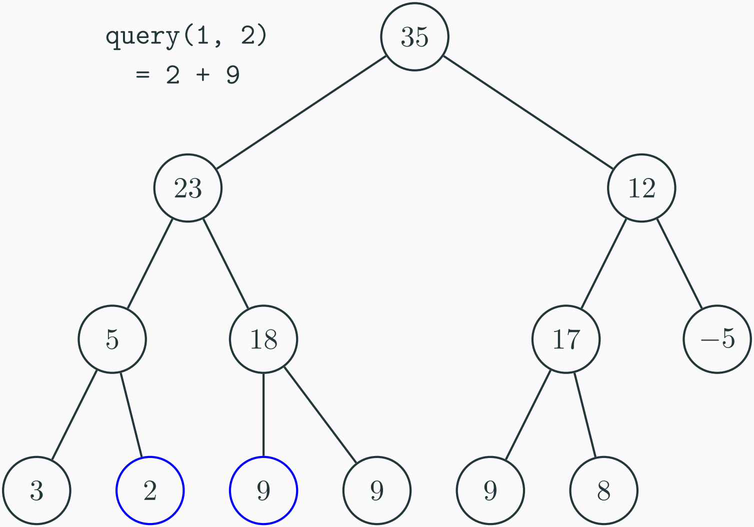


Querying



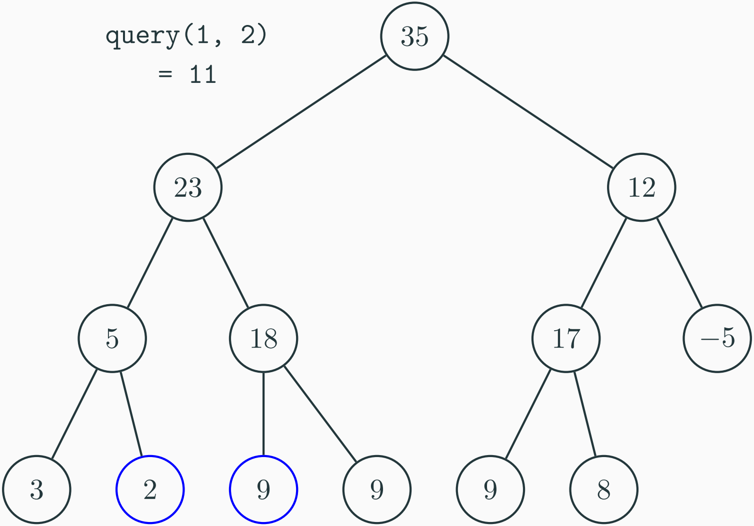
Querying

```
query(1, 2)
  = 2 + 9
```

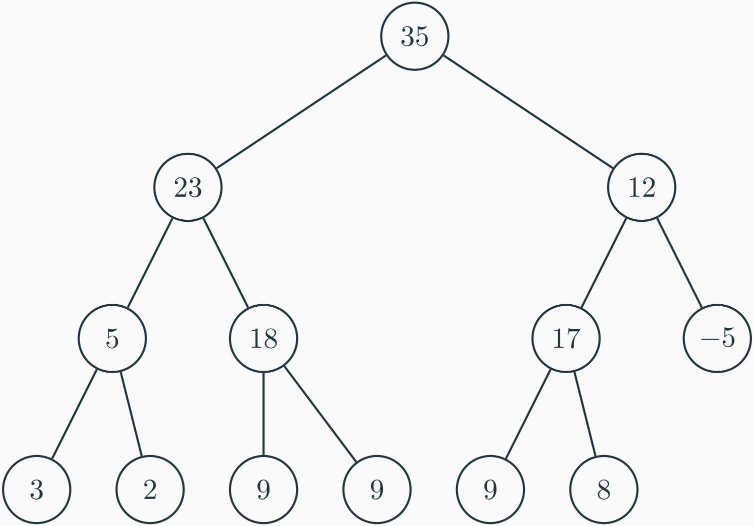


Querying

`query(1, 2)`
`= 11`



Querying



Querying a Segment Tree - Code

```c
int query(segment_tree *tree, int l, int r) {
    if (tree == NULL) return 0;
    if (l <= tree->from && tree->to <= r) return tree->value;
    if (tree->to < l) return 0;
    if (r < tree->from) return 0;
    return query(tree->left, l, r) + query(tree->right, l, r);
}
```

Segment Tree

- Simple to use Segment Trees for $\min$, $\max$, $\gcd$, and other similar operators, basically the same code.

Segment Tree

- Simple to use Segment Trees for $\min$, $\max$, $\gcd$, and other similar operators, basically the same code.
- Any associative operator will work.

Segment Tree

- Simple to use Segment Trees for $\min$, $\max$, $\gcd$, and other similar operators, basically the same code.
- Any associative operator will work.
- So any operator f such that $f(a, f(b, c)) = f(f(a, b), c)$ for all a, b, c.

Example problem: Movie Collection

- https://open.kattis.com/problems/moviecollection

Range updates

- So far, we have only allowed updates to affect a single element.

Range updates

- So far, we have only allowed updates to affect a single element.
- Might want to update multiple elements simultaneously, iterating for each is expensive.

Range updates

- So far, we have only allowed updates to affect a single element.
- Might want to update multiple elements simultaneously, iterating for each is expensive.
- Example: For all indices from 4 to 7 add 13.

Range updates

- So far, we have only allowed updates to affect a single element.
- Might want to update multiple elements simultaneously, iterating for each is expensive.
- Example: For all indices from 4 to 7 add 13.
- Can we make use of our segmented structure to update all indices in range $[l, r]$?

Range updates

- So far, we have only allowed updates to affect a single element.
- Might want to update multiple elements simultaneously, iterating for each is expensive.
- Example: For all indices from 4 to 7 add 13.
- Can we make use of our segmented structure to update all indices in range $[l, r]$?
- Lazy people tend to find efficient ways of doing all that **needs** to be done, but no more.

Range updates

- So far, we have only allowed updates to affect a single element.
- Might want to update multiple elements simultaneously, iterating for each is expensive.
- Example: For all indices from 4 to 7 add 13.
- Can we make use of our segmented structure to update all indices in range $[l, r]$?
- Lazy people tend to find efficient ways of doing all that **needs** to be done, but no more.
- After updating a value, there is no guarantee you will use the updated value afterwards.

Range updates

- So far, we have only allowed updates to affect a single element.
- Might want to update multiple elements simultaneously, iterating for each is expensive.
- Example: For all indices from 4 to 7 add 13.
- Can we make use of our segmented structure to update all indices in range $[l, r]$?
- Lazy people tend to find efficient ways of doing all that **needs** to be done, but no more.
- After updating a value, there is no guarantee you will use the updated value afterwards.
- Idea: Be lazy and procrastinate changes until they are needed!

Lazy propagation

- Add another variable for each node, storing the lazy value

Lazy propagation

- Add another variable for each node, storing the lazy value
- Follow same steps as querying in order to update, applying the update if the node is completely within the range.

Lazy propagation

- Add another variable for each node, storing the lazy value
- Follow same steps as querying in order to update, applying the update if the node is completely within the range.
- When making updates, do not change the original data variable, but rather the lazy variable.

Lazy propagation

- Add another variable for each node, storing the lazy value
- Follow same steps as querying in order to update, applying the update if the node is completely within the range.
- When making updates, do not change the original data variable, but rather the lazy variable.
- When looking at a node, apply the lazy value to the node

Lazy propagation

- Add another variable for each node, storing the lazy value
- Follow same steps as querying in order to update, applying the update if the node is completely within the range.
- When making updates, do not change the original data variable, but rather the lazy variable.
- When looking at a node, apply the lazy value to the node
- After applying, push the lazy value to the two child nodes

Lazy propagation

- Add another variable for each node, storing the lazy value
- Follow same steps as querying in order to update, applying the update if the node is completely within the range.
- When making updates, do not change the original data variable, but rather the lazy variable.
- When looking at a node, apply the lazy value to the node
- After applying, push the lazy value to the two child nodes
- Reset the lazy value.

Lazy propagation

- Add another variable for each node, storing the lazy value
- Follow same steps as querying in order to update, applying the update if the node is completely within the range.
- When making updates, do not change the original data variable, but rather the lazy variable.
- When looking at a node, apply the lazy value to the node
- After applying, push the lazy value to the two child nodes
- Reset the lazy value.
- Traverse to child nodes if needed.

Code example

See implementation example, for example here.