Data Structures

Atli FF

15. september 2023

School of Computer Science
Reykjavík University

Today's material

- Built-in data structures and their applications
- Augmenting a data structure
- Union-Find
- Precomputations like prefix sums
- Square root decomposition
- Segment trees
- Sparse tables

Applications of Arrays and Linked Lists

- Too many to list, more so for arrays

Applications of Arrays and Linked Lists

- Too many to list, more so for arrays
- Most problems require storing data, usually in an array

Applications of Arrays and Linked Lists

- Too many to list, more so for arrays
- Most problems require storing data, usually in an array
- On modern machines, arrays are almost always a better choice than a linked list

Applications of Arrays and Linked Lists

- Too many to list, more so for arrays
- Most problems require storing data, usually in an array
- On modern machines, arrays are almost always a better choice than a linked list
- There are however a few cases where linked lists are better

Example problem: Broken Keyboard

- http://uva.onlinejudge.org/external/119/11988.html

Applications of Stacks

- Processing events in a last-in first-out order

Applications of Stacks

- Processing events in a last-in first-out order
- Simulating recursion

Applications of Stacks

- Processing events in a last-in first-out order
- Simulating recursion
- Depth-first search in a graph

Applications of Stacks

- Processing events in a last-in first-out order
- Simulating recursion
- Depth-first search in a graph
- Reverse a sequence

Applications of Stacks

- Processing events in a last-in first-out order
- Simulating recursion
- Depth-first search in a graph
- Reverse a sequence
- Matching brackets

Applications of Stacks

- Processing events in a last-in first-out order
- Simulating recursion
- Depth-first search in a graph
- Reverse a sequence
- Matching brackets
- And a lot more

Applications of Queues

- Processing events in a first-in first-out order

Applications of Queues

- Processing events in a first-in first-out order
- Breadth-first search in a graph

Applications of Queues

- Processing events in a first-in first-out order
- Breadth-first search in a graph
- And a lot more

Applications of Priority Queues

- Processing events in order of priority

Applications of Priority Queues

- Processing events in order of priority
- Finding a shortest path in a graph

Applications of Priority Queues

- Processing events in order of priority
- Finding a shortest path in a graph
- Some greedy algorithms

Applications of Priority Queues

- Processing events in order of priority
- Finding a shortest path in a graph
- Some greedy algorithms
- And a lot more

Applications of Sets

- Keep track of distinct items

Applications of Sets

- Keep track of distinct items
- Have we seen an item before?

Applications of Sets

- Keep track of distinct items
- Have we seen an item before?
- If implemented as a binary search tree:

Applications of Sets

- Keep track of distinct items
- Have we seen an item before?
- If implemented as a binary search tree:
    - Find the successor/predecessor of an element

Applications of Sets

- Keep track of distinct items
- Have we seen an item before?
- If implemented as a binary search tree:
  - Find the successor/predecessor of an element
  - Count how many elements are less than a given element

Applications of Sets

- Keep track of distinct items
- Have we seen an item before?
- If implemented as a binary search tree:
  - Find the successor/predecessor of an element
  - Count how many elements are less than a given element
  - Count how many elements are between two given elements

Applications of Sets

- Keep track of distinct items
- Have we seen an item before?
- If implemented as a binary search tree:
  - Find the successor/predecessor of an element
  - Count how many elements are less than a given element
  - Count how many elements are between two given elements
  - Find the kth largest element

Applications of Sets

- Keep track of distinct items
- Have we seen an item before?
- If implemented as a binary search tree:
  - Find the successor/predecessor of an element
  - Count how many elements are less than a given element
  - Count how many elements are between two given elements
  - Find the kth largest element
- And a lot more

Applications of Maps

- Associating a value with a key

Applications of Maps

- Associating a value with a key
- As a frequency table

Applications of Maps

- Associating a value with a key
- As a frequency table
- As a memory when doing Dynamic Programming

Applications of Maps

- Associating a value with a key
- As a frequency table
- As a memory when doing Dynamic Programming
- And a lot more

Augmenting Data Structures

- Sometimes we can store extra information in our data structures to gain more functionality

Augmenting Data Structures

- Sometimes we can store extra information in our data structures to gain more functionality
- Usually we can't do this to data structures in the standard library

Augmenting Data Structures

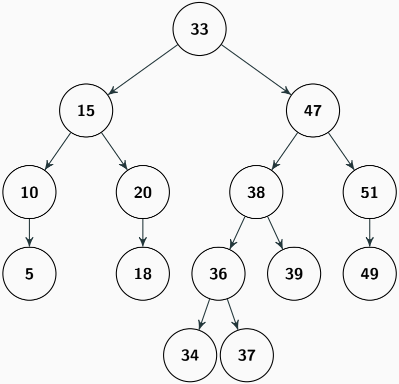
- Sometimes we can store extra information in our data structures to gain more functionality
- Usually we can't do this to data structures in the standard library
- Need our own implementation that we can customize

Augmenting Data Structures

- Sometimes we can store extra information in our data structures to gain more functionality
- Usually we can't do this to data structures in the standard library
- Need our own implementation that we can customize
- Example: Augmenting binary search trees

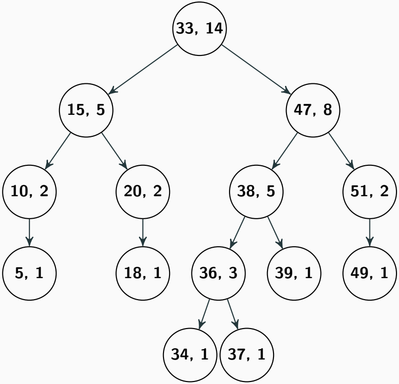
Augmenting Binary Search Trees

- We have a binary search tree and want to efficiently:
    - Count number of elements $< x$
    - Find the kth smallest element
- Naive method is to go through all vertices, but that is slow: $O(n)$



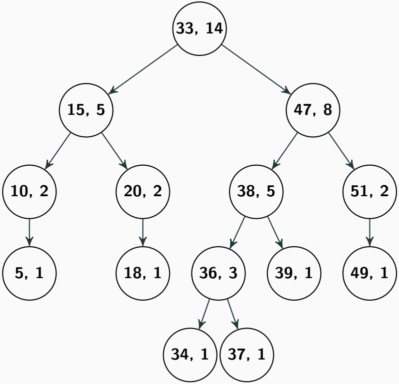
Augmenting Binary Search Trees

- Idea: In each vertex store the size of the subtree
- This information can be maintained when we insert/delete elements without increasing time complexity



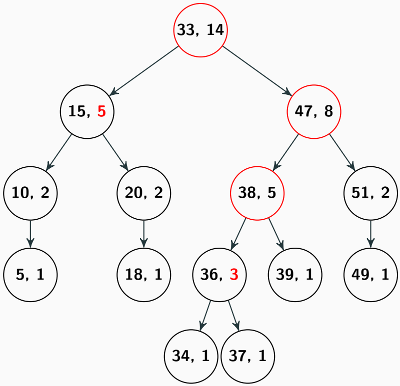
Augmenting Binary Search Trees

- Count number of elements < 38
  - Search for 38 in the tree
  - Count the vertices that we pass by that are less than x
  - When we are at a vertex where we should go right, get the size of the left subtree and add it to our count



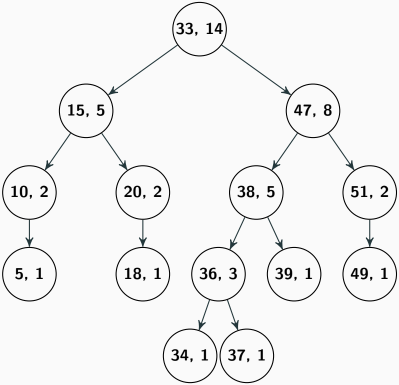
Augmenting Binary Search Trees

- Count number of elements < 38
  - Search for 38 in the tree
  - Count the vertices that we pass by that are less than x
  - When we are at a vertex where we should go right, get the size of the left subtree and add it to our count
- Time complexity $O(\log n)$



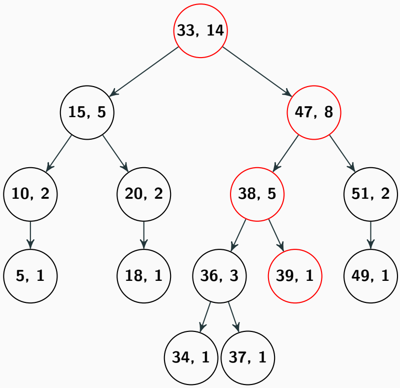
Augmenting Binary Search Trees

- Find kth smallest element
  - We're on a vertex whose left subtree is of size m
  - If $k = m + 1$, we found it
  - If $k \leq m$, look for the kth smallest element in the left subtree
  - If $k > m + 1$, look for the $k - m - 1$st smallest element in the right subtree



Augmenting Binary Search Trees

- Find kth smallest element
  - We're on a vertex whose left subtree is of size m
  - If $k = m + 1$, we found it
  - If $k \leq m$, look for the kth smallest element in the left subtree
  - If $k > m + 1$, look for the $k - m - 1$st smallest element in the right subtree
- Example: $k = 11$



Union-Find

- We have n items

Union-Find

- We have n items
- Maintains a collection of disjoint sets (or equivalently, an equivalence relation)

Union-Find

- We have n items
- Maintains a collection of disjoint sets (or equivalently, an equivalence relation)
- Each of the n items is in exactly one set

Union-Find

- We have n items
- Maintains a collection of disjoint sets (or equivalently, an equivalence relation)
- Each of the n items is in exactly one set
- We represent each set with one of its members, a representative element

Union-Find

- We have n items
- Maintains a collection of disjoint sets (or equivalently, an equivalence relation)
- Each of the n items is in exactly one set
- We represent each set with one of its members, a representative element
- Supports two operations efficiently: `find(x)` and `union(x,y)`.

Union-Find

- We have n items
- Maintains a collection of disjoint sets (or equivalently, an equivalence relation)
- Each of the n items is in exactly one set
- We represent each set with one of its members, a representative element
- Supports two operations efficiently: `find(x)` and `union(x,y)`.
- Operation `find(x)` finds the representative of the set x is in

Union-Find

- We have n items
- Maintains a collection of disjoint sets (or equivalently, an equivalence relation)
- Each of the n items is in exactly one set
- We represent each set with one of its members, a representative element
- Supports two operations efficiently: `find(x)` and `union(x,y)`.
- Operation `find(x)` finds the representative of the set x is in
- Operation `union(x, y)` unions the sets of which x and y are members.

Union-Find

- It is generally initialized with all items being in their own set.

Union-Find

- It is generally initialized with all items being in their own set.
- So for $n = 5$ we start out with $\{\{1\}, \{2\}, \{3\}, \{4\}, \{5\}\}$.

Union-Find

- It is generally initialized with all items being in their own set.
- So for $n = 5$ we start out with $\{\{1\}, \{2\}, \{3\}, \{4\}, \{5\}\}$.
- `join(1, 3)` then changes this to $\{\{1, 3\}, \{2\}, \{4\}, \{5\}\}$.

Union-Find

- It is generally initialized with all items being in their own set.
- So for $n = 5$ we start out with $\{\{1\}, \{2\}, \{3\}, \{4\}, \{5\}\}$.
- `join(1, 3)` then changes this to $\{\{1, 3\}, \{2\}, \{4\}, \{5\}\}$.
- `join(2, 5)` then results in $\{\{1, 3\}, \{2, 5\}, \{4\}\}$.

Union-Find

- It is generally initialized with all items being in their own set.
- So for $n = 5$ we start out with $\{\{1\}, \{2\}, \{3\}, \{4\}, \{5\}\}$.
- `join(1, 3)` then changes this to $\{\{1, 3\}, \{2\}, \{4\}, \{5\}\}$.
- `join(2, 5)` then results in $\{\{1, 3\}, \{2, 5\}, \{4\}\}$.
- `join(2, 4)` then results in $\{\{1, 3\}, \{2, 4, 5\}\}$.

Union-Find

- It is generally initialized with all items being in their own set.
- So for $n = 5$ we start out with $\{\{1\},\{2\},\{3\},\{4\},\{5\}\}$.
- `join(1, 3)` then changes this to $\{\{1,3\},\{2\},\{4\},\{5\}\}$.
- `join(2, 5)` then results in $\{\{1,3\},\{2,5\},\{4\}\}$.
- `join(2, 4)` then results in $\{\{1,3\},\{2,4,5\}\}$.
- `join(1, 4)` finally results in $\{\{1,2,3,4,5\}\}$.

Union-Find

- It is generally initialized with all items being in their own set.
- So for $n = 5$ we start out with $\{\{1\}, \{2\}, \{3\}, \{4\}, \{5\}\}$.
- `join(1, 3)` then changes this to $\{\{1, 3\}, \{2\}, \{4\}, \{5\}\}$.
- `join(2, 5)` then results in $\{\{1, 3\}, \{2, 5\}, \{4\}\}$.
- `join(2, 4)` then results in $\{\{1, 3\}, \{2, 4, 5\}\}$.
- `join(1, 4)` finally results in $\{\{1, 2, 3, 4, 5\}\}$.
- At any given point `find(x)` returns some value in the same set as x.

Union-Find

- It is generally initialized with all items being in their own set.
- So for $n = 5$ we start out with $\{\{1\}, \{2\}, \{3\}, \{4\}, \{5\}\}$.
- `join(1, 3)` then changes this to $\{\{1, 3\}, \{2\}, \{4\}, \{5\}\}$.
- `join(2, 5)` then results in $\{\{1, 3\}, \{2, 5\}, \{4\}\}$.
- `join(2, 4)` then results in $\{\{1, 3\}, \{2, 4, 5\}\}$.
- `join(1, 4)` finally results in $\{\{1, 2, 3, 4, 5\}\}$.
- At any given point `find(x)` returns some value in the same set as x.
- The important bit is that `find(x)` returns the same value for all elements of the same set, the representative.

Union-Find

- We can do this by maintaining an array of parents, letting the i-th value be the index of the parent of the i-th item.

Union-Find

- We can do this by maintaining an array of parents, letting the i-th value be the index of the parent of the i-th item.
- If a value has no parent, we can denote this somehow, make it its own parent, give it the value -1, exactly what we do is not important.

Union-Find

- We can do this by maintaining an array of parents, letting the i-th value be the index of the parent of the i-th item.
- If a value has no parent, we can denote this somehow, make it its own parent, give it the value -1, exactly what we do is not important.
- To get the representative of x we go to the parent of our current item (starting at x) until the item has no parent.

Union-Find

- We can do this by maintaining an array of parents, letting the i-th value be the index of the parent of the i-th item.
- If a value has no parent, we can denote this somehow, make it its own parent, give it the value -1, exactly what we do is not important.
- To get the representative of x we go to the parent of our current item (starting at x) until the item has no parent.
- Then to unite x, y we simply make the representative of x the parent of the representative of y.

Naïve Union-Find implementation

```
struct union_find {
    vector<int> parent;
    union_find(int n) {
        parent = vector<int>(n);
        for(int i = 0; i < n; i++) {
            parent[i] = i;
        }
    }
    int find(int x) {
        return parent[x] == x ? x : find(parent[x]);
    }
    void unite(int x, int y) {
        parent[find(x)] = find(y);
    }
};
```

Union-Find

- The problem here is that for specific queries, the parent chains may end up being of length $\mathcal{O}(n)$, making each query linear.

Union-Find

- The problem here is that for specific queries, the parent chains may end up being of length $\mathcal{O}(n)$, making each query linear.
- The key to making this more efficient is making those chains shorter.

Union-Find

- The problem here is that for specific queries, the parent chains may end up being of length $\mathcal{O}(n)$, making each query linear.
- The key to making this more efficient is making those chains shorter.
- We do this by flattening the chain each time we query `find`, so the amortized complexity becomes good.

Union-Find

- The problem here is that for specific queries, the parent chains may end up being of length $\mathcal{O}(n)$, making each query linear.
- The key to making this more efficient is making those chains shorter.
- We do this by flattening the chain each time we query `find`, so the amortized complexity becomes good.
- The worst case is still $\mathcal{O}(n)$ but the amortized complexity is $\mathcal{O}(\alpha(n))$ which may as well be a constant, as it is < 5 for n equal to the number of atoms in the observable universe.

Path compressed Union-Find implementation

```
struct union_find {
    vector<int> parent;
    union_find(int n) {
        parent = vector<int>(n);
        for (int i = 0; i < n; i++) {
            parent[i] = i;
        }
    }
    int find(int x) {
        if(parent[x] == x) return x;
        return parent[x] = find(parent[x]);
    }
    void unite(int x, int y) {
        parent[find(x)] = find(y);
    }
};
```

Union-Find applications

- Union-Find maintains a collection of disjoint sets

Union-Find applications

- Union-Find maintains a collection of disjoint sets
- When are we dealing with such collections?

Union-Find applications

- Union-Find maintains a collection of disjoint sets
- When are we dealing with such collections?
- Usually when we want to work with equivalence relations like graph connectivity

Union-Find applications

- Union-Find maintains a collection of disjoint sets
- When are we dealing with such collections?
- Usually when we want to work with equivalence relations like graph connectivity
- By modifying the data structure it can also contain more queryable data

Union-Find applications

- Union-Find maintains a collection of disjoint sets
- When are we dealing with such collections?
- Usually when we want to work with equivalence relations like graph connectivity
- By modifying the data structure it can also contain more queryable data
    - Number of different sets currently

Union-Find applications

- Union-Find maintains a collection of disjoint sets
- When are we dealing with such collections?
- Usually when we want to work with equivalence relations like graph connectivity
- By modifying the data structure it can also contain more queryable data
  - Number of different sets currently
  - Current size of the set containing x

Union-Find applications

- Union-Find maintains a collection of disjoint sets
- When are we dealing with such collections?
- Usually when we want to work with equivalence relations like graph connectivity
- By modifying the data structure it can also contain more queryable data
  - Number of different sets currently
  - Current size of the set containing x
  - An iterable list of all elements of the set containing x

Union-Find applications

- Union-Find maintains a collection of disjoint sets
- When are we dealing with such collections?
- Usually when we want to work with equivalence relations like graph connectivity
- By modifying the data structure it can also contain more queryable data
  - Number of different sets currently
  - Current size of the set containing x
  - An iterable list of all elements of the set containing x
- When tracking size you can use it to always perform small-to-large merges for $\mathcal{O}(\log n)$ time complexity.

Example problem: Skolavslutningen

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

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:
  - $\max(A[i], A[i+1], \ldots, A[j-1], A[j])$
  - $\min(A[i], A[i+1], \ldots, A[j-1], A[j])$
  - $\text{sum}(A[i], A[i+1], \ldots, A[j-1], A[j])$
- We want to answer these queries efficiently, or in other words, without looking through all elements.

Range queries

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

Range sum on a static array

- Let's look at range sums on a constant array

Range sum on a static array

- Let's look at range sums on a constant array
- How do we support these queries efficiently?

Range sum on a static array

- Let's look at range sums on a constant array
- How do we support these queries efficiently?
- Simplification: only support queries of the form $\mathrm{sum}(0, j)$

Range sum on a static array

- Let's look at range sums on a constant array
- How do we support these queries efficiently?
- Simplification: only support queries of the form $\mathrm{sum}(0, j)$
- Notice that $\mathrm{sum}(i, j) = \mathrm{sum}(0, j) - \mathrm{sum}(0, i-1)$

Range sum on a static array

- So we're only interested in prefix sums

Range sum on a static array

- So we're only interested in prefix sums
- But there are only n of them...

Range sum on a static array

- So we're only interested in prefix sums
- But there are only n of them...
- Just compute them all once in the beginning

Range sum on a static array

- So we're only interested in prefix sums
- But there are only n of them...
- Just compute them all once in the beginning

1	0	7	8	5	9	3

Range sum on a static array

- So we're only interested in prefix sums
- But there are only n of them...
- Just compute them all once in the beginning

1	0	7	8	5	9	3
1						

Range sum on a static array

- So we're only interested in prefix sums
- But there are only n of them...
- Just compute them all once in the beginning

1	0	7	8	5	9	3
1	1					

Range sum on a static array

- So we're only interested in prefix sums
- But there are only n of them...
- Just compute them all once in the beginning

1	0	7	8	5	9	3
1	1	8				

Range sum on a static array

- So we're only interested in prefix sums
- But there are only n of them...
- Just compute them all once in the beginning

1	0	7	8	5	9	3
1	1	8	16			

Range sum on a static array

- So we're only interested in prefix sums
- But there are only n of them...
- Just compute them all once in the beginning

1	0	7	8	5	9	3
1	1	8	16	21		

Range sum on a static array

- So we're only interested in prefix sums
- But there are only n of them...
- Just compute them all once in the beginning

1	0	7	8	5	9	3
1	1	8	16	21	30	

Range sum on a static array

- So we're only interested in prefix sums
- But there are only n of them...
- Just compute them all once in the beginning

1	0	7	8	5	9	3
1	1	8	16	21	30	33

Range sum on a static array

- So we're only interested in prefix sums
- But there are only n of them...
- Just compute them all once in the beginning

1	0	7	8	5	9	3
1	1	8	16	21	30	33

- $O(n)$ time to preprocess

Range sum on a static array

- So we're only interested in prefix sums
- But there are only n of them...
- Just compute them all once in the beginning

1	0	7	8	5	9	3
1	1	8	16	21	30	33

- $O(n)$ time to preprocess
- $O(1)$ time each query

Range sum on a static array

- So we're only interested in prefix sums
- But there are only n of them...
- Just compute them all once in the beginning

1	0	7	8	5	9	3
1	1	8	16	21	30	33

- $O(n)$ time to preprocess
- $O(1)$ time each query

- Can we support updating efficiently?

Range sum on a static array

- So we're only interested in prefix sums
- But there are only n of them...
- Just compute them all once in the beginning

1	0	7	8	5	9	3
1	1	8	16	21	30	33

- $O(n)$ time to preprocess
- $O(1)$ time each query

- Can we support updating efficiently? No, at least not without modification

Generalizing

- This works on any invertible function.

Generalizing

- This works on any invertible function.
- If we want the product we can store the products and use $\mathrm{mul}(i, j) = \mathrm{mul}(0, j)/\mathrm{mul}(0, i-1)$.

Generalizing

- This works on any invertible function.
- If we want the product we can store the products and use $\mathrm{mul}(i, j) = \mathrm{mul}(0, j)/\mathrm{mul}(0, i - 1)$.
- This also works for multidimensional arrays, but the math is more involved.

Generalizing

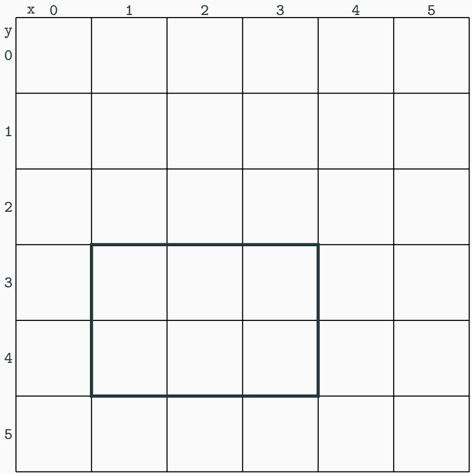
- This works on any invertible function.
- If we want the product we can store the products and use $\mathrm{mul}(i, j) = \mathrm{mul}(0, j)/\mathrm{mul}(0, i-1)$.
- This also works for multidimensional arrays, but the math is more involved.
- We let $\mathrm{sum}(x_i, x_j, y_i, y_j)$ denote the query for the sum from x_i to x_j along the x-dimension, and the same for y.

Generalizing

- This works on any invertible function.
- If we want the product we can store the products and use $\mathrm{mul}(i, j) = \mathrm{mul}(0, j)/\mathrm{mul}(0, i-1)$.
- This also works for multidimensional arrays, but the math is more involved.
- We let $\mathrm{sum}(x_i, x_j, y_i, y_j)$ denote the query for the sum from x_i to x_j along the x-dimension, and the same for y.
- Then the formula becomes

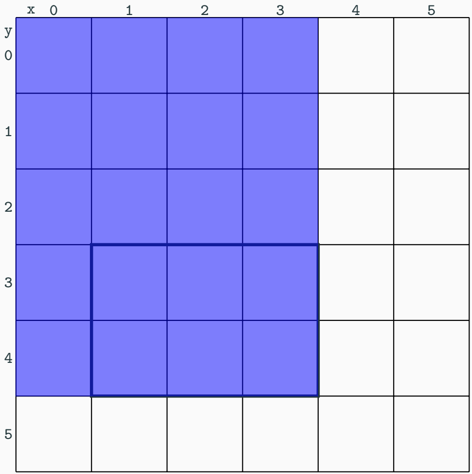
$$\begin{aligned}\mathrm{sum}(x_i, x_j, y_i, y_j) &= \mathrm{sum}(0, x_j, 0, y_j) \\ &- \mathrm{sum}(0, x_{i-1}, 0, y_j) \\ &- \mathrm{sum}(0, x_j, 0, y_{i-1}) \\ &+ \mathrm{sum}(0, x_{i-1}, 0, y_{i-1})\end{aligned}$$

2D sum



`query(1, 3, 3, 4)`

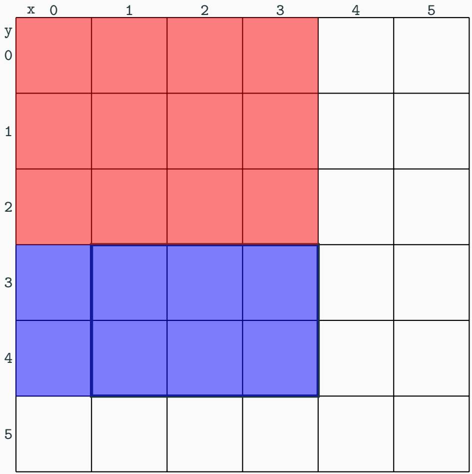
2D sum



`query(1, 3, 3, 4)`

`query(0, 3, 0, 4)`

2D sum

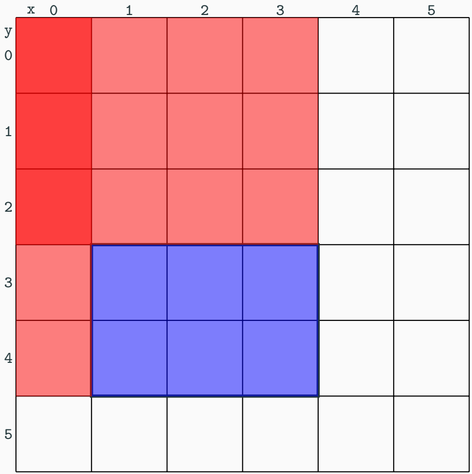


`query(1, 3, 3, 4)`

`query(0, 3, 0, 4)`

`query(0, 4, 0, 2)`

2D sum



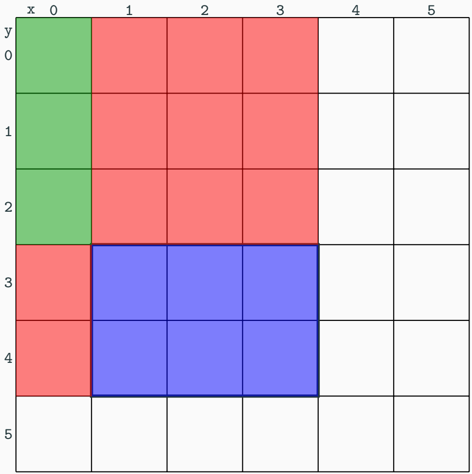
query(1, 3, 3, 4)

query(0, 3, 0, 4)

query(0, 4, 0, 2)

query(0, 0, 0, 4)

2D sum




```
query(1, 3, 3, 4)

query(0, 3, 0, 4)

query(0, 4, 0, 2)

query(0, 0, 0, 4)

query(0, 0, 0, 2)
```

Range sum on a mutable array

- What if we want to support:

Range sum on a mutable array

- What if we want to support:
    - sum over a range

Range sum on a mutable array

- What if we want to support:
    - sum over a range
    - updating an element

Range sum on a mutable array

- What if we want to support:
  - sum over a range
  - updating an element
- How do we support these queries efficiently?

First attempt: Buckets

- Group values into buckets of size k and store result of each bucket

First attempt: Buckets

- Group values into buckets of size k and store result of each bucket
- Updating is easy:

First attempt: Buckets

- Group values into buckets of size k and store result of each bucket
- Updating is easy:
    - change the array element

First attempt: Buckets

- Group values into buckets of size k and store result of each bucket
- Updating is easy:
  - change the array element
  - recompute corresponding bucket

First attempt: Buckets

- Group values into buckets of size k and store result of each bucket
- Updating is easy:
  - change the array element
  - recompute corresponding bucket
- Time complexity: $O(k)$

First attempt: Buckets

- Group values into buckets of size k and store result of each bucket
- Updating is easy:
    - change the array element
    - recompute corresponding bucket
- Time complexity: $O(k)$
- Again we want to query over a range

First attempt: Buckets

- Group values into buckets of size k and store result of each bucket
- Updating is easy:
    - change the array element
    - recompute corresponding bucket
- Time complexity: $O(k)$
- Again we want to query over a range
    - When a bucket is contained in the range, use the stored sum for the bucket

First attempt: Buckets

- Group values into buckets of size k and store result of each bucket
- Updating is easy:
  - change the array element
  - recompute corresponding bucket
- Time complexity: $O(k)$
- Again we want to query over a range
  - When a bucket is contained in the range, use the stored sum for the bucket
  - This (sometimes) allows us to “jump” over intervals of size k

First attempt: Buckets

- Group values into buckets of size k and store result of each bucket
- Updating is easy:
  - change the array element
  - recompute corresponding bucket
- Time complexity: $O(k)$
- Again we want to query over a range
  - When a bucket is contained in the range, use the stored sum for the bucket
  - This (sometimes) allows us to "jump" over intervals of size k
  - Only have to go inside at most two buckets (each end)

First attempt: Buckets

- Group values into buckets of size k and store result of each bucket
- Updating is easy:
  - change the array element
  - recompute corresponding bucket
- Time complexity: $O(k)$
- Again we want to query over a range
  - When a bucket is contained in the range, use the stored sum for the bucket
  - This (sometimes) allows us to "jump" over intervals of size k
  - Only have to go inside at most two buckets (each end)
  - Have to consider at most n/k buckets and 2 buckets of size k

First attempt: Buckets

- Group values into buckets of size k and store result of each bucket
- Updating is easy:
  - change the array element
  - recompute corresponding bucket
- Time complexity: $O(k)$
- Again we want to query over a range
  - When a bucket is contained in the range, use the stored sum for the bucket
  - This (sometimes) allows us to "jump" over intervals of size k
  - Only have to go inside at most two buckets (each end)
  - Have to consider at most n/k buckets and 2 buckets of size k
- Time complexity: $O(n/k + k)$

Buckets: Choosing k

- Now we have a data structure that supports:
  - Querying in $O(n/k + k)$

Buckets: Choosing k

- Now we have a data structure that supports:
  - Updating in $O(k)$
  - Querying in $O(n/k + k)$

Buckets: Choosing k

- Now we have a data structure that supports:
  - Updating in $O(k)$
  - Querying in $O(n/k + k)$
- What k to pick?

Buckets: Choosing k

- Now we have a data structure that supports:
    - Updating in $O(k)$
    - Querying in $O(n/k + k)$
- What k to pick?
- Time complexity is minimized for $k = \sqrt{n}$:

Buckets: Choosing k

- Now we have a data structure that supports:
    - Updating in $O(k)$
    - Querying in $O(n/k + k)$
- What k to pick?
- Time complexity is minimized for $k = \sqrt{n}$:
    - Updating in $O(\sqrt{n})$

Buckets: Choosing k

- Now we have a data structure that supports:
  - Updating in $O(k)$
  - Querying in $O(n/k + k)$
- What k to pick?
- Time complexity is minimized for $k = \sqrt{n}$:
  - Updating in $O(\sqrt{n})$
  - Querying in $O(n/\sqrt{n} + \sqrt{n}) = O(\sqrt{n})$

Buckets: Choosing k

- Now we have a data structure that supports:
  - Updating in $O(k)$
  - Querying in $O(n/k + k)$
- What k to pick?
- Time complexity is minimized for $k = \sqrt{n}$:
  - Updating in $O(\sqrt{n})$
  - Querying in $O(n/\sqrt{n} + \sqrt{n}) = O(\sqrt{n})$
- Also known as square root decomposition, and is a very powerful technique

Example problem: Supercomputer

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

Range queries

- Now we know how to do these queries in $O(\sqrt{n})$

Range queries

- Now we know how to do these queries in $O(\sqrt{n})$
- May be too slow if n is large and many queries

Range queries

- Now we know how to do these queries in $O(\sqrt{n})$
- May be too slow if n is large and many queries
- Can we do better?

Second attempt: Segment Tree

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

Second attempt: 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.

Second attempt: 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.

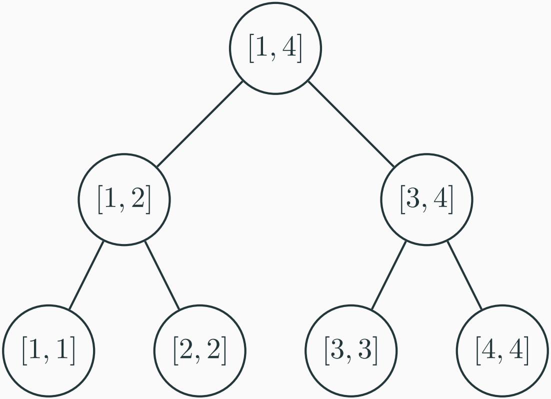
Second attempt: Segment Tree

- 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.

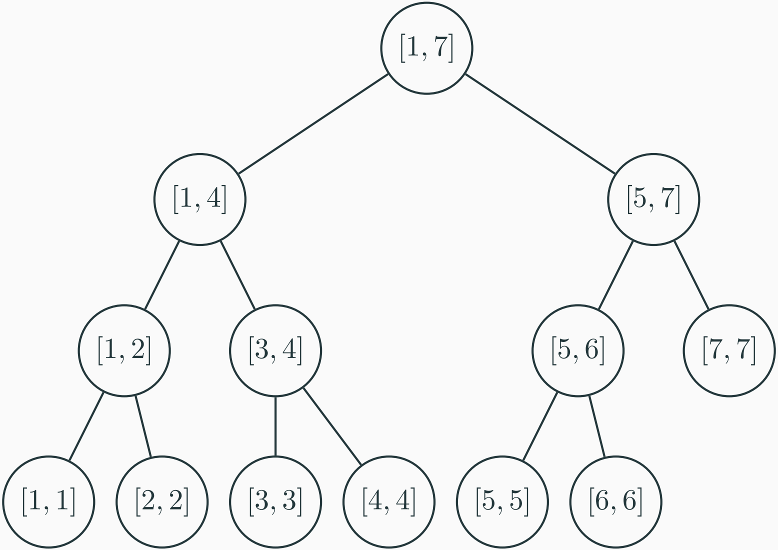
Second attempt: Segment Tree

- 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, $n = 4$



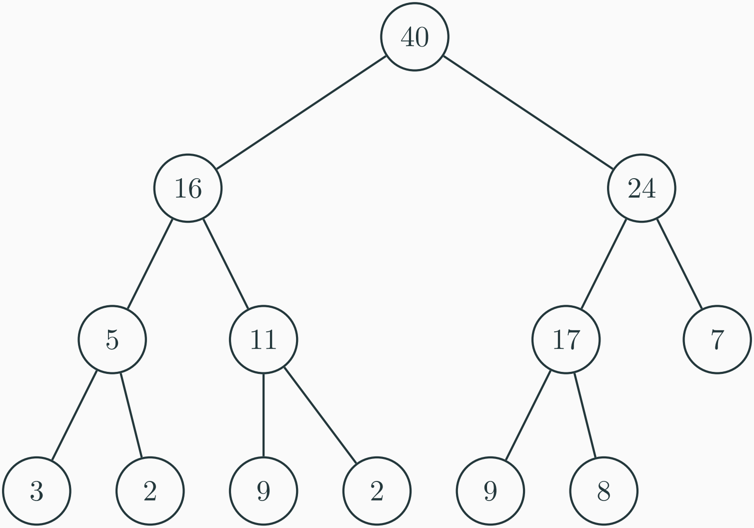
Drawn Segment Tree, $n = 7$



Segment Tree - Code

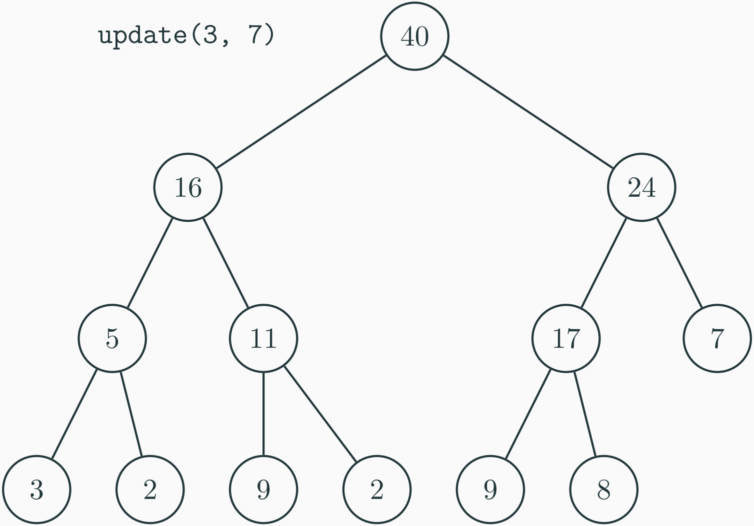
```
struct segment_tree {
    segment_tree *left, *right;
    int from, to, value;
    segment_tree(int from, int to)
        : from(from), to(to), left(NULL), right(NULL), value(0) { }
};

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



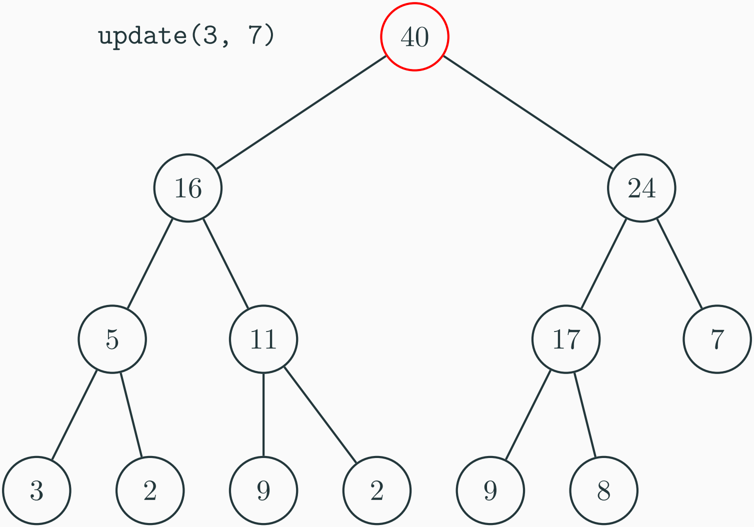
Updates

`update(3, 7)`

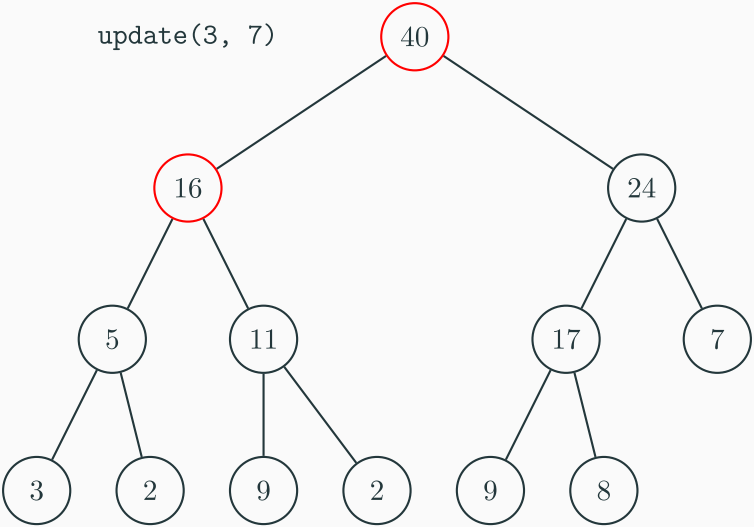


Updates

`update(3, 7)`

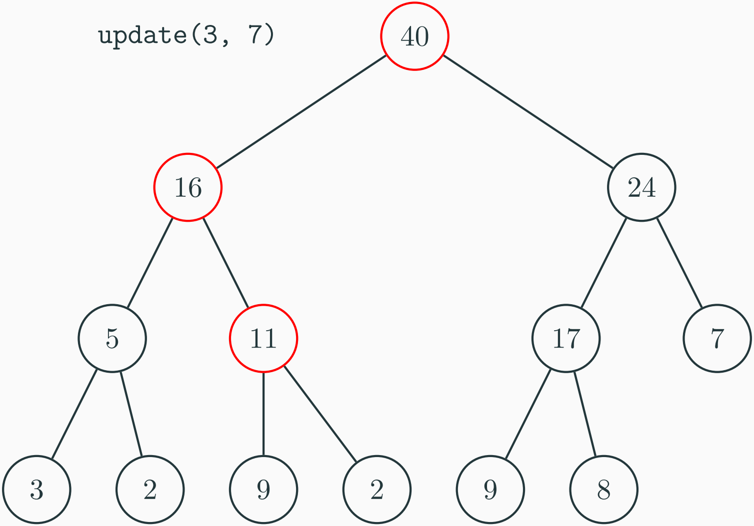


Updates

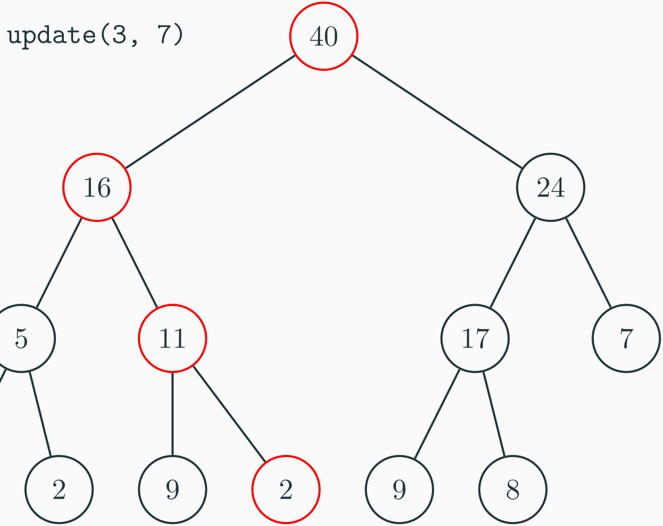


Updates

`update(3, 7)`

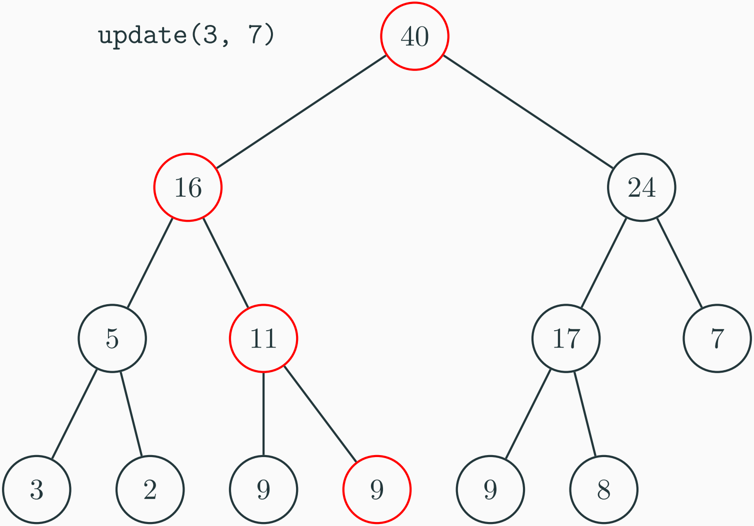


Updates

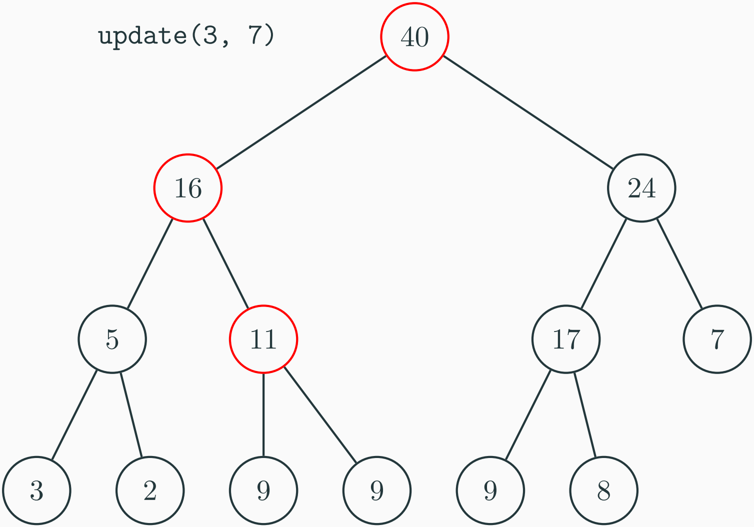


Updates

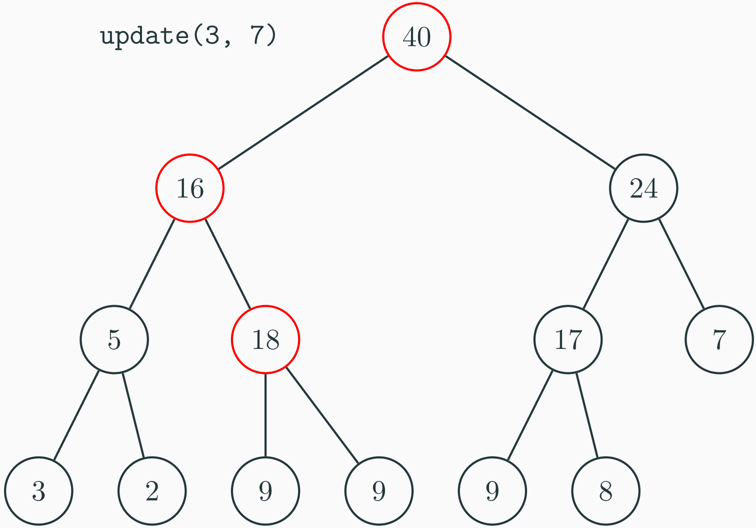
`update(3, 7)`



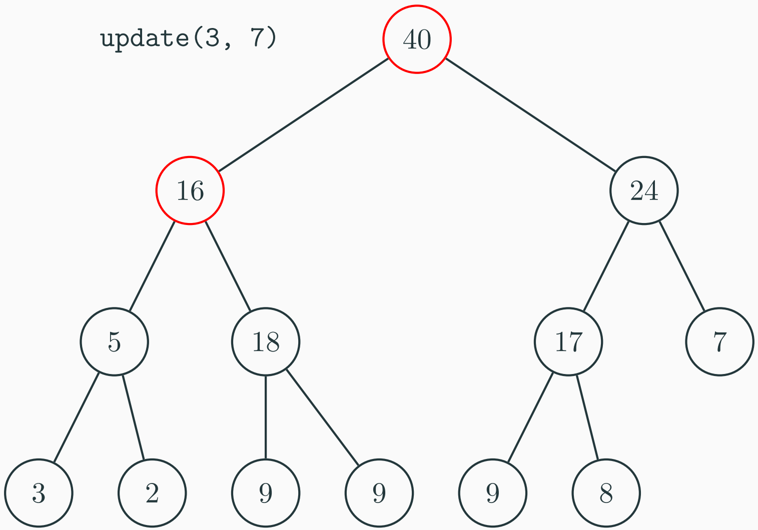
Updates



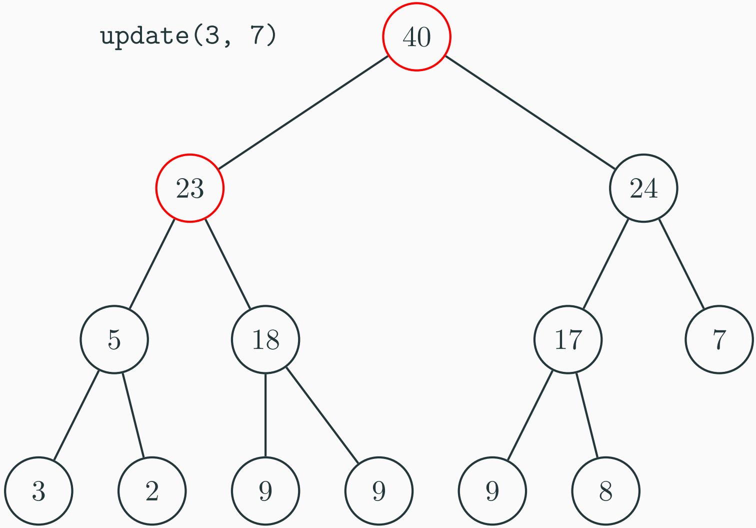
Updates



Updates

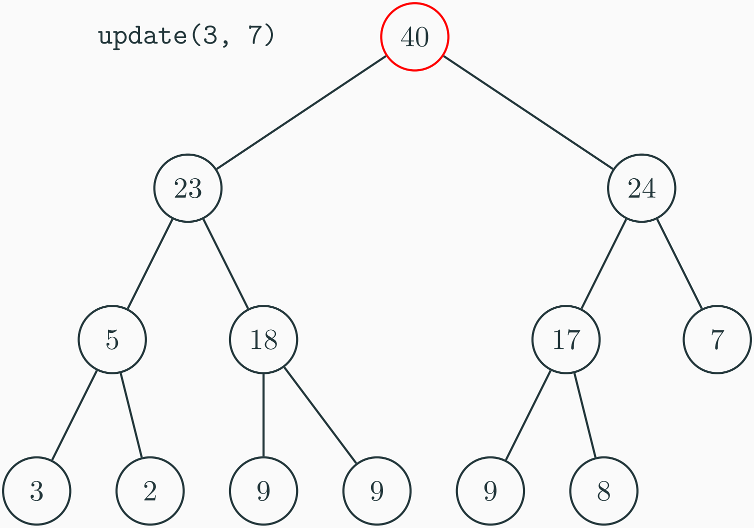


Updates



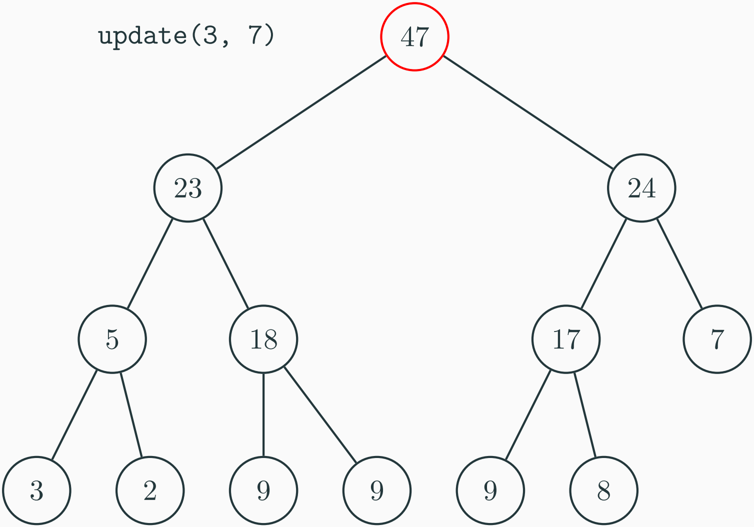
Updates

`update(3, 7)`

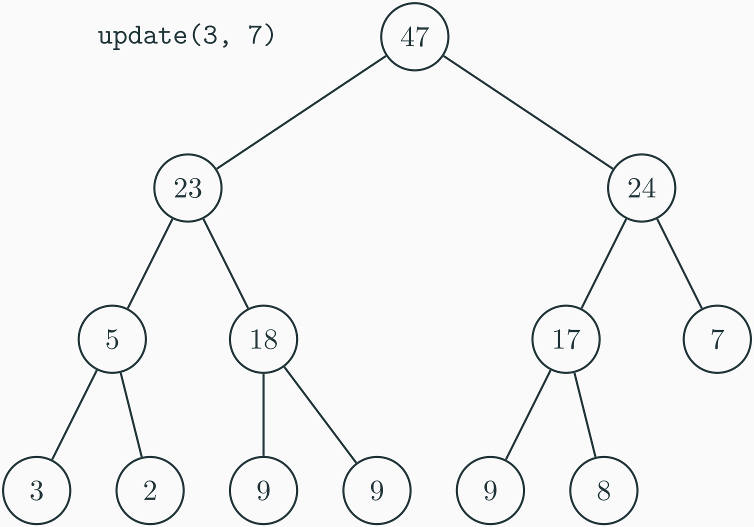


Updates

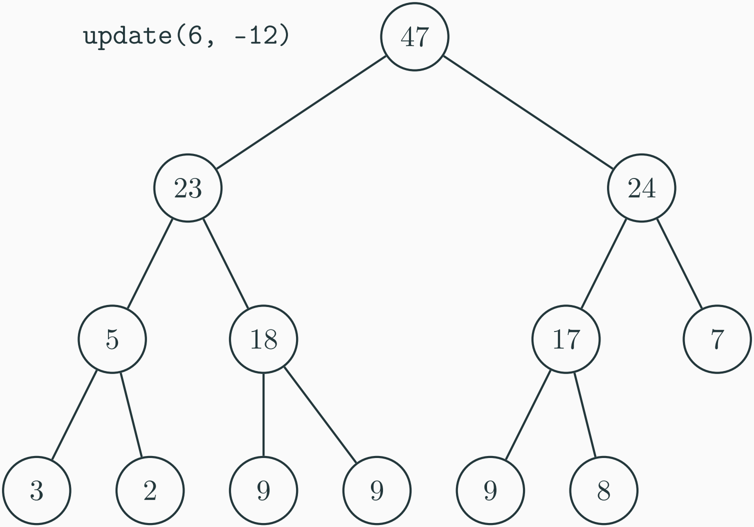
`update(3, 7)`



Updates

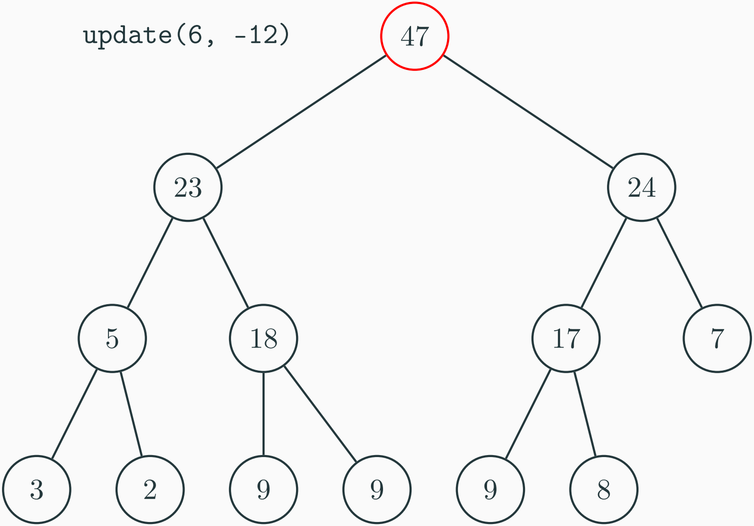


Updates



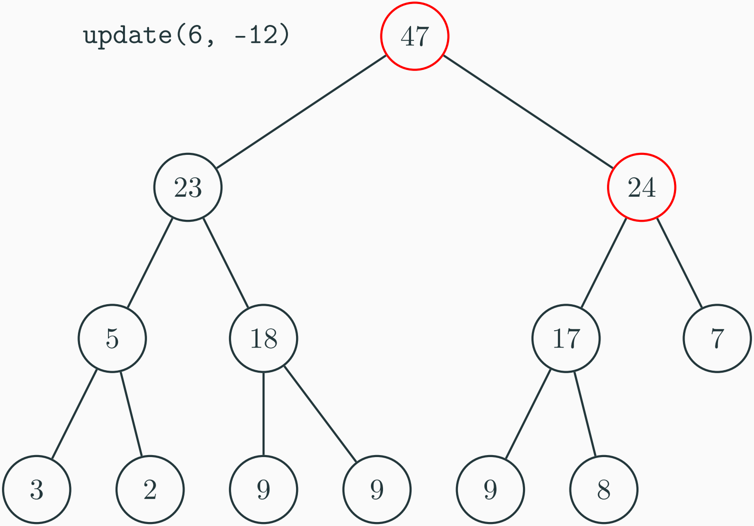
Updates

`update(6, -12)`



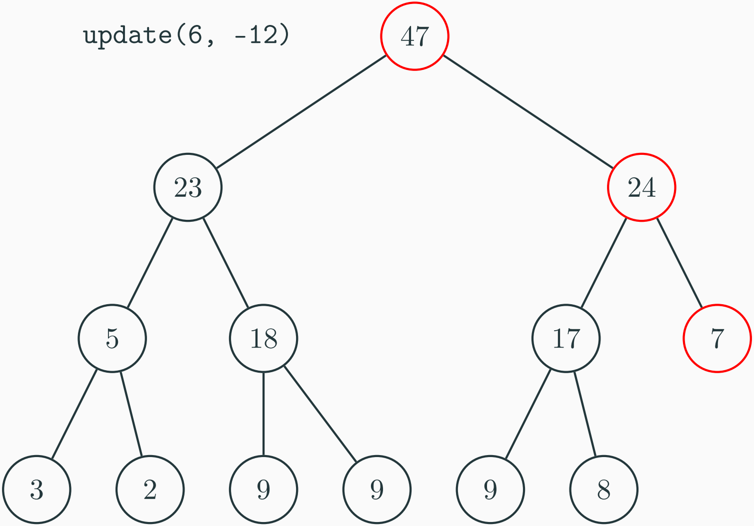
Updates

update(6, -12)



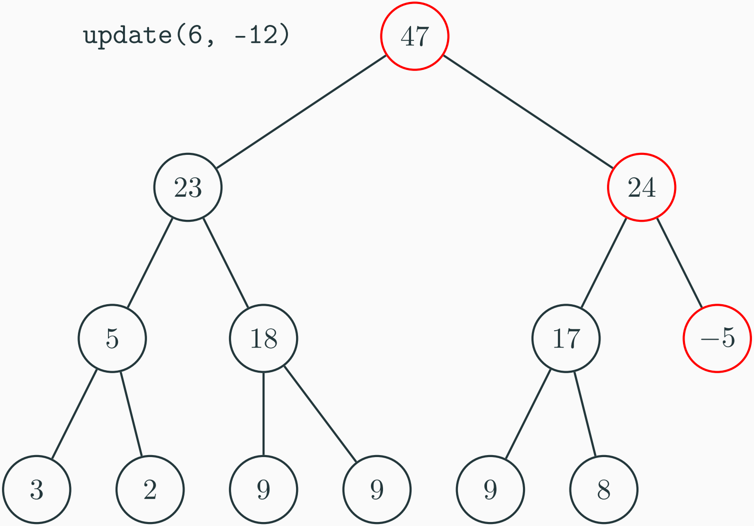
Updates

`update(6, -12)`

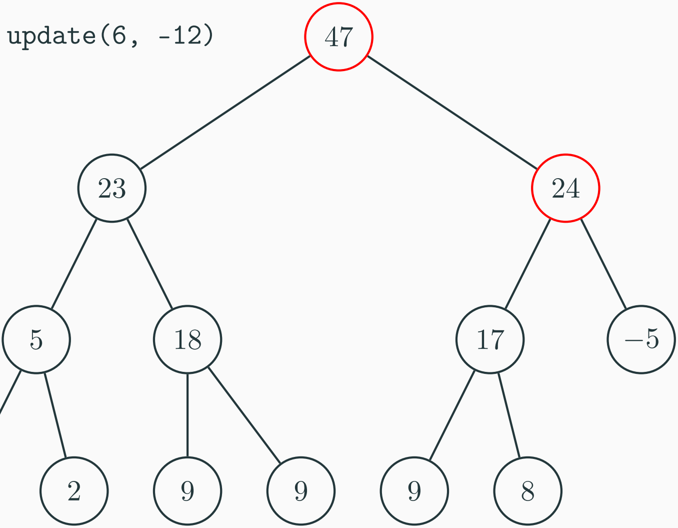


Updates

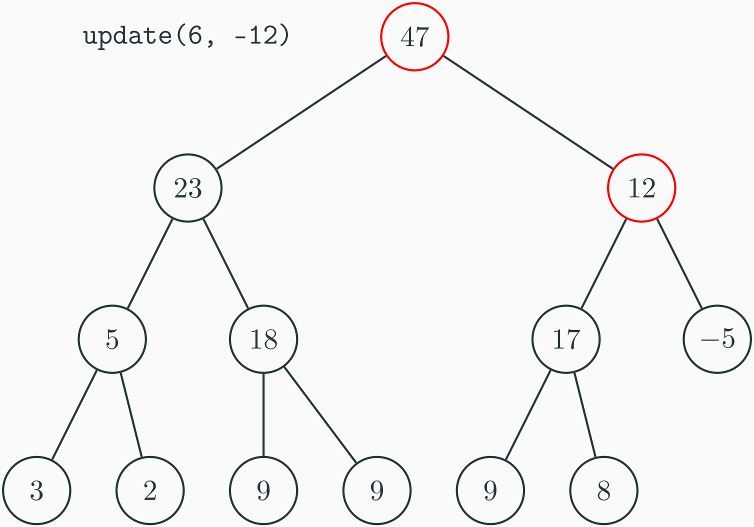
`update(6, -12)`



Updates

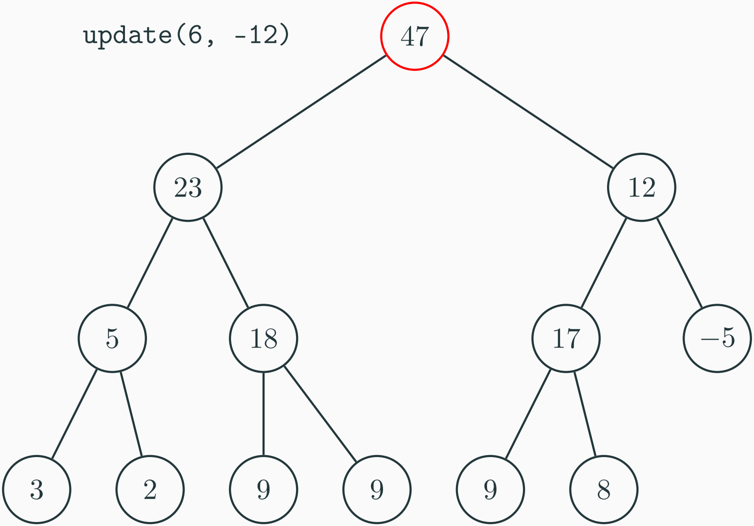


Updates

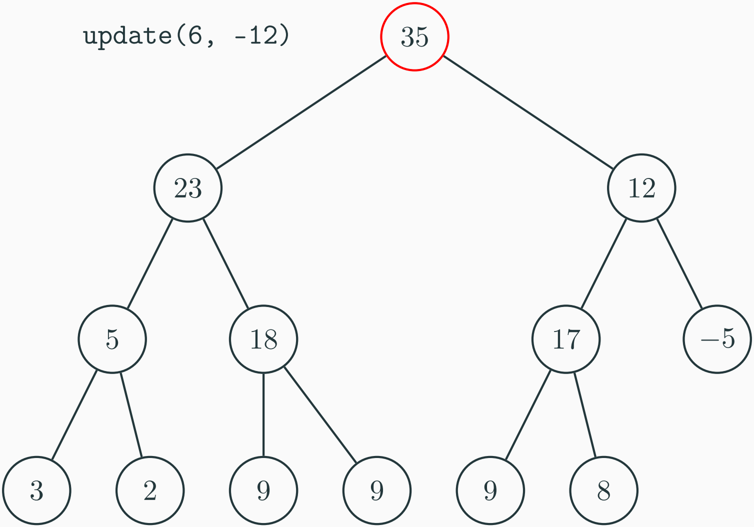


Updates

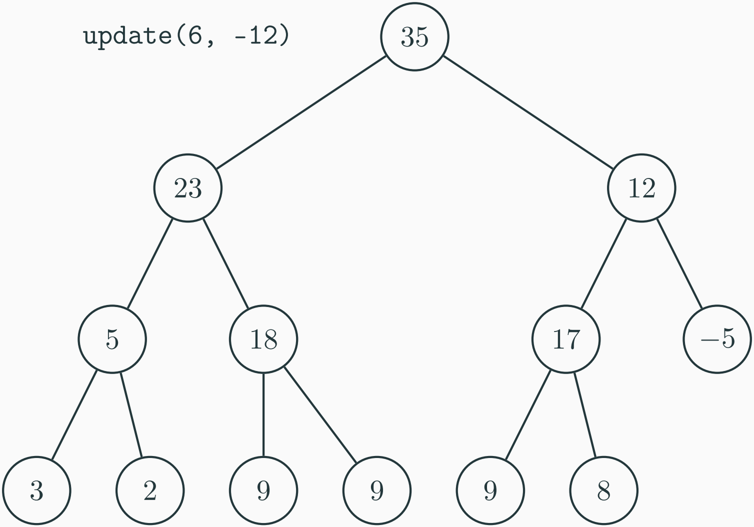
`update(6, -12)`



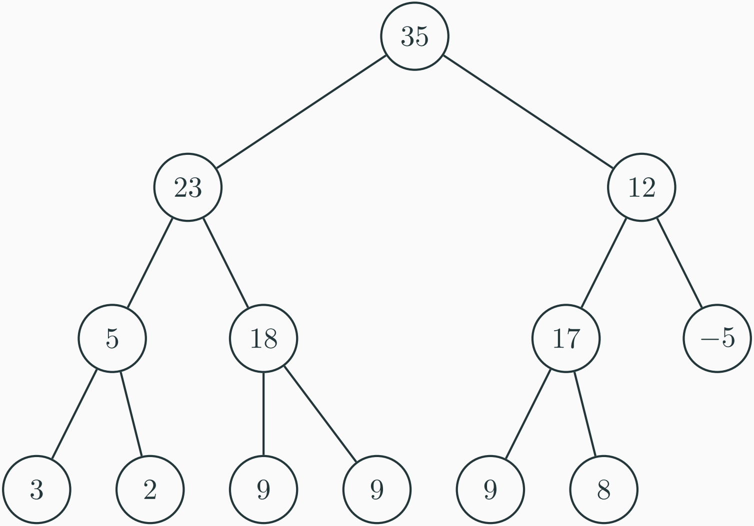
Updates



Updates



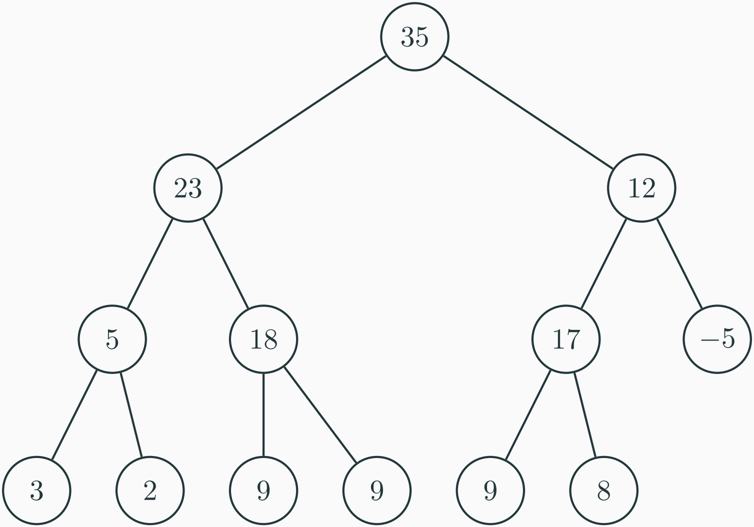
Updates



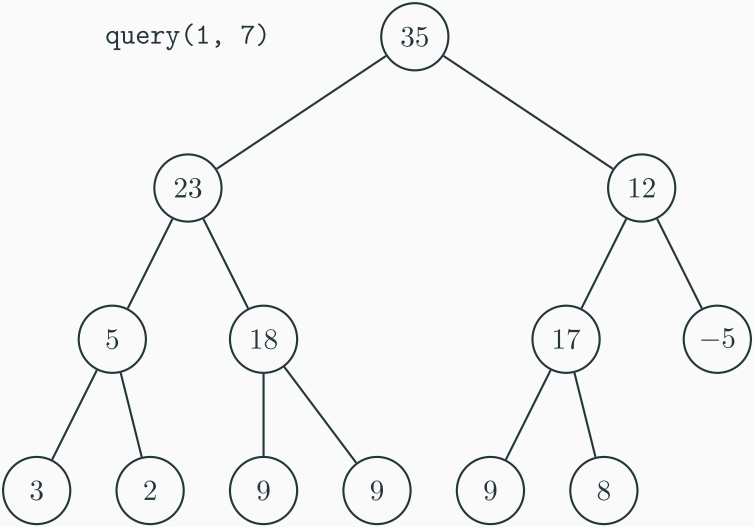
Updating a Segment Tree - Code

```
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

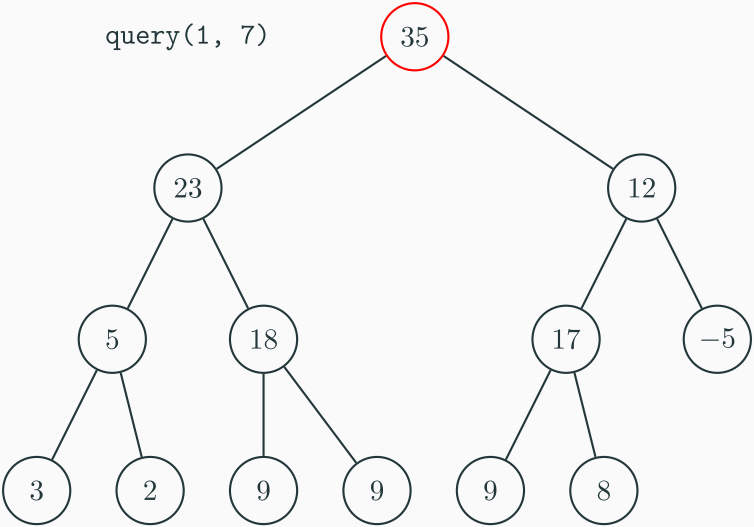


Querying

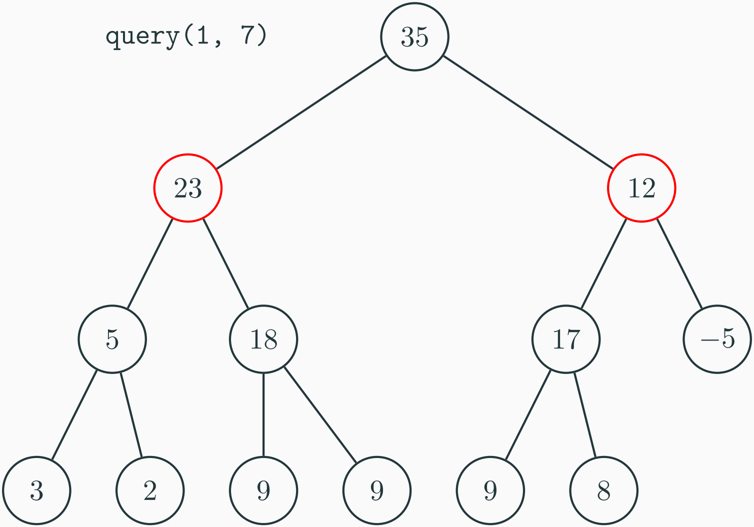


Querying

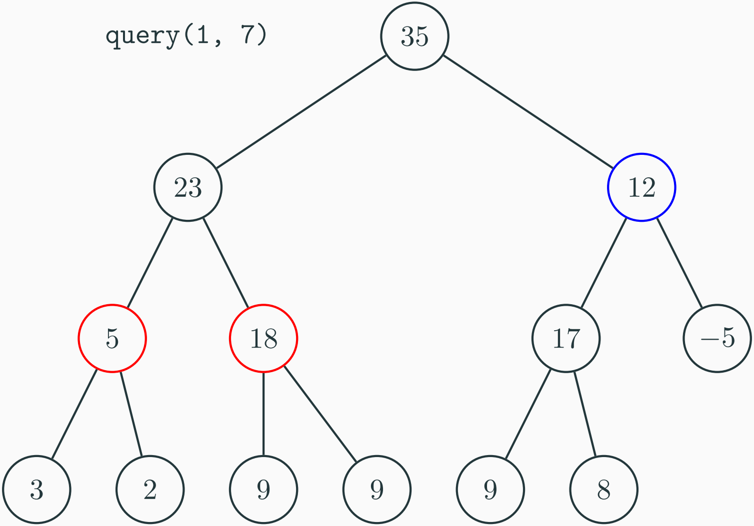
`query(1, 7)`



Querying

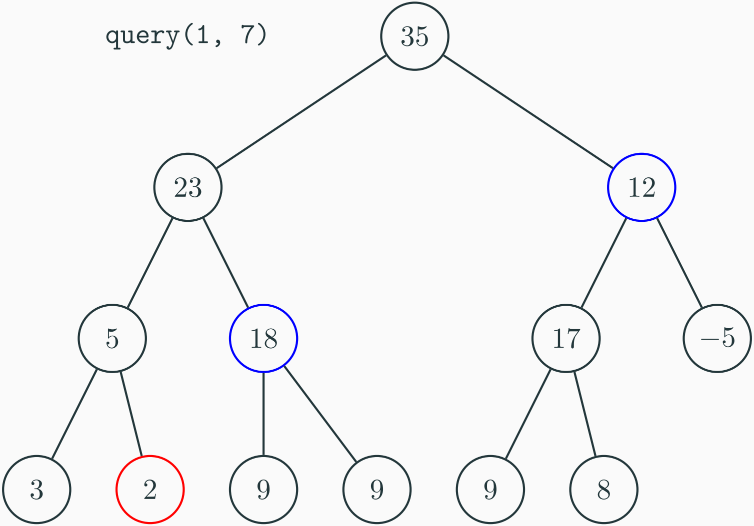


Querying

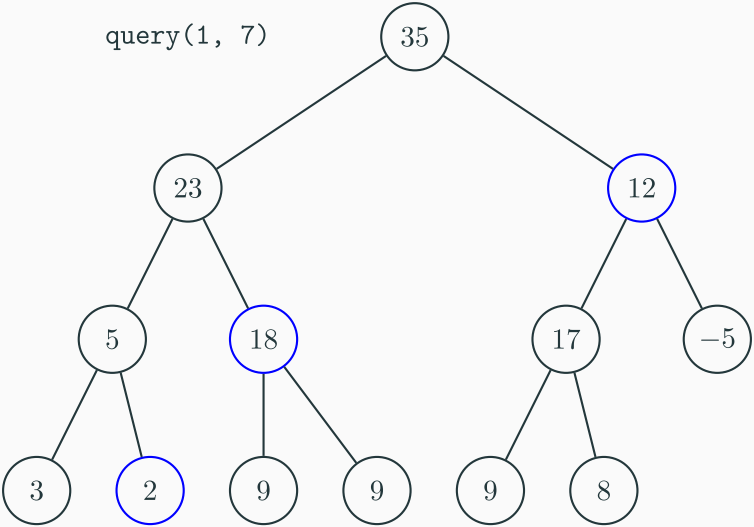


Querying

`query(1, 7)`

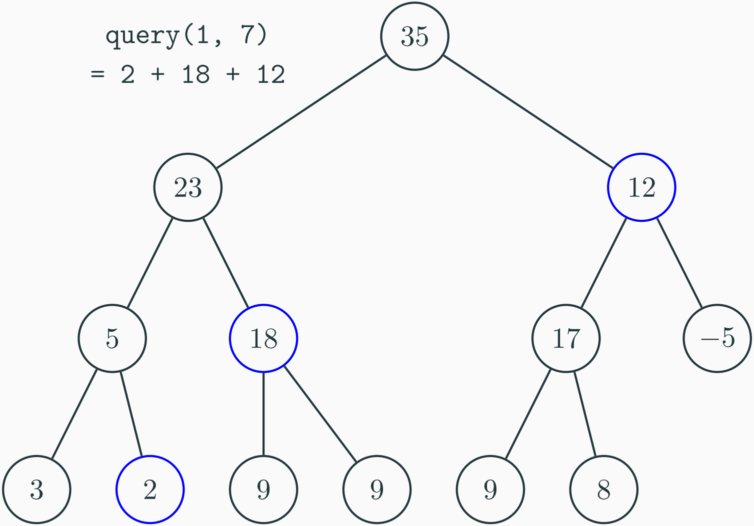


Querying

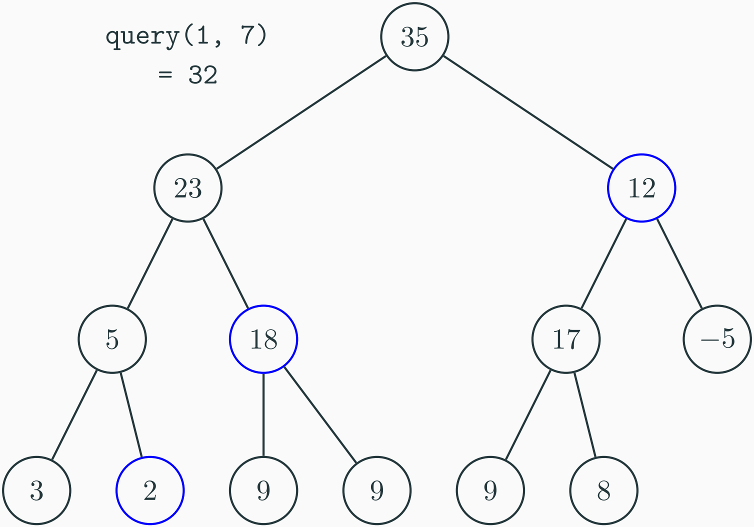


Querying

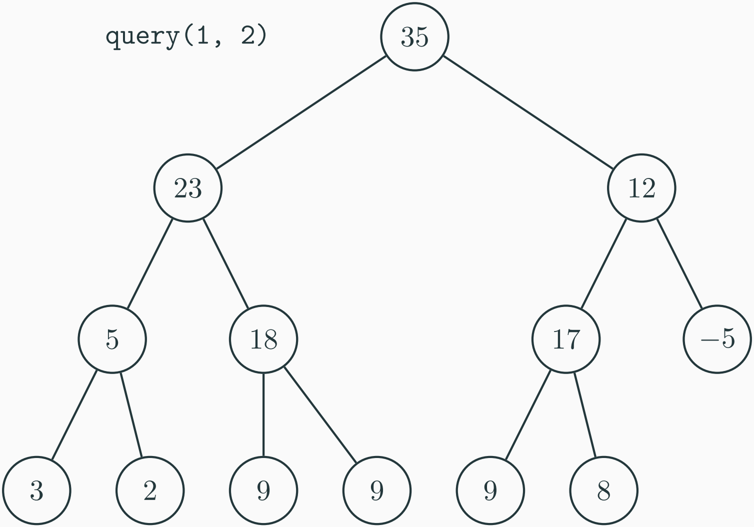
```
query(1, 7)
= 2 + 18 + 12
```



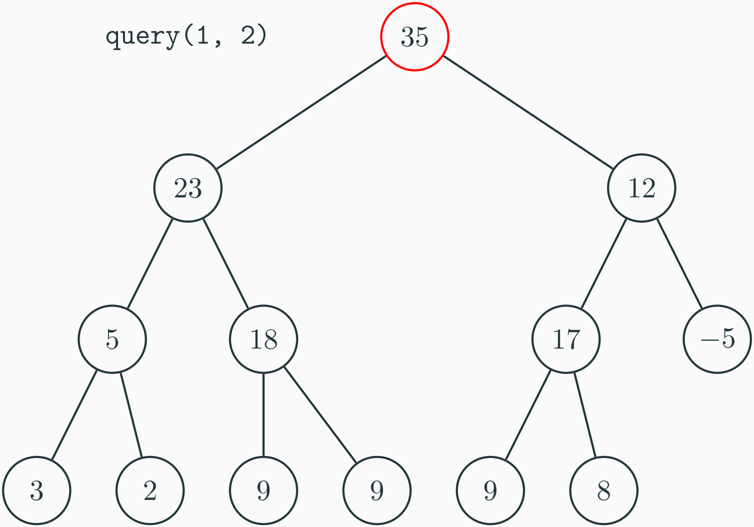
Querying



Querying

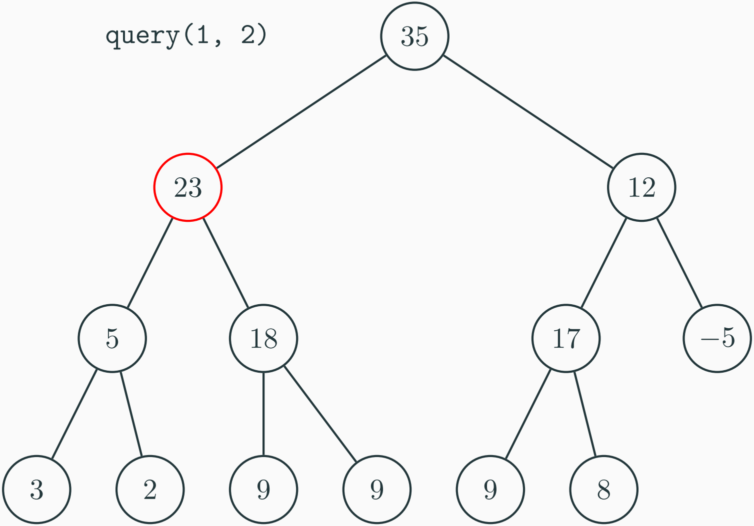


Querying

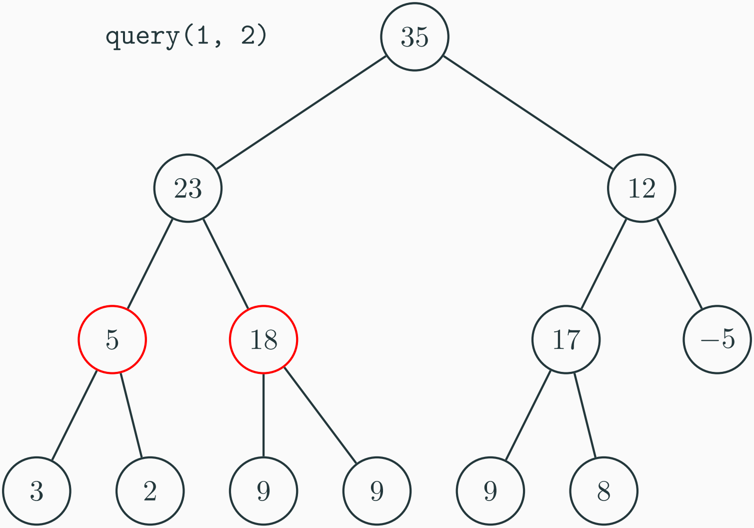


Querying

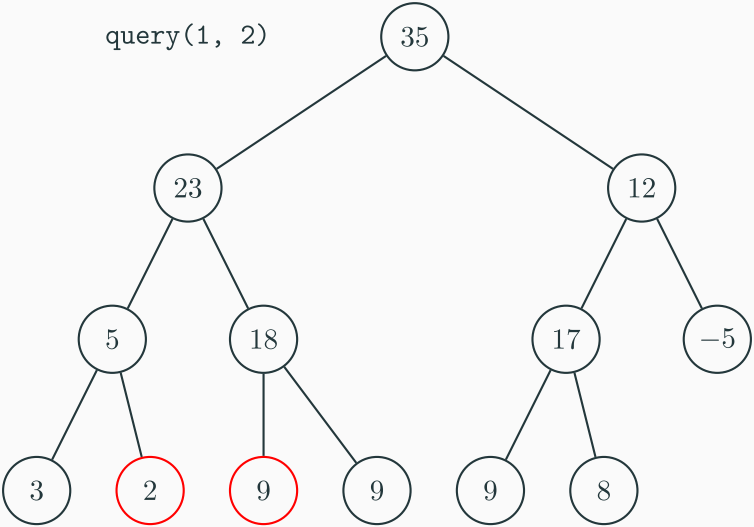
`query(1, 2)`



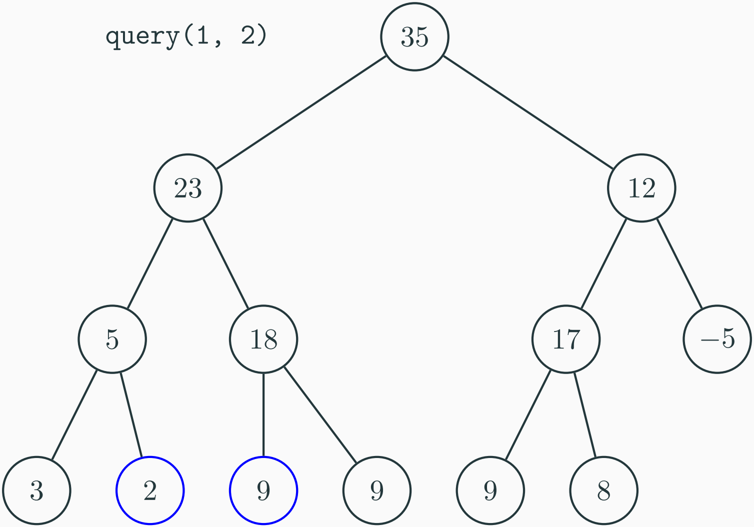
Querying



Querying

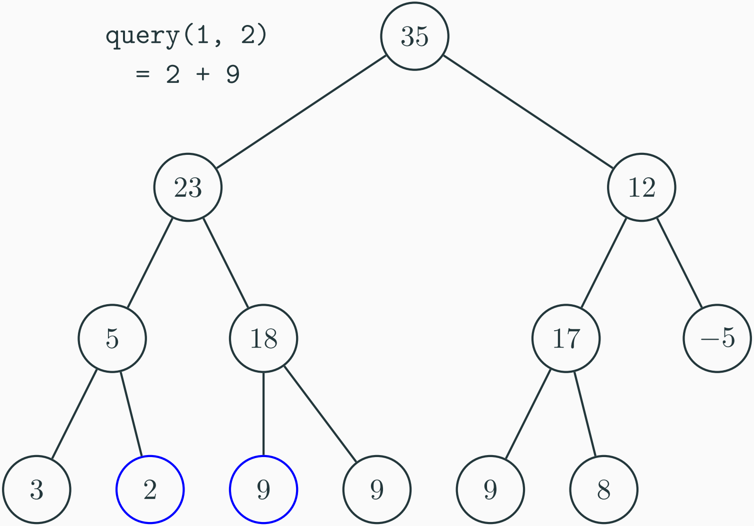


Querying

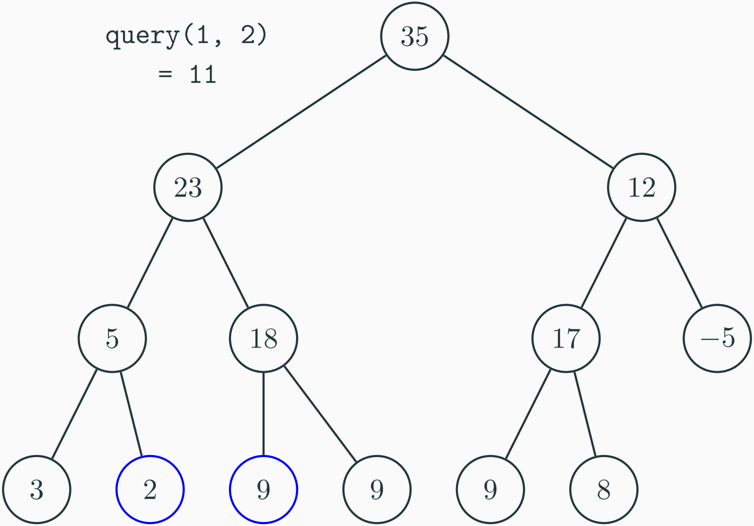


Querying

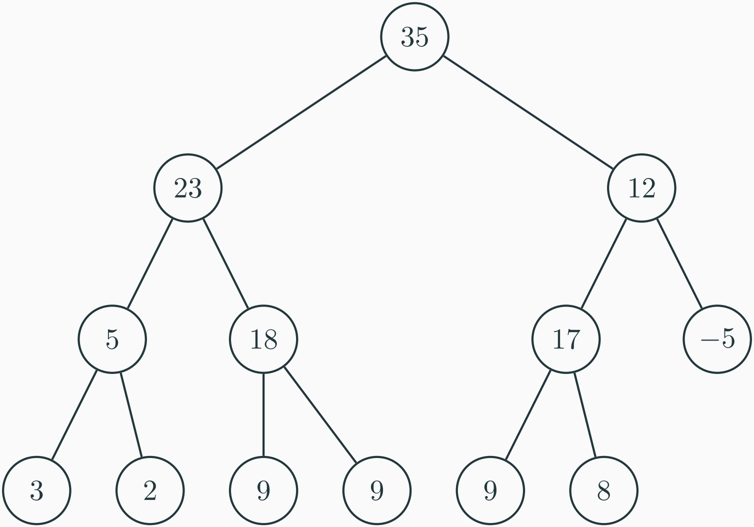
```
query(1, 2)
  = 2 + 9
```



Querying



Querying



Querying a Segment Tree - Code

```
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.

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.
- Also possible to update a range of values in $O(\log n)$, which will be covered in bonus slides.

Example problem: Movie Collection

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

Another $\log(n)$ idea

- What if we tried something more akin to an array.

Another $\log(n)$ idea

- What if we tried something more akin to an array.
- Could we store $\log(n)$ amounts of data per element somehow?

Another $\log(n)$ idea

- What if we tried something more akin to an array.
- Could we store $\log(n)$ amounts of data per element somehow?
- Yes! For each i we can store the sum on the interval $[i, i + 2^j - 1]$ for $\log$ many j.

Another $\log(n)$ idea

- What if we tried something more akin to an array.
- Could we store $\log(n)$ amounts of data per element somehow?
- Yes! For each i we can store the sum on the interval $[i, i + 2^j - 1]$ for $\log$ many j.
- Then to retrieve a sum from i to j we always take the biggest chunk we can that's stored at i, which will always be at least half.

Another $\log(n)$ idea

- What if we tried something more akin to an array.
- Could we store $\log(n)$ amounts of data per element somehow?
- Yes! For each i we can store the sum on the interval $[i, i + 2^j - 1]$ for $\log$ many j.
- Then to retrieve a sum from i to j we always take the biggest chunk we can that's stored at i, which will always be at least half.
- Then we continue until we reach j, moving i along and collecting the results.

Another $\log(n)$ idea

- What if we tried something more akin to an array.
- Could we store $\log(n)$ amounts of data per element somehow?
- Yes! For each i we can store the sum on the interval $[i, i + 2^j - 1]$ for $\log$ many j.
- Then to retrieve a sum from i to j we always take the biggest chunk we can that's stored at i, which will always be at least half.
- Then we continue until we reach j, moving i along and collecting the results.
- This is what is known as a sparse table.

Sparse tables

- Calculating all of these values takes $\mathcal{O}(n \log(n))$ because we can calculate the values in order of increasing j.

Sparse tables

- Calculating all of these values takes $\mathcal{O}(n\log(n))$ because we can calculate the values in order of increasing j.
- Then when we calculate the sum of $[i, i+2^j-1]$ we just combine the earlier results of $[i, i+2^{j-1}-1]$ and $[i+2^{j-1}, i+2^j-1]$.

Sparse tables

- Calculating all of these values takes $\mathcal{O}(n \log(n))$ because we can calculate the values in order of increasing j.
- Then when we calculate the sum of $[i, i + 2^j - 1]$ we just combine the earlier results of $[i, i + 2^{j-1} - 1]$ and $[i + 2^{j-1}, i + 2^j - 1]$.
- Querying takes $\mathcal{O}(\log(n))$, however updating is slow and difficult.

Sparse tables

- Calculating all of these values takes $\mathcal{O}(n\log(n))$ because we can calculate the values in order of increasing j.
- Then when we calculate the sum of $[i, i+2^j-1]$ we just combine the earlier results of $[i, i+2^{j-1}-1]$ and $[i+2^{j-1}, i+2^j-1]$.
- Querying takes $\mathcal{O}(\log(n))$, however updating is slow and difficult.
- Why would we then ever use this instead of segment trees?

Binary lifting

- The reason might be is that with sparse tables we can do many things that segment trees can not because of how the results are combined.

Binary lifting

- The reason might be is that with sparse tables we can do many things that segment trees can not because of how the results are combined.
- Let us consider binary lifting in particular.

Binary lifting

- The reason might be is that with sparse tables we can do many things that segment trees can not because of how the results are combined.
- Let us consider binary lifting in particular.
- Suppose we have some function f that rearranges the values $\{0, 1, \ldots, n-1\}$ and we get q queries asking what happens to x if we apply f exactly m times to x.

Binary lifting

- The reason might be is that with sparse tables we can do many things that segment trees can not because of how the results are combined.
- Let us consider binary lifting in particular.
- Suppose we have some function f that rearranges the values $\{0, 1, \ldots, n-1\}$ and we get q queries asking what happens to x if we apply f exactly m times to x.
- The naïve solution is to calculate it every time, giving a time complexity of $\mathcal{O}(qm\mathcal{O}(f))$.

Binary lifting

- The reason might be is that with sparse tables we can do many things that segment trees can not because of how the results are combined.
- Let us consider binary lifting in particular.
- Suppose we have some function f that rearranges the values $\{0, 1, \ldots, n-1\}$ and we get q queries asking what happens to x if we apply f exactly m times to x.
- The naïve solution is to calculate it every time, giving a time complexity of $\mathcal{O}(qm\mathcal{O}(f))$.
- How might we use sparse tables to do better?

Binary lifting ctd.

- Let $f^{[y]}(x)$ denote the result of applying f exactly y times to x

Binary lifting ctd.

- Let $f^{[y]}(x)$ denote the result of applying f exactly y times to x
- For each i we store $f^{[2^j]}(i)$ as a sparse table

Binary lifting ctd.

- Let $f^{[y]}(x)$ denote the result of applying f exactly y times to x
- For each i we store $f^{[2^j]}(i)$ as a sparse table
- Then we can compute these in increasing order of j, calculating $j = 1$ using f itself and then for larger j letting $f^{[2^j]}(x) = f^{[2^{j-1}]}(f^{[2^{j-1}]}(x))$

Binary lifting ctd.

- Let $f^{[y]}(x)$ denote the result of applying f exactly y times to x
- For each i we store $f^{[2^j]}(i)$ as a sparse table
- Then we can compute these in increasing order of j, calculating $j = 1$ using f itself and then for larger j letting $f^{[2^j]}(x) = f^{[2^{j-1}]}(f^{[2^{j-1}]}(x))$
- Thus we can precompute the table in $\mathcal{O}(n(\mathcal{O}(f) + \log(n)))$ and each query takes $\mathcal{O}(\log(m))$, a much better time complexity

Sparse table example

7	1	6	4	8	0	9	2	2	7	1	6	$j = 0$

Sparse table example

												$j = 1$
7	1	6	4	8	0	9	2	2	7	1	6	$j = 0$

Sparse table example

8												$j = 1$
7	1	6	4	8	0	9	2	2	7	1	6	$j = 0$

Sparse table example

8	7											$j = 1$
7	1	6	4	8	0	9	2	2	7	1	6	$j = 0$

Sparse table example

8	7	10										$j = 1$
7	1	6	4	8	0	9	2	2	7	1	6	$j = 0$

Sparse table example

8	7	10	12									$j = 1$
7	1	6	4	8	0	9	2	2	7	1	6	$j = 0$

Sparse table example

8	7	10	12	8								$j = 1$
7	1	6	4	8	0	9	2	2	7	1	6	$j = 0$

Sparse table example

8	7	10	12	8	9							$j = 1$
7	1	6	4	8	0	9	2	2	7	1	6	$j = 0$

Sparse table example

8	7	10	12	8	9	11						$j = 1$
7	1	6	4	8	0	9	2	2	7	1	6	$j = 0$

Sparse table example

8	7	10	12	8	9	11	4					$j = 1$
7	1	6	4	8	0	9	2	2	7	1	6	$j = 0$

Sparse table example

8	7	10	12	8	9	11	4	9				$j = 1$
7	1	6	4	8	0	9	2	2	7	1	6	$j = 0$

Sparse table example

8	7	10	12	8	9	11	4	9	8			$j = 1$
7	1	6	4	8	0	9	2	2	7	1	6	$j = 0$

Sparse table example

8	7	10	12	8	9	11	4	9	8	7		$j = 1$
7	1	6	4	8	0	9	2	2	7	1	6	$j = 0$

Sparse table example

8	7	10	12	8	9	11	4	9	8	7	6	$j = 1$
7	1	6	4	8	0	9	2	2	7	1	6	$j = 0$

Sparse table example

18	19	18	21	19	13	20	12	16	14	7	6	$j = 2$
8	7	10	12	8	9	11	4	9	8	7	6	$j = 1$
7	1	6	4	8	0	9	2	2	7	1	6	$j = 0$

Sparse table example

37	32	38	33	35	27	27	18	16	14	7	6	$j = 3$
18	19	18	21	19	13	20	12	16	14	7	6	$j = 2$
8	7	10	12	8	9	11	4	9	8	7	6	$j = 1$
7	1	6	4	8	0	9	2	2	7	1	6	$j = 0$

Sparse table example

`query(1, 8) = 19 + 9 + 2`

37	32	38	33	35	27	27	18	16	14	7	6	$j = 3$
18	19	18	21	19	13	20	12	16	14	7	6	$j = 2$
8	7	10	12	8	9	11	4	9	8	7	6	$j = 1$
7	1	6	4	8	0	9	2	2	7	1	6	$j = 0$

Sparse table example

`query(0, 9) = 37 + 9`

$\boxed{37}$	32	38	33	35	27	27	18	16	14	7	6	$j = 3$
18	19	18	21	19	13	20	12	16	14	7	6	$j = 2$
8	7	10	12	8	9	11	4	$\boxed{9}$	8	7	6	$j = 1$
7	1	6	4	8	0	9	2	2	7	1	6	$j = 0$

Example problem: Stikl

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