Data structures

Computer Science Enrichment Club - Algorithms Division

October 27, 2017

Today we're going to cover

- Review the Union-Find data structure, and look at applications
- Study range queries
- Quick look at Square Root Decomposition
- Learn about Segment Trees

Union-Find

- We have n items
- Maintains a collection of disjoint sets
- Each of the n items is in exactly one set

- $items = \{1, 2, 3, 4, 5, 6\}$
- $collections = \{1, 4\}, \{3, 5, 6\}, \{2\}$
- $collections = \{1\}, \{2\}, \{3\}, \{4\}, \{5\}, \{6\}$

- Supports two operations efficiently: `find(x)` and `union(x,y)`.

Union-Find

- $\textit{items} = \{1, 2, 3, 4, 5, 6\}$
- $\textit{collections} = \{1, 4\}, \{3, 5, 6\}, \{2\}$

- `find(x)` returns a representative item from the set that x is in
  - `find(1) = 1`
  - `find(4) = 1`
  - `find(3) = 5`
  - `find(5) = 5`
  - `find(6) = 5`
  - `find(2) = 2`

- a and b are in the same set if and only if `find(a) == find(b)`

Union-Find

- *items* $= \{1, 2, 3, 4, 5, 6\}$
- *collections* $= \{1, 4\}, \{3, 5, 6\}, \{2\}$

- `union(x, y)` merges the set containing x and the set containing y together.

  - `union(4, 2)`
  - *collections* $= \{1, 2, 4\}, \{3, 5, 6\}$
  - `union(3, 6)`
  - *collections* $= \{1, 2, 4\}, \{3, 5, 6\}$
  - `union(2, 6)`
  - *collections* $= \{1, 2, 3, 4, 5, 6\}$

Union-Find implementation

- Quick Union with path compression
- Extremely simple implementation
- Extremely efficient

```
struct union_find {
    vector<int> parent;
    union_find(int n) {
        parent = vector<int>(n);
        for (int i = 0; i < n; i++) {
            parent[i] = i;
        }
    }

    // find and union
};
```

Union-Find implementation

```
// find and union

int find(int x) {
    if (parent[x] == x) {
        return x;
    } else {
        parent[x] = find(parent[x]);
        return parent[x];
    }
}

void unite(int x, int y) {
    parent[find(x)] = find(y);
}
```

Union-Find implementation (short)

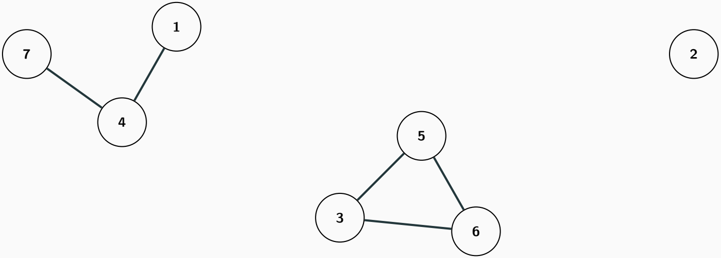
- If you're in a hurry...

```
#define MAXN 1000
int p[MAXN];

int find(int x) {
    return p[x] == x ? x : p[x] = find(p[x]); }
void unite(int x, int y) { p[find(x)] = find(y); }

for (int i = 0; i < MAXN; i++) p[i] = i;
```

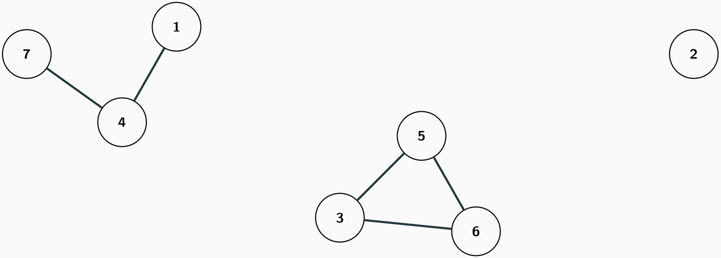
Union-Find applications

- Union-Find maintains a collection of disjoint sets
- When are we dealing with such collections?
- Most common example is in graphs

Disjoint sets in graphs

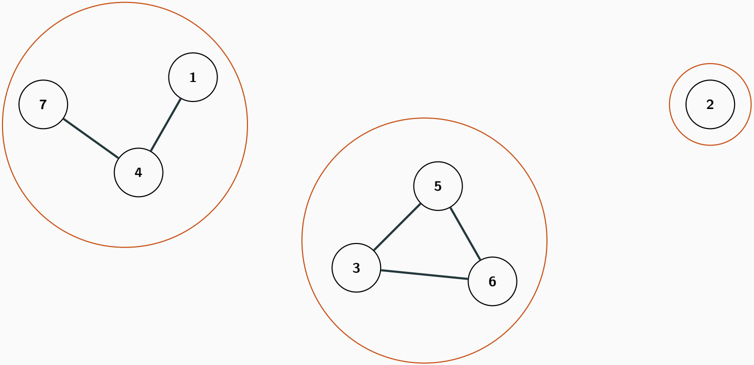


Disjoint sets in graphs



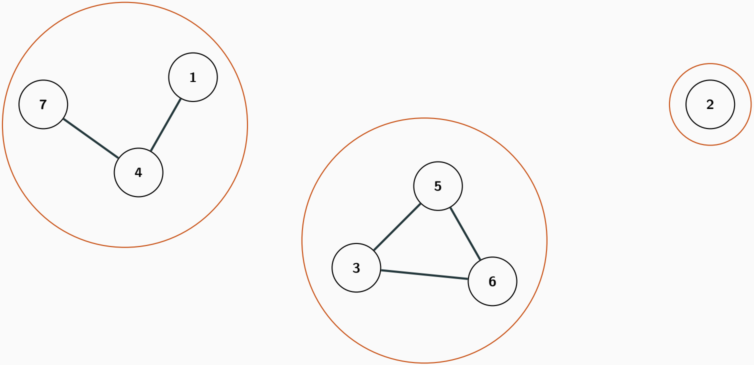
- *items* $= \{1, 2, 3, 4, 5, 6, 7\}$

Disjoint sets in graphs



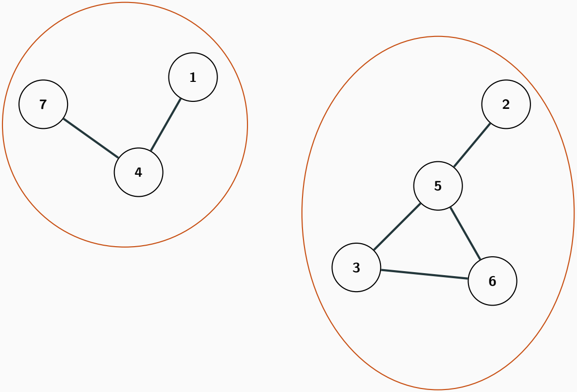
- *items* $= \{1, 2, 3, 4, 5, 6, 7\}$
- *collections* $= \{1, 4, 7\}, \{2\}, \{3, 5, 6\}$

Disjoint sets in graphs



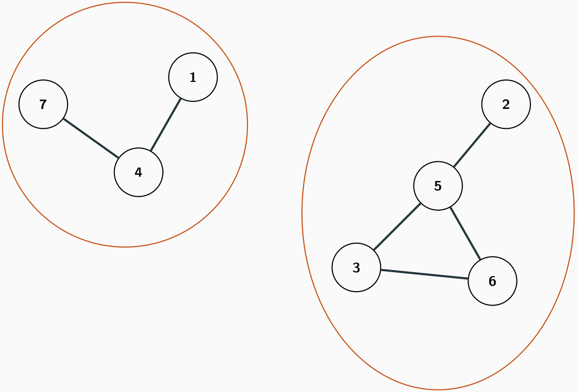
- *items* $= \{1, 2, 3, 4, 5, 6, 7\}$
- *collections* $= \{1, 4, 7\}, \{2\}, \{3, 5, 6\}$
- `union(2, 5)`

Disjoint sets in graphs



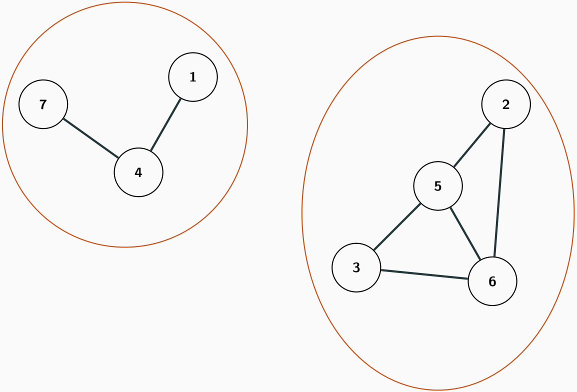
- *items* $= \{1, 2, 3, 4, 5, 6, 7\}$
- *collections* $= \{1, 4, 7\}, \{2, 3, 5, 6\}$

Disjoint sets in graphs



- *items* $= \{1, 2, 3, 4, 5, 6, 7\}$
- *collections* $= \{1, 4, 7\}, \{2, 3, 5, 6\}$
- `union(6, 2)`

Disjoint sets in graphs



- *items* $= \{1, 2, 3, 4, 5, 6, 7\}$
- *collections* $= \{1, 4, 7\}, \{2, 3, 5, 6\}$

Example problem: Where's My Internet??

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

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, i.e. 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 static array (i.e. updating is not supported)

1	0	7	8	5	9	3

Range sum on a static array

- Let's look at range sums on a static array (i.e. updating is not supported)

1	0	7	8	5	9	3

- $\mathrm{sum}(0, 6)$

Range sum on a static array

- Let's look at range sums on a static array (i.e. updating is not supported)

1	0	7	8	5	9	3

- $\mathrm{sum}(0, 6) = 33$

Range sum on a static array

- Let's look at range sums on a static array (i.e. updating is not supported)

1	0	7	8	5	9	3

- $\text{sum}(0, 6) = 33$
- $\text{sum}(2, 5)$

Range sum on a static array

- Let's look at range sums on a static array (i.e. updating is not supported)

1	0	7	8	5	9	3

- $\mathrm{sum}(0, 6) = 33$
- $\mathrm{sum}(2, 5) = 29$

Range sum on a static array

- Let's look at range sums on a static array (i.e. updating is not supported)

1	0	7	8	5	9	3

- $\text{sum}(0, 6) = 33$
- $\text{sum}(2, 5) = 29$
- $\text{sum}(2, 2)$

Range sum on a static array

- Let's look at range sums on a static array (i.e. updating is not supported)

1	0	7	8	5	9	3

- $\text{sum}(0, 6) = 33$
- $\text{sum}(2, 5) = 29$
- $\text{sum}(2, 2) = 7$

Range sum on a static array

- Let's look at range sums on a static array (i.e. updating is not supported)

1	0	7	8	5	9	3

- $\mathrm{sum}(0, 6) = 33$
- $\mathrm{sum}(2, 5) = 29$
- $\mathrm{sum}(2, 2) = 7$

- How do we support these queries efficiently?

Range sum on a static array

- Simplification: only support queries of the form $\mathrm{sum}(0, j)$
- Notice that $\mathrm{sum}(i, j) = \mathrm{sum}(0, j) - \mathrm{sum}(0, i-1)$

1	0	7	8	5	9	3

$$=$$

1	0	7	8	5	9	3

$$-$$

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

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
- $O(1)$ 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

Range sum on a dynamic array

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

1	0	7	8	5	9	3

Range sum on a dynamic array

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

1	0	7	8	5	9	3

- $\text{sum}(0, 6)$

Range sum on a dynamic array

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

1	0	7	8	5	9	3

- $\text{sum}(0, 6) = 33$

Range sum on a dynamic array

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

1	0	7	8	5	9	3

- $\text{sum}(0, 6) = 33$
- $\text{update}(3, -2)$

Range sum on a dynamic array

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

1	0	7	-2	5	9	3

- $\text{sum}(0, 6) = 33$
- $\text{update}(3, -2)$

Range sum on a dynamic array

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

1	0	7	-2	5	9	3

- $\mathrm{sum}(0, 6) = 33$
- $\mathrm{update}(3, -2)$
- $\mathrm{sum}(0, 6)$

Range sum on a dynamic array

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

1	0	7	-2	5	9	3

- $\text{sum}(0, 6) = 33$
- $\text{update}(3, -2)$
- $\text{sum}(0, 6) = 23$

Range sum on a dynamic array

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

1	0	7	-2	5	9	3

- $\text{sum}(0, 6) = 33$
- $\text{update}(3, -2)$
- $\text{sum}(0, 6) = 23$

- How do we support these queries efficiently?

First attempt: Buckets

- Group values into buckets of size k
- E.g. $k = 2$:

1	0	7	8	5	9	3

First attempt: Buckets

- Group values into buckets of size k
- E.g. $k = 2$:

1	0	7	8	5	9	3

First attempt: Buckets

- Group values into buckets of size k
- E.g. $k = 2$:

1	0	7	8	5	9	3

- There are roughly n/k buckets

First attempt: Buckets

- Group values into buckets of size k
- E.g. $k = 2$:

1	0	7	8	5	9	3

- There are roughly n/k buckets
- Store the sum of elements inside each bucket:

First attempt: Buckets

- Group values into buckets of size k
- E.g. $k = 2$:

1	0	7	8	5	9	3

- There are roughly n/k buckets
- Store the sum of elements inside each bucket:

1	15	14	3

Buckets: Updating

1	0	7	8	5	9	3

1	15	14	3

- Updating is easy:
    - change the array element
    - recompute corresponding bucket

Buckets: Updating

1	0	7	8	5	9	3

1	15	14	3

- Updating is easy:
    - change the array element
    - recompute corresponding bucket
- $\mathrm{update}(3, -2)$

Buckets: Updating

1	0	7	8	5	9	3

1	15	14	3

- Updating is easy:
  - change the array element
  - recompute corresponding bucket
- $\text{update}(3, -2)$

Buckets: Updating

1	0	7	-2	5	9	3

1	15	14	3

- Updating is easy:
  - change the array element
  - recompute corresponding bucket
- $\mathrm{update}(3, -2)$

Buckets: Updating

1	0	7	-2	5	9	3

1	15	14	3

- Updating is easy:
    - change the array element
    - recompute corresponding bucket
- $\text{update}(3, -2)$

Buckets: Updating

1	0	7	-2	5	9	3

1	15	14	3

- Updating is easy:
    - change the array element
    - recompute corresponding bucket
- $\text{update}(3, -2)$

Buckets: Updating

1	0	7	-2	5	9	3

1	5	14	3

- Updating is easy:
    - change the array element
    - recompute corresponding bucket
- $\text{update}(3, -2)$

Buckets: Updating

1	0	7	-2	5	9	3

1	5	14	3

- Updating is easy:
  - change the array element
  - recompute corresponding bucket
- $\text{update}(3, -2)$
- Time complexity: $O(k)$

Buckets: Updating

1	0	7	-2	5	9	3

1	5	14	3

- Updating is easy:
  - change the array element
  - recompute corresponding bucket
- $\text{update}(3, -2)$
- Time complexity: $O(k)$
- Easy to do in $O(1)$, but doesn't really matter (we'll see why)

Buckets: Querying

1	0	7	8	5	9	3

1	15	14	3

- 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

Buckets: Querying

1	0	7	8	5	9	3

1	15	14	3

- 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
- $\text{query}(1, 5)$

Buckets: Querying

1	0	7	8	5	9	3

1	15	14	3

- 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
- $\mathrm{query}(1, 5)$

Buckets: Querying

1	0	7	8	5	9	3

1	15	14	3

- 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
- $\mathrm{query}(1, 5) = 0 + 15 + 14 = 29$

Buckets: Querying

1	0	7	8	5	9	3

1	15	14	3

- 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
- $\mathrm{query}(1,5) = 0 + 15 + 14 = 29$
- What about time complexity?

Buckets: Querying

1	0	7	8	5	9	3

1	15	14	3

- 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
- $\text{query}(1, 5) = 0 + 15 + 14 = 29$
- What about time complexity?
  - Only have to go inside at most two buckets (each end)

Buckets: Querying

1	0	7	8	5	9	3

1	15	14	3

- 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
- $\text{query}(1, 5) = 0 + 15 + 14 = 29$
- What about time complexity?
  - Only have to go inside at most two buckets (each end)
  - Have to consider at most n/k buckets

Buckets: Querying

1	0	7	8	5	9	3

1	15	14	3

- 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
- $\mathrm{query}(1, 5) = 0 + 15 + 14 = 29$
- What about time complexity?
  - Only have to go inside at most two buckets (each end)
  - Have to consider at most n/k buckets
  - In total roughly $n/k + 2k$

Buckets: Querying

1	0	7	8	5	9	3

1	15	14	3

- 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
- $\text{query}(1,5) = 0 + 15 + 14 = 29$
- What about time complexity?
  - Only have to go inside at most two buckets (each end)
  - Have to consider at most n/k buckets
  - In total roughly $n/k + 2k$
  - Time complexity: $O(n/k + k)$

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 $k = \sqrt{n}$:
  - 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 $O(\sqrt{n})$
- May be too slow if n is large and many queries

- Can we do better?

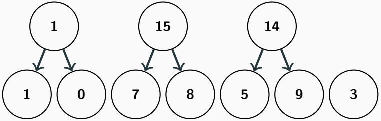
Second attempt: Segment Tree

1	0	7	8	5	9	3

Second attempt: Segment Tree

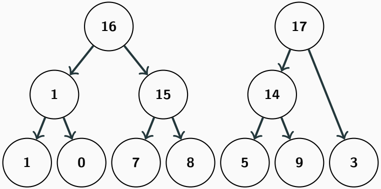
1	0	7	8	5	9	3

Second attempt: Segment Tree



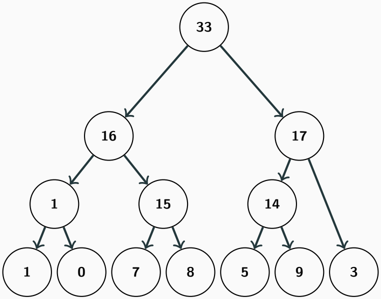
1	0	7	8	5	9	3

Second attempt: Segment Tree



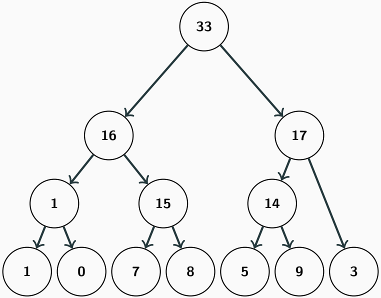
1	0	7	8	5	9	3

Second attempt: Segment Tree



1	0	7	8	5	9	3

Second attempt: Segment Tree



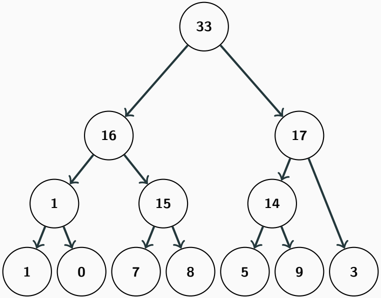
1	0	7	8	5	9	3

- Each vertex contains the sum of some segment of the array

Segment Tree - Code

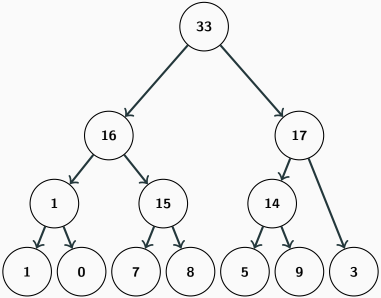
```
struct segment_tree {
    segment_tree *left, *right;
    int from, to, value;
    segment_tree(int from, int to)
        : from(from), to(to), left(NULL), right(NULL), value(0) { }
};

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;
}
```

Querying a Segment Tree



1	0	7	8	5	9	3

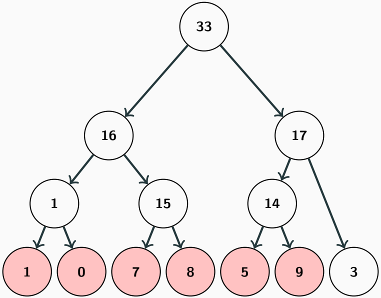
Querying a Segment Tree



1	0	7	8	5	9	3

- $\mathrm{sum}(0, 5)$

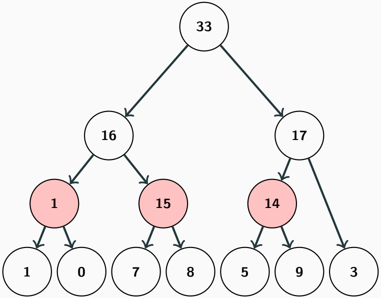
Querying a Segment Tree



1	0	7	8	5	9	3

- $\text{sum}(0, 5)$

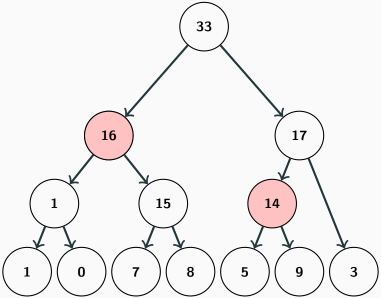
Querying a Segment Tree



1	0	7	8	5	9	3

- $\mathrm{sum}(0, 5)$

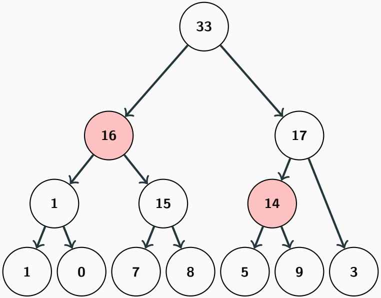
Querying a Segment Tree



1	0	7	8	5	9	3

- $\mathrm{sum}(0, 5)$

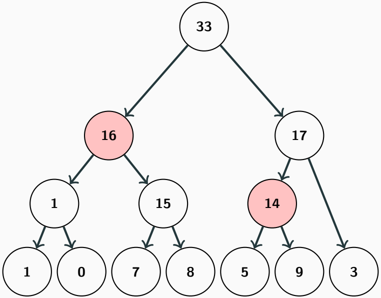
Querying a Segment Tree



1	0	7	8	5	9	3

- $\text{sum}(0, 5) = 16 + 14 = 30$

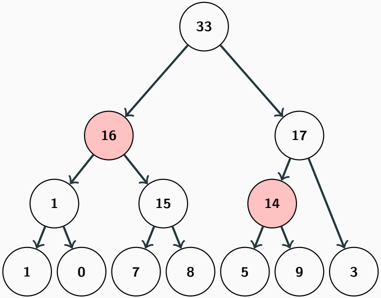
Querying a Segment Tree



1	0	7	8	5	9	3

- $\text{sum}(0, 5) = 16 + 14 = 30$
- We only need to consider a few vertices to get the entire range

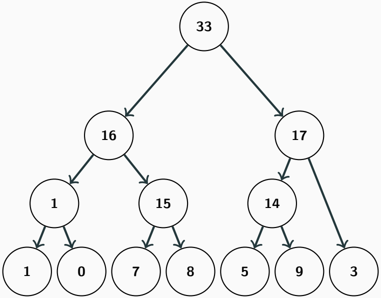
Querying a Segment Tree



1	0	7	8	5	9	3

- $\mathrm{sum}(0, 5) = 16 + 14 = 30$
- We only need to consider a few vertices to get the entire range
- But how do we find them?

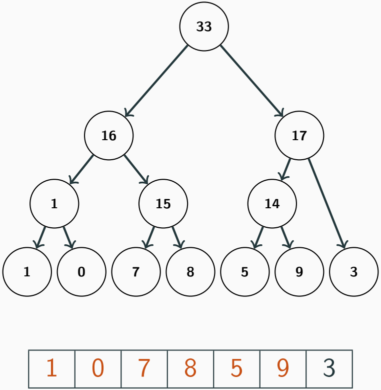
Querying a Segment Tree



1	0	7	8	5	9	3

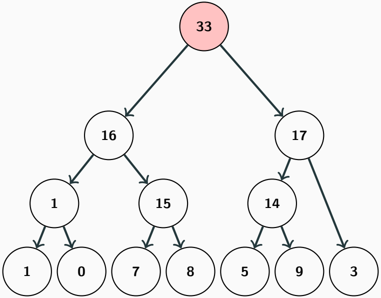
- $\mathrm{sum}(0, 5)$

Querying a Segment Tree



- $\mathrm{sum}(0, 5)$

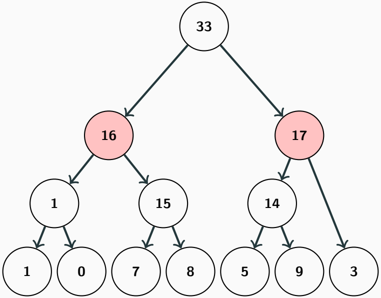
Querying a Segment Tree



1	0	7	8	5	9	3

- $\text{sum}(0, 5)$

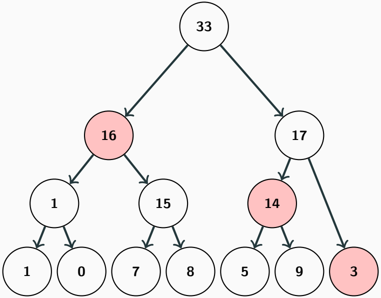
Querying a Segment Tree



1	0	7	8	5	9	3

- $\mathrm{sum}(0, 5)$

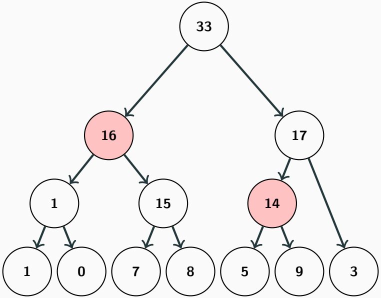
Querying a Segment Tree



1	0	7	8	5	9	3

- $\text{sum}(0, 5)$

Querying a Segment Tree



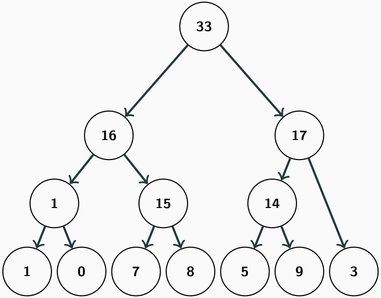
1	0	7	8	5	9	3

- $\mathrm{sum}(0, 5)$

Querying a Segment Tree - Code

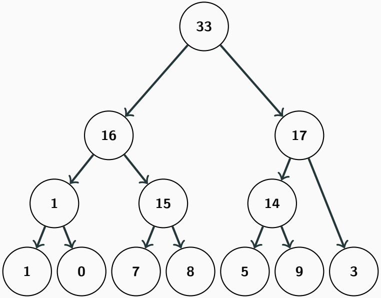
```
int query(segment_tree *tree, int l, int r) {
    if (tree == NULL) return 0;
    if (l <= tree->from && tree->to <= r) return tree->value;
    if (tree->to < l) return 0;
    if (r < tree->from) return 0;
    return query(tree->left, l, r) + query(tree->right, l, r);
}
```

Updating a Segment Tree



1	0	7	8	5	9	3

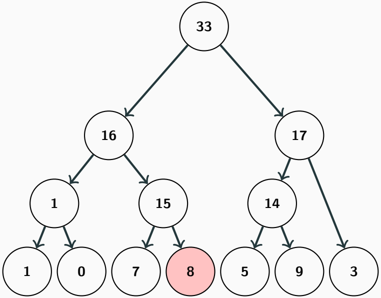
Updating a Segment Tree



1	0	7	8	5	9	3

- *update*$(3, 5)$

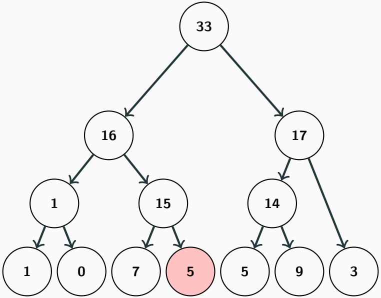
Updating a Segment Tree



1	0	7	8	5	9	3

- *update*$(3, 5)$

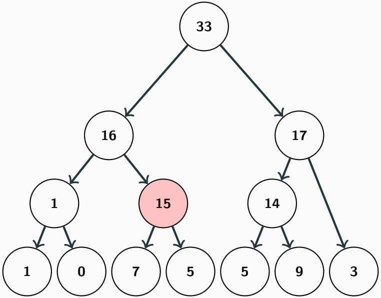
Updating a Segment Tree



1	0	7	5	5	9	3

- $update(3, 5)$

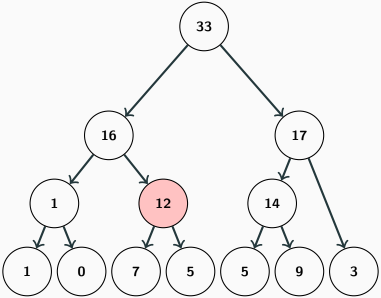
Updating a Segment Tree



1	0	7	5	5	9	3

- *update*$(3, 5)$

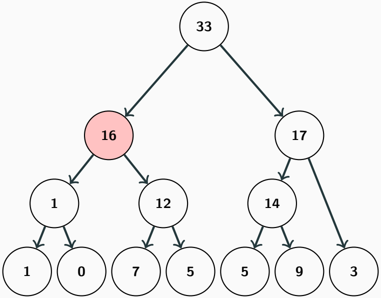
Updating a Segment Tree



1	0	7	5	5	9	3

- *update*$(3, 5)$

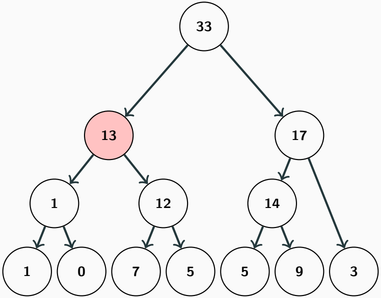
Updating a Segment Tree



1	0	7	5	5	9	3

- *update*$(3, 5)$

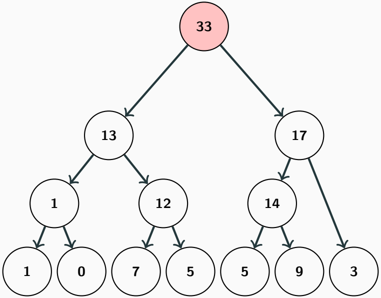
Updating a Segment Tree



1	0	7	5	5	9	3

- $update(3, 5)$

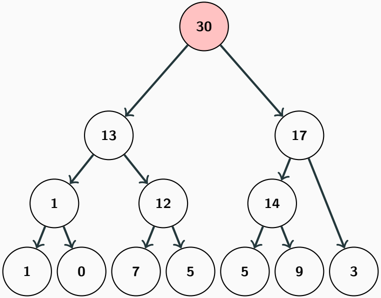
Updating a Segment Tree



1	0	7	5	5	9	3

- $update(3, 5)$

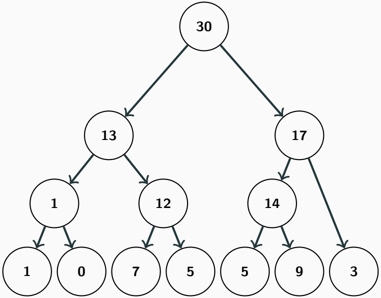
Updating a Segment Tree



1	0	7	5	5	9	3

- *update*$(3, 5)$

Updating a Segment Tree



1	0	7	5	5	9	3

- *update*$(3, 5)$

Updating a Segment Tree - Code

```c
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;
}
```

Segment Tree

- Now we can
  - build a Segment Tree
  - query a range
  - update a single value

Segment Tree

- Now we can
  - build a Segment Tree
  - query a range
  - update a single value
- But how efficient are these operations?

Segment Tree

- Now we can
  - build a Segment Tree in $O(n)$
  - query a range
  - update a single value
- But how efficient are these operations?

Segment Tree

- Now we can
  - build a Segment Tree in $O(n)$
  - query a range in $O(\log n)$
  - update a single value
- But how efficient are these operations?

Segment Tree

- Now we can
  - build a Segment Tree in $O(n)$
  - query a range in $O(\log n)$
  - update a single value in $O(\log n)$
- But how efficient are these operations?

Segment Tree

- Now we can
  - build a Segment Tree in $O(n)$
  - query a range in $O(\log n)$
  - update a single value in $O(\log n)$
- But how efficient are these operations?

- Trivial to use Segment Trees for min, max, gcd, and other similar operators, basically the same code

Segment Tree

- Now we can
  - build a Segment Tree in $O(n)$
  - query a range in $O(\log n)$
  - update a single value in $O(\log n)$
- But how efficient are these operations?

- Trivial to use Segment Trees for min, max, gcd, and other similar operators, basically the same code
- Also possible to update a range of values in $O(\log n)$ (Google for Segment Trees with Lazy Propagation if you want to learn more)

Example problem: Supercomputer

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