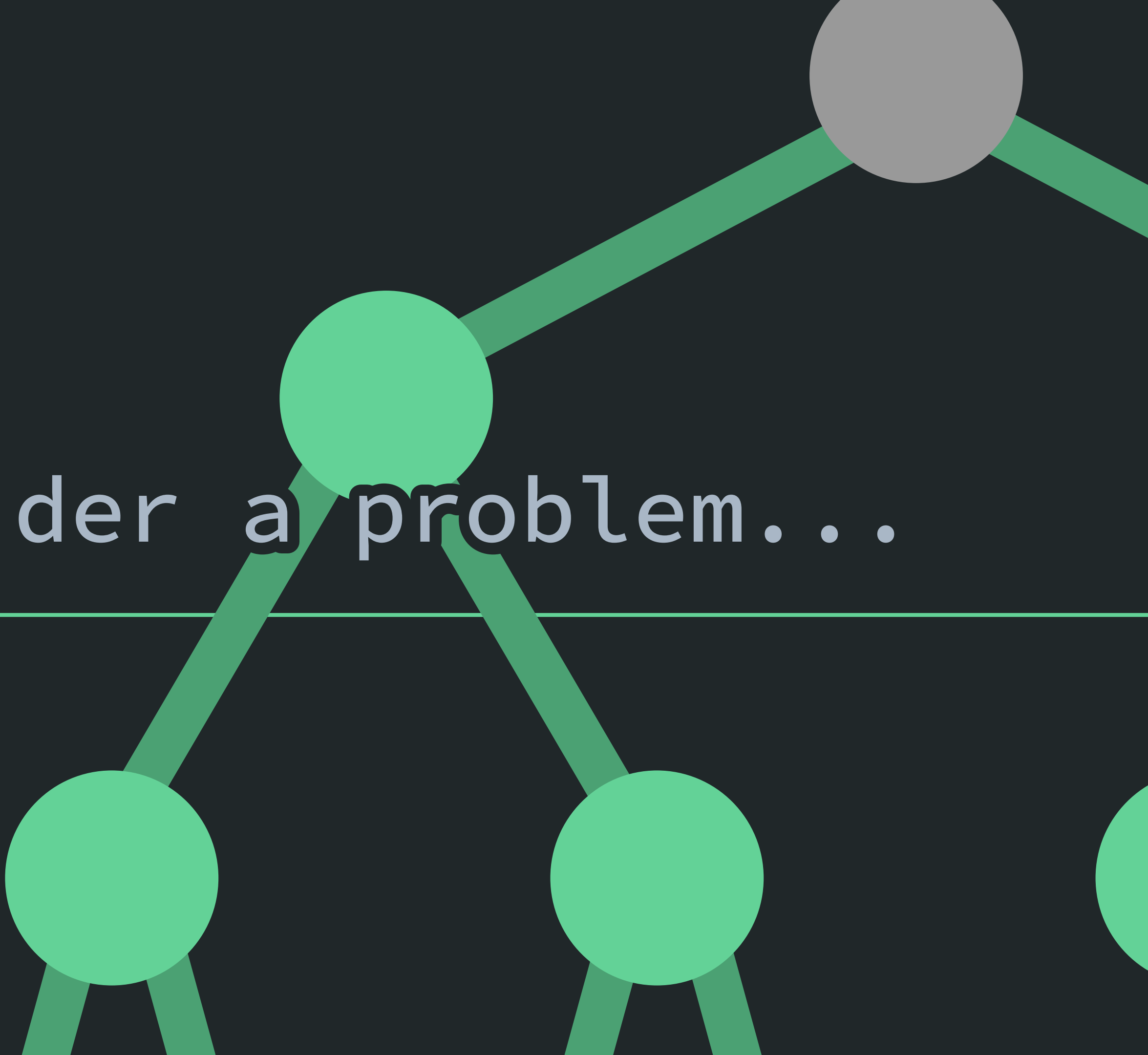
Let’s consider a problem...



Given an array $arr[0...n-1]$, perform

1. U # updates that increase the value at index i by val
2. Q # queries to find the sum over the interval $[l, r]$, where $l<r$ and are indices of arr

Queries and updates can be performed in any order

Example

```
arr = [0,1,2,3,4,5,6,7]

# Q = 2, U = 1
query(l=0, r=6)         # returns 21 (0+1+2+3+4+5+6)
update(i=2, val=1)      # update arr to [0,1,3,3,4,5,6,7]
query(l=1, r=5)         # returns 16 (1+3+3+4+5)
```

A Prefix Sum Array

1. For updates, apply

```
arr[i] += val
```

2. If a query comes after an update, process the array sum of arr so that
3. For each query, sum over the range $[l, r]$ could be found by

```
arraySum[r]-arraySum[l-1]
```

4. Optimal for large groups updates and queries

Implementation

```python
arr = [0,1,2,3,4,5,6,7]
arraySum = []*len(arr)

def update (i, val):
    arr[i] += val;

# Called if a query comes after an update
def buildArraySum():
    arraySum = arr[0]
    for i in range(1, len(arr)):
        arraySum = arraySum[i-1]+arr[i]

# Return the sum over the range
def query (l, r):
    return arraySum[r]-arraySum[l-1]

```

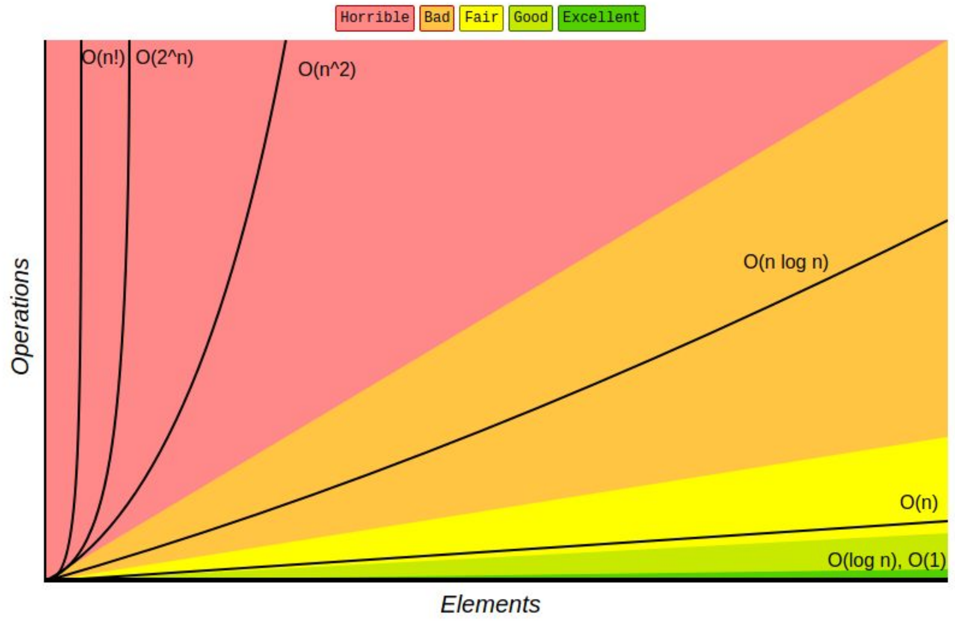
The Problem

1. Slow with consecutive updates and queries
2. Processing array sum takes O(n) time
3. Overall time complexity: O(Q*n)

Large Array + Millions of Queries

= not good

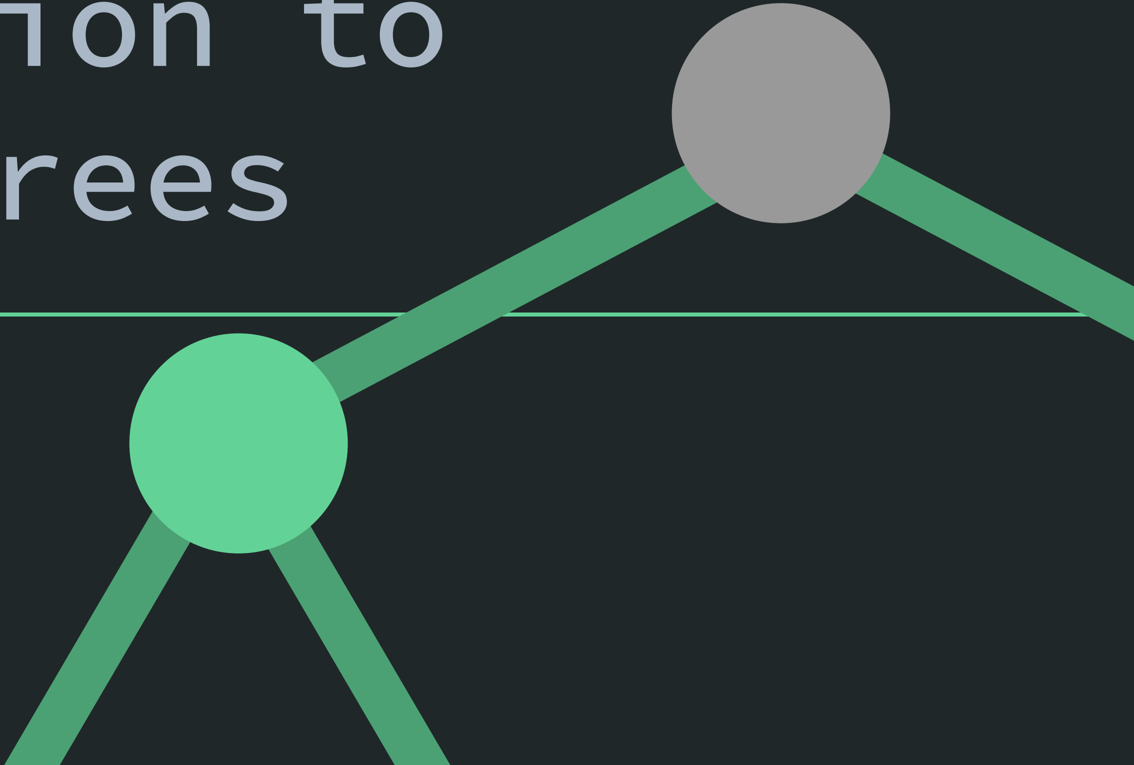
Can we do it better? Yes!



Introduction to Segment Trees

Felix Fong

Note: view slides in presentation mode



Segment Tree

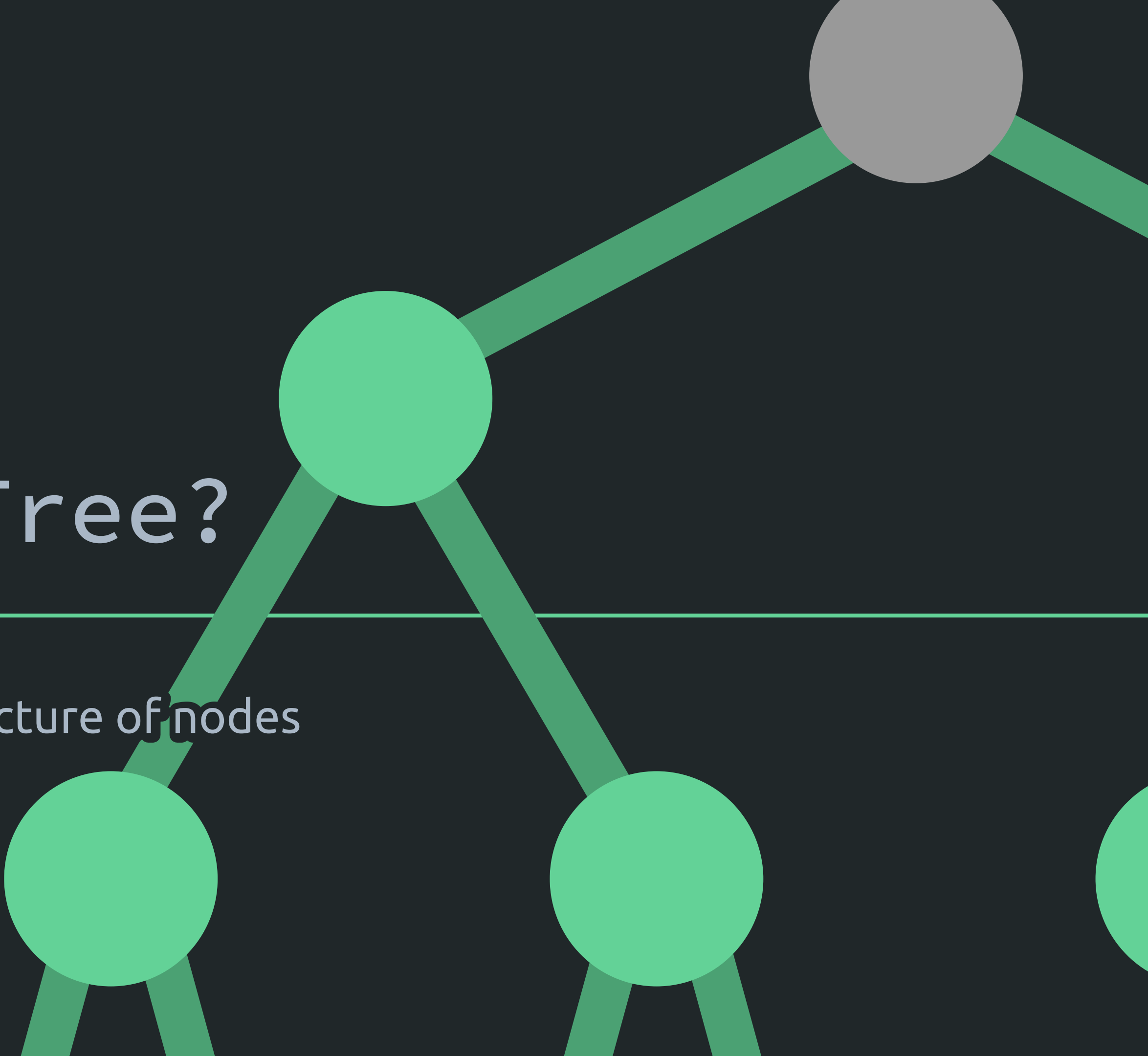
- A data structure that stores information over segments of an array
- $O(\log_2 n)$ complexity for updates and range queries

Pros

- Same uses as sum arrays
- Supports irreversible operations
- Offers min/max, gcd/lcm, bitwise, etc. over $[l, r]$

Cons

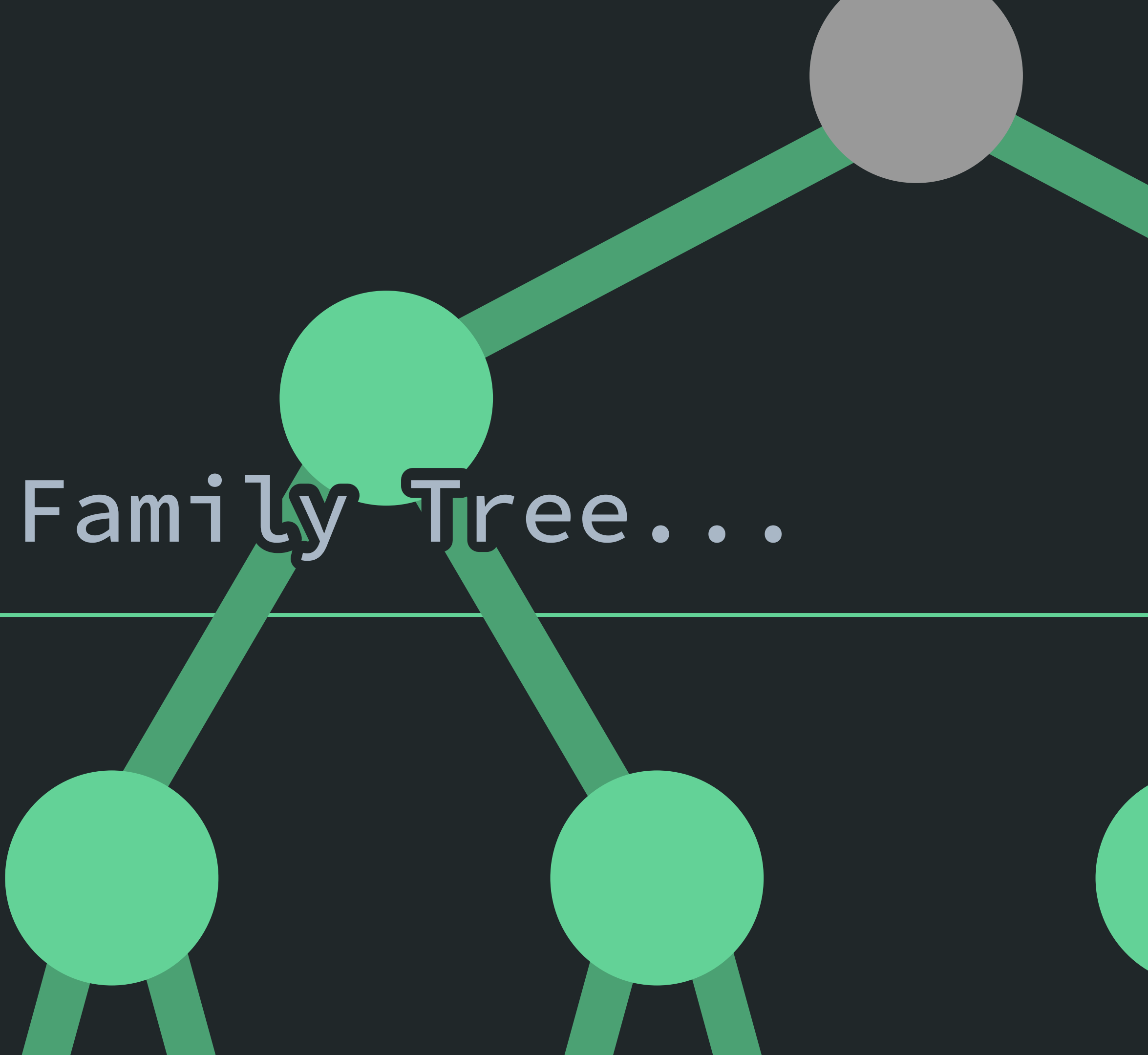
- Takes O(2n) memory
- Given large updates and few queries, sum arrays are faster
- Does not exist in the library



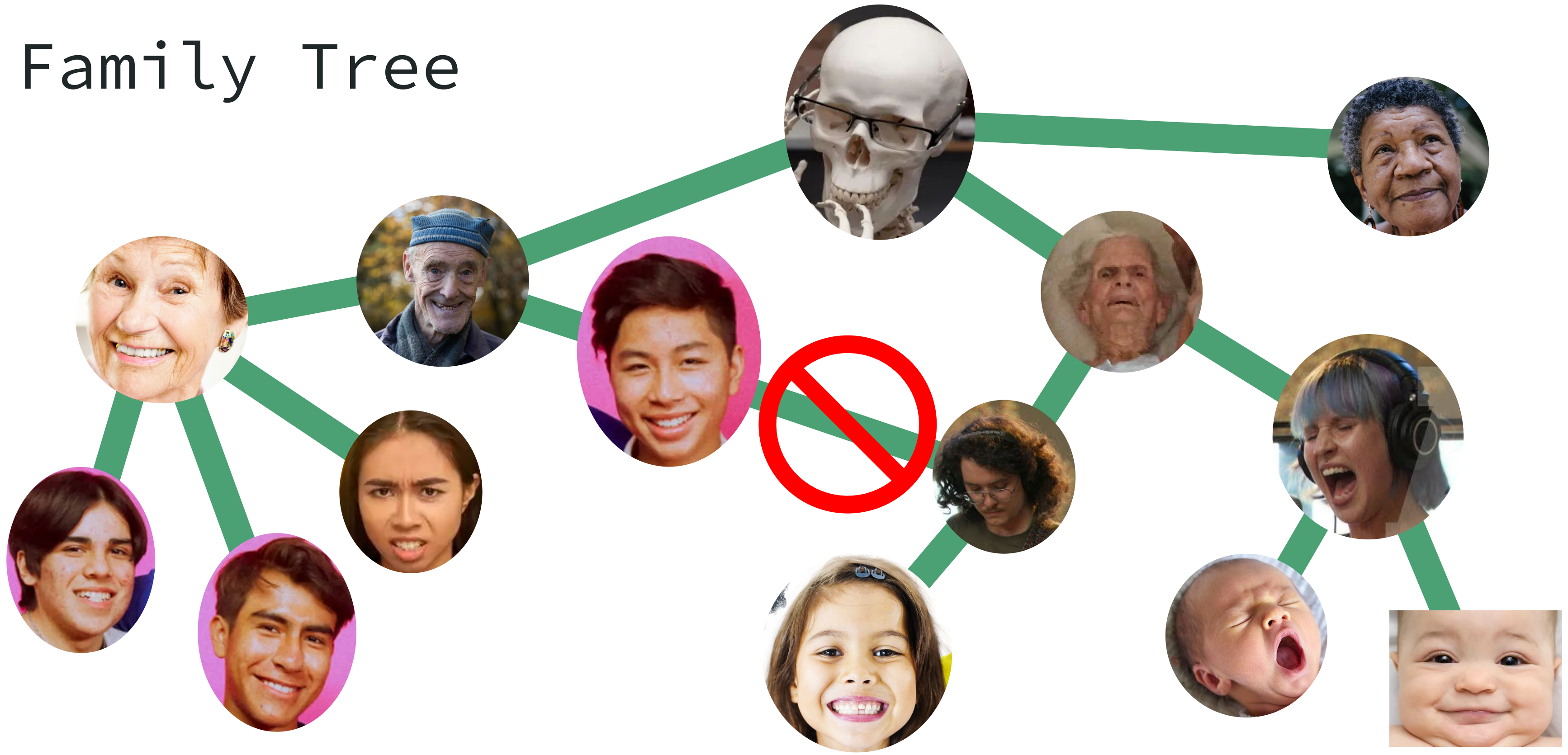
What is a Tree?

- A **hierarchical** data structure of nodes

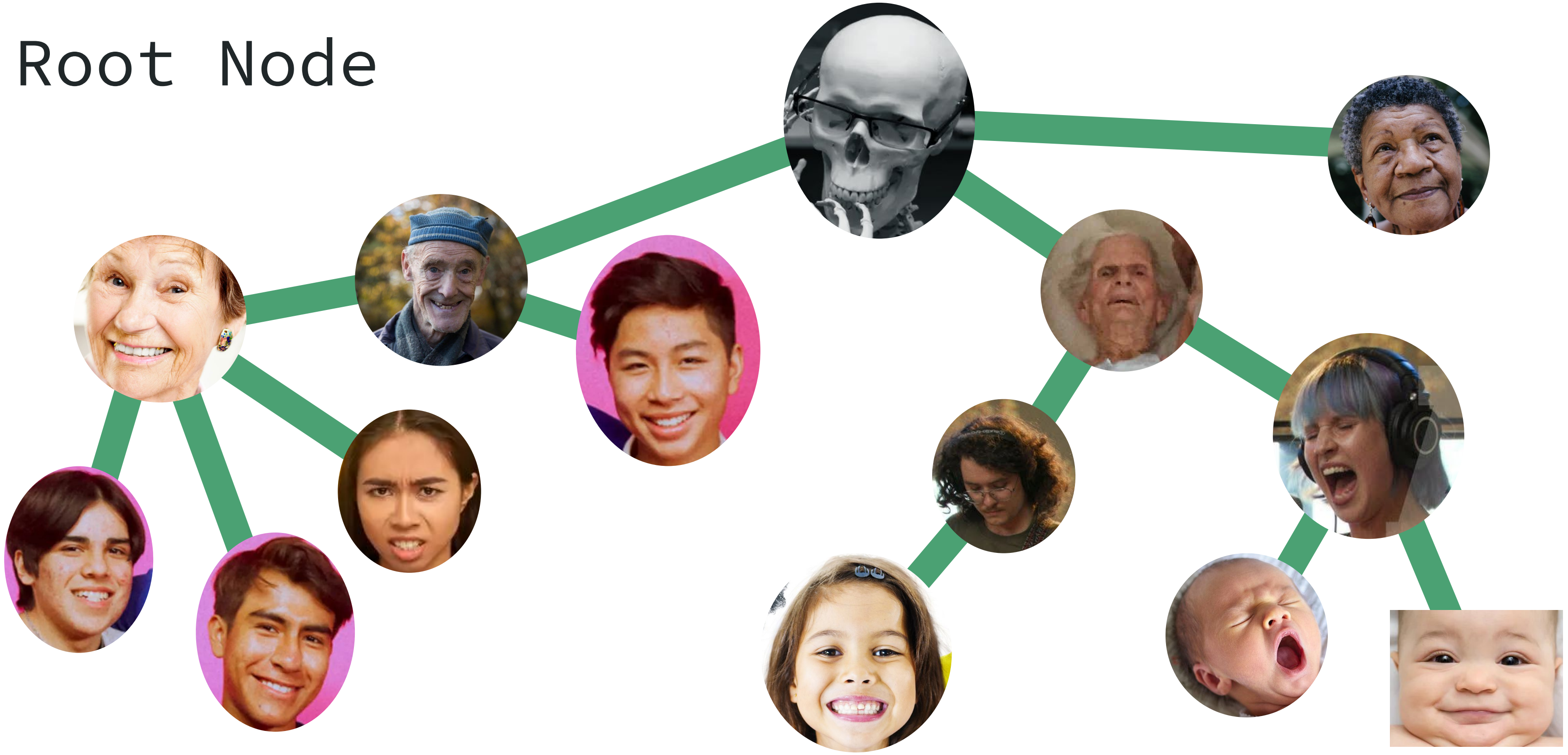
Think of a Family Tree...



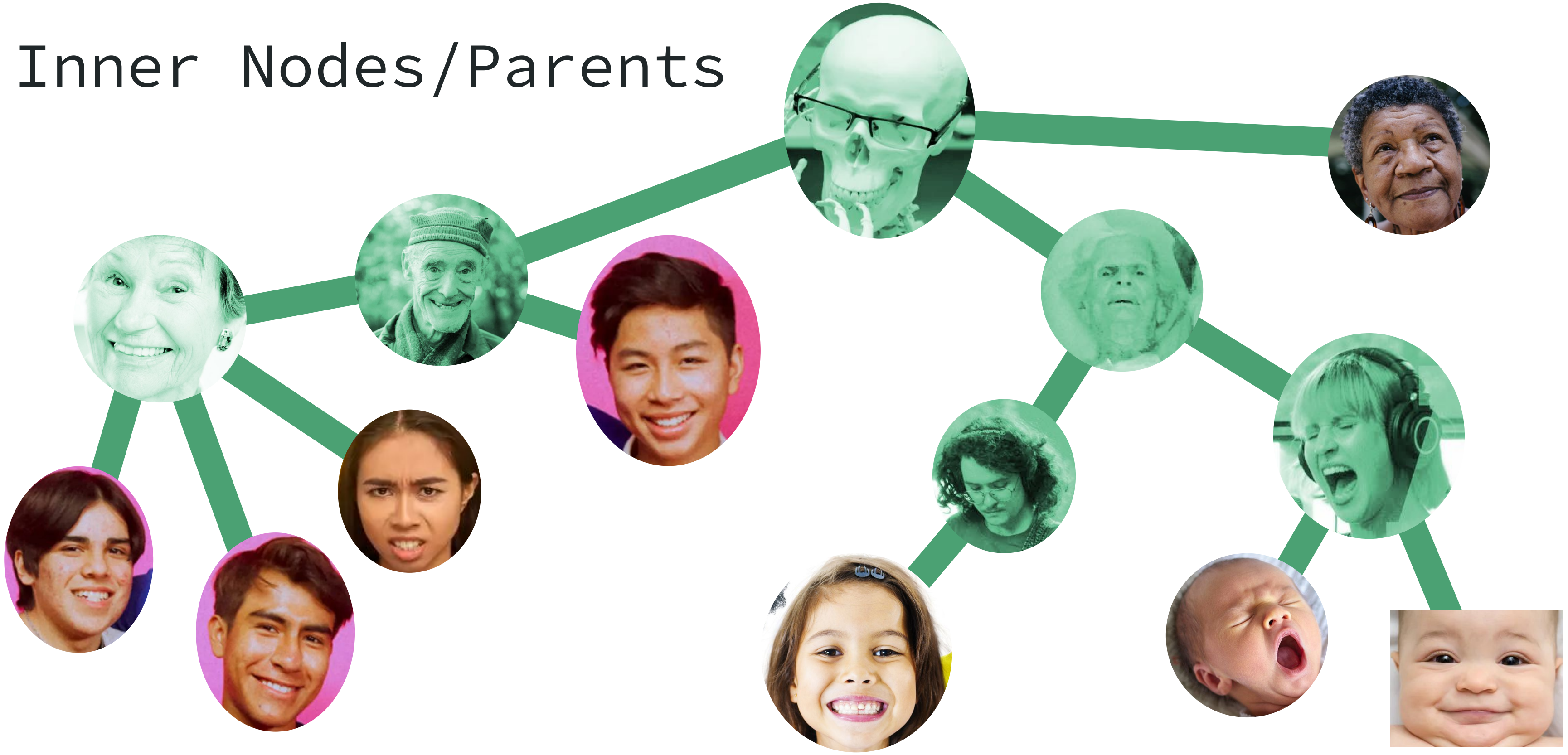
Family Tree



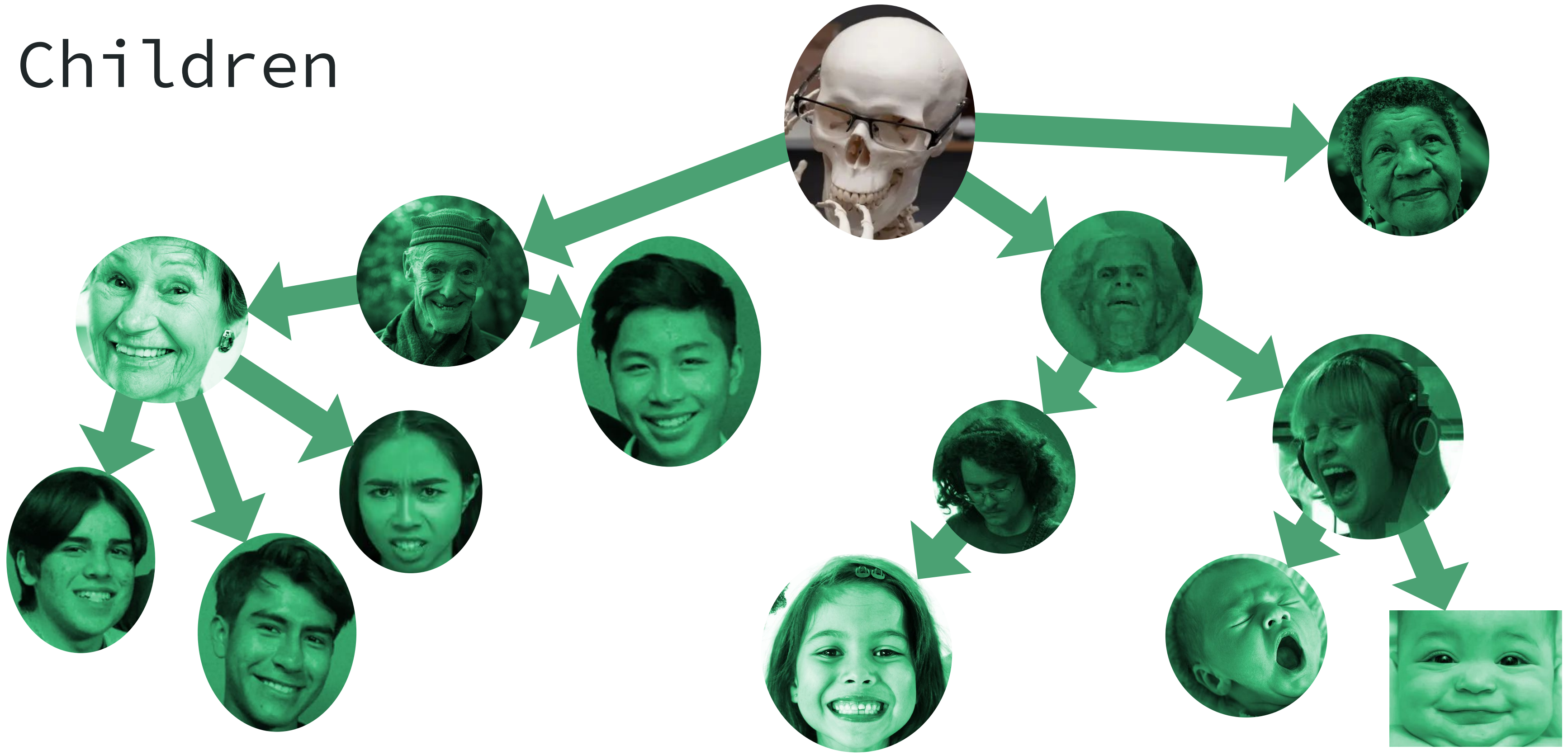
Root Node



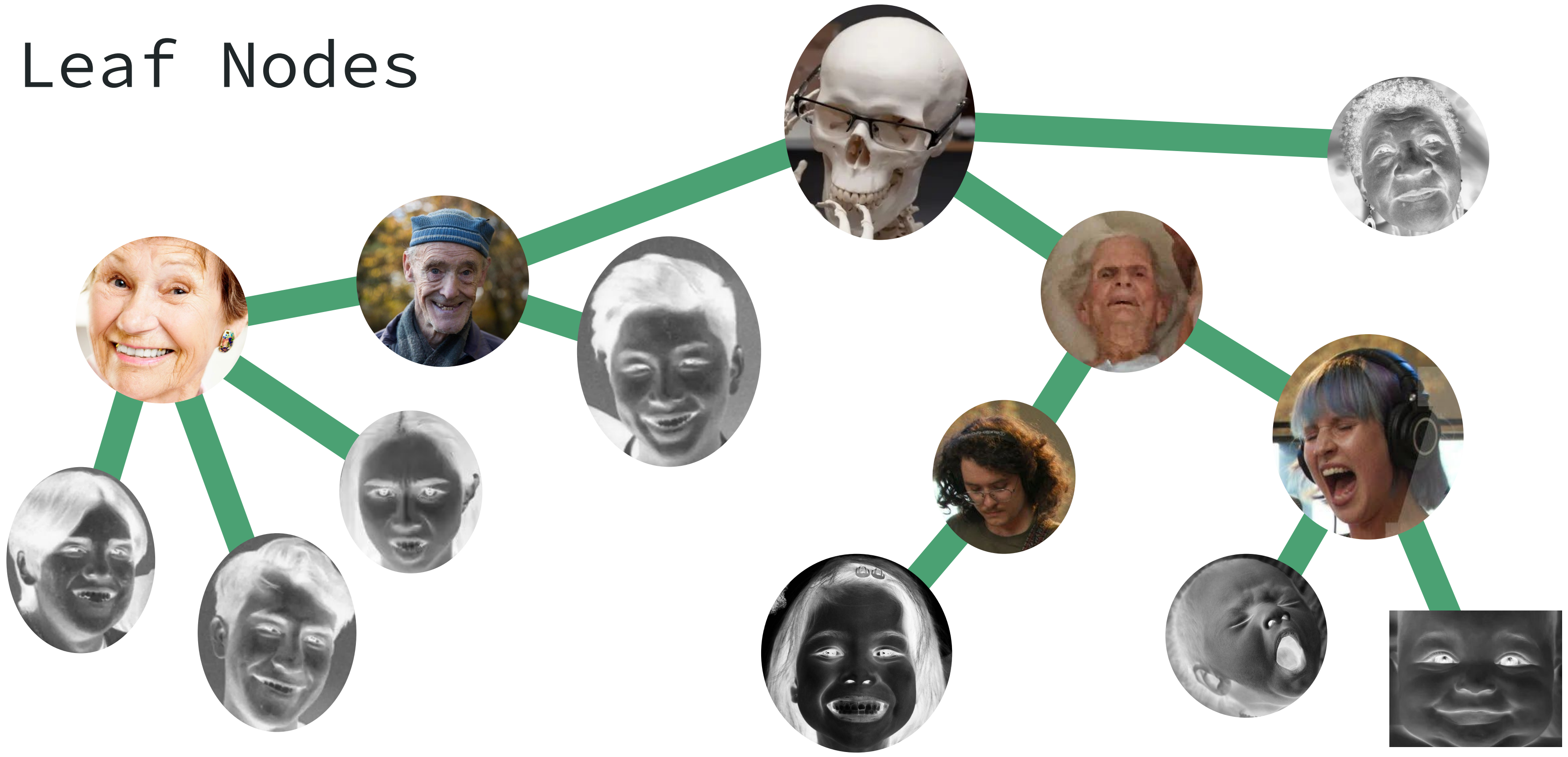
Inner Nodes/Parents



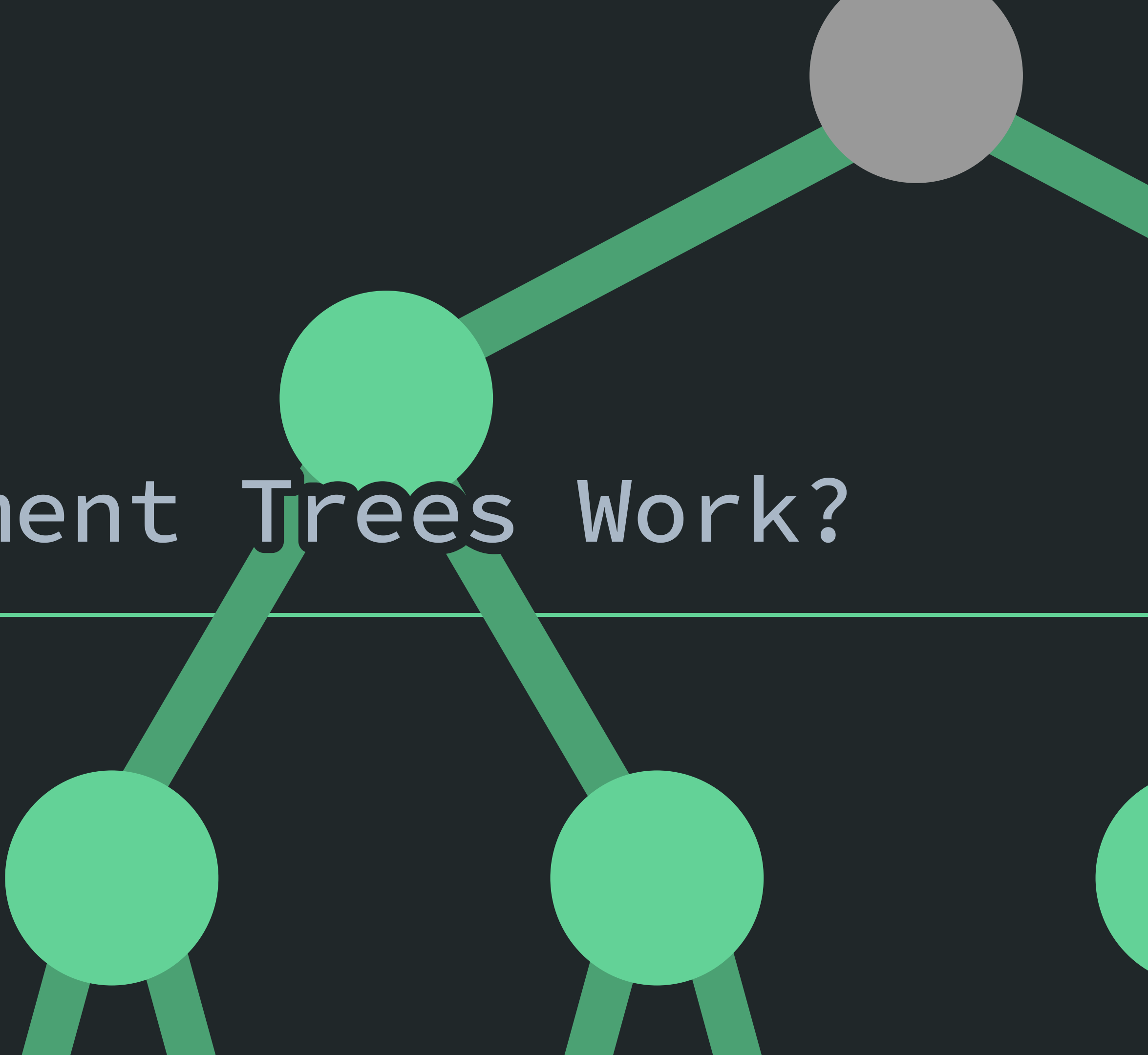
Children



Leaf Nodes

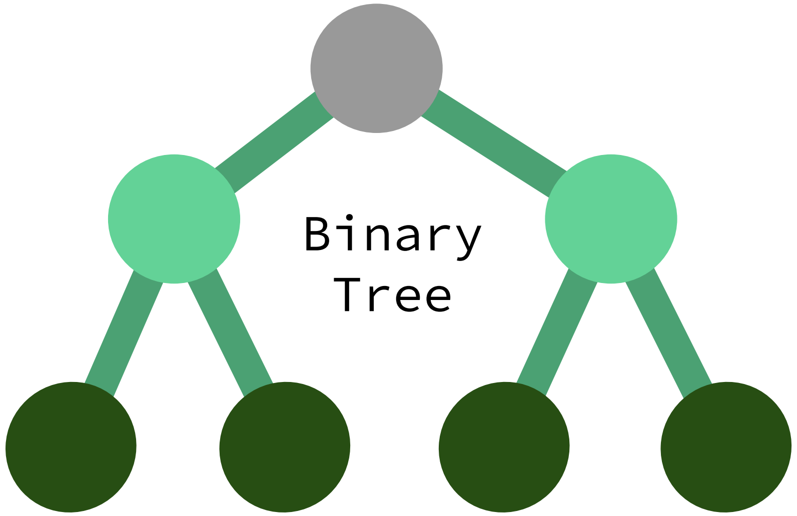


How do Segment Trees Work?

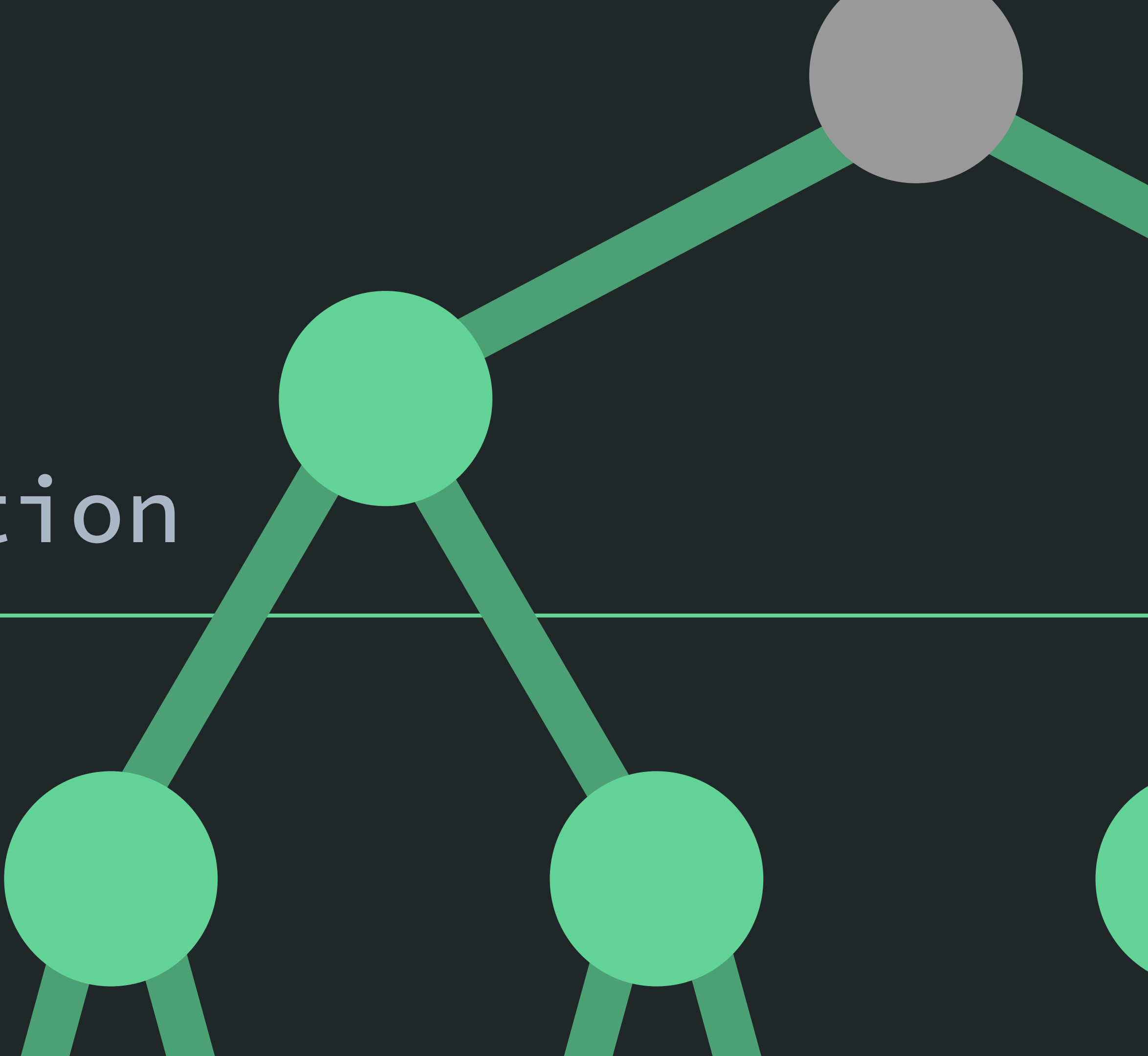


Details

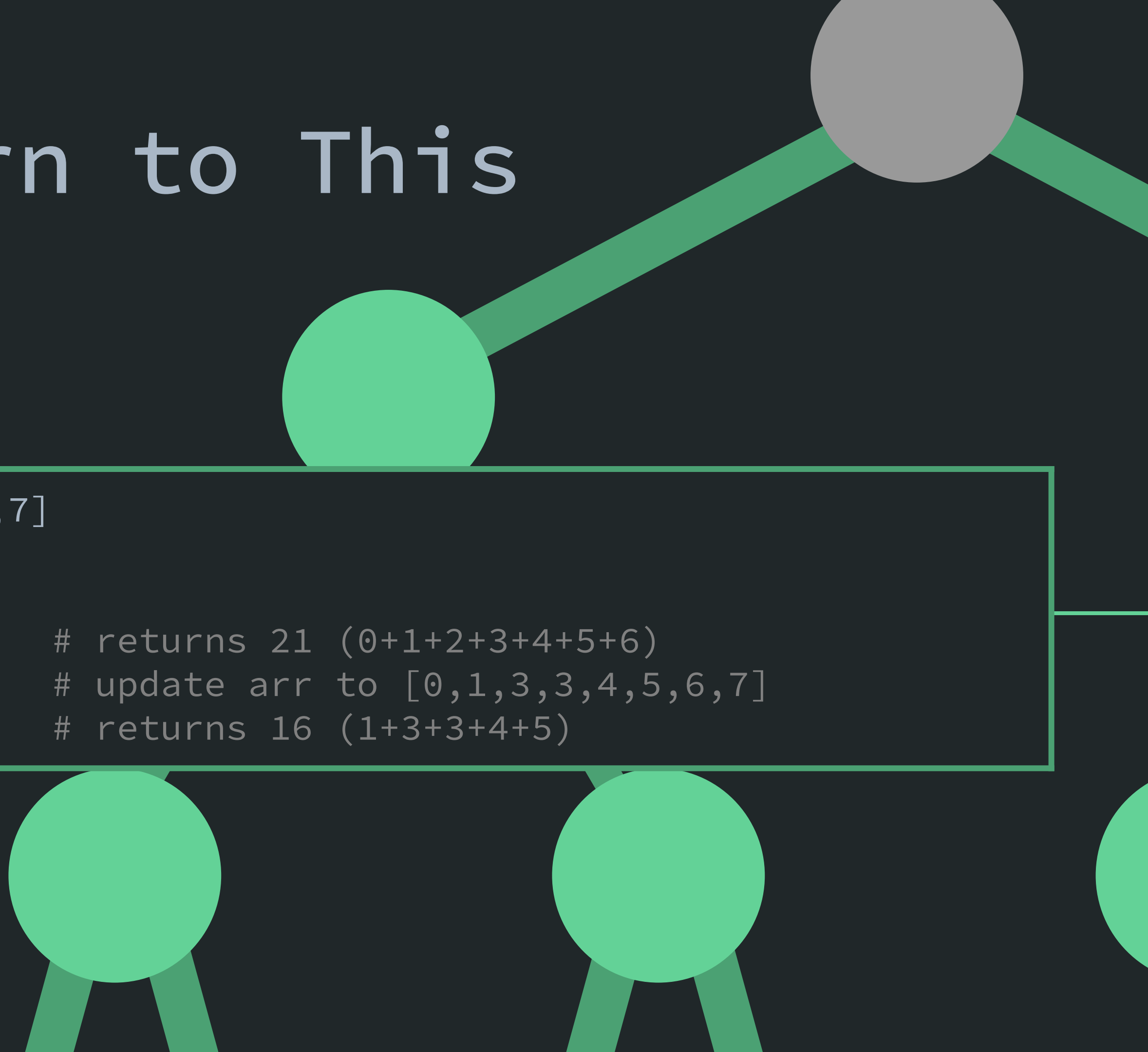
- Segment Trees are Binary Trees (parents only allowed 2 children)
- `# inner nodes = max # leaf nodes - 1`
- Leaf nodes contain the original array
- Each parent stores an operation's result between its children
  - Provides precomputed results over intervals of an array
- Can be stored in an array



Implementation

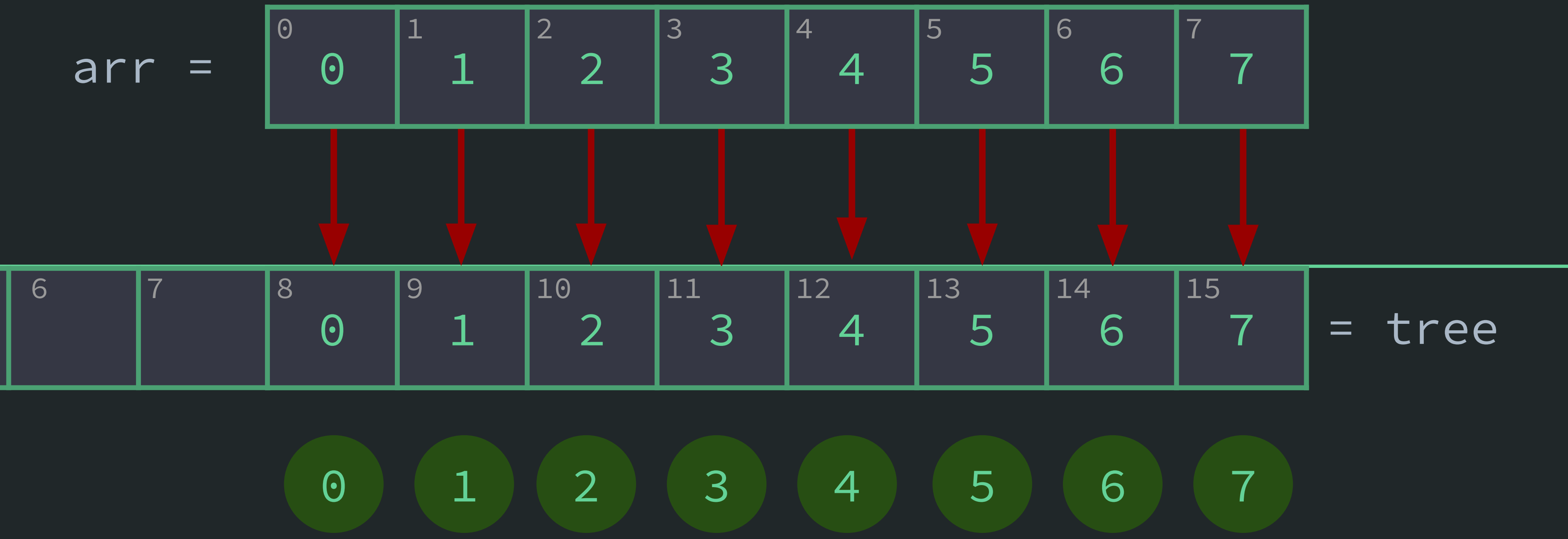


Let’s Return to This



```
arr = [0,1,2,3,4,5,6,7]

# Q = 2, U = 1
query(l=0, r=6)          # returns 21 (0+1+2+3+4+5+6)
update(i=2, val=1)       # update arr to [0,1,3,3,4,5,6,7]
query(l=1, r=5)          # returns 16 (1+3+3+4+5)
```

Construction - Leaf Nodes

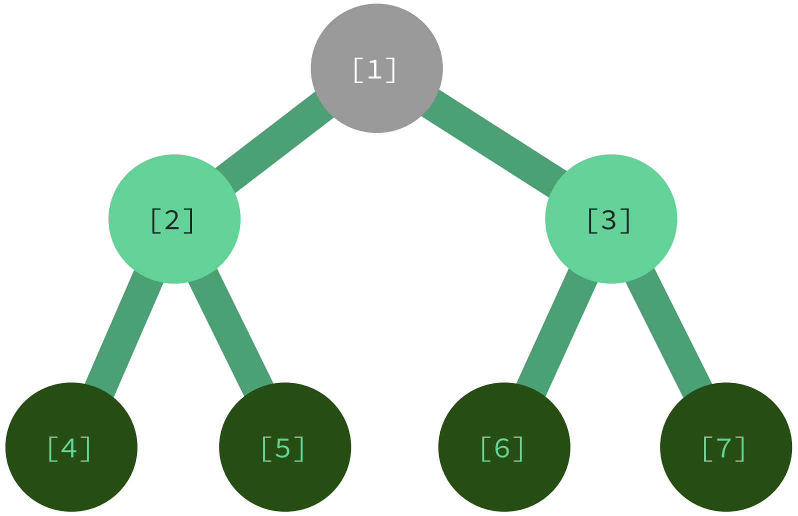


Implementation

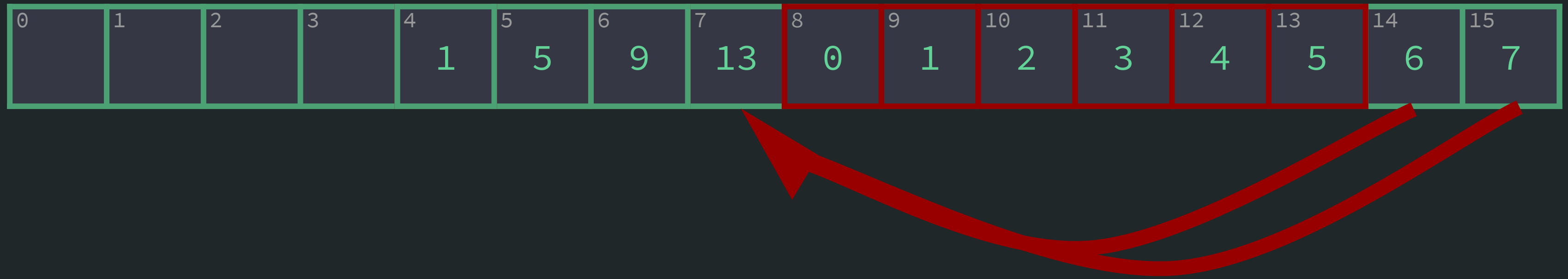
```
// Copy arr into the leaf nodes
for (int i = 0; i<arrLen; ++i) {
    tree[i+arrLen] = arr[i];
}

// Alternative
// System.arraycopy(arr, 0, tree, arrlen, arrlen);
```

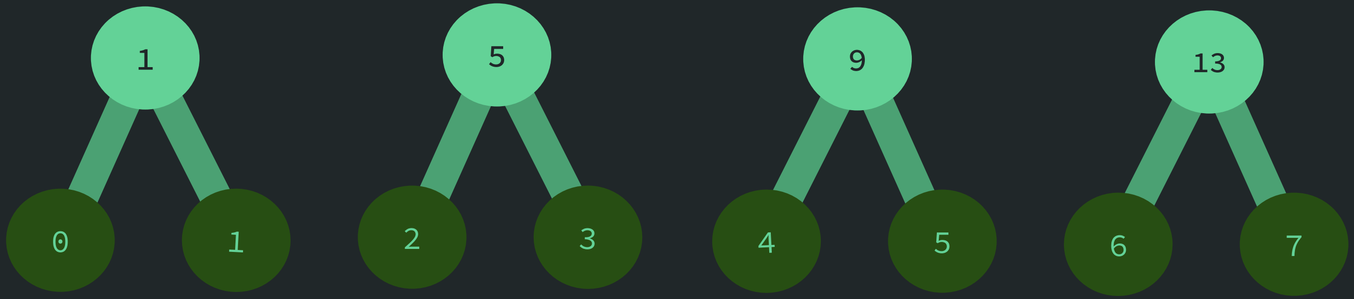
How do We Find a Node’s Children and Parent?

- Let i be the current node
- i’s children:
  - Left child: `2*i` (always even)
  - Right child: `2*i+1` (always odd)
- i’s parent:
  - `i/2` (java truncates decimals
    -> same index for left and right child)

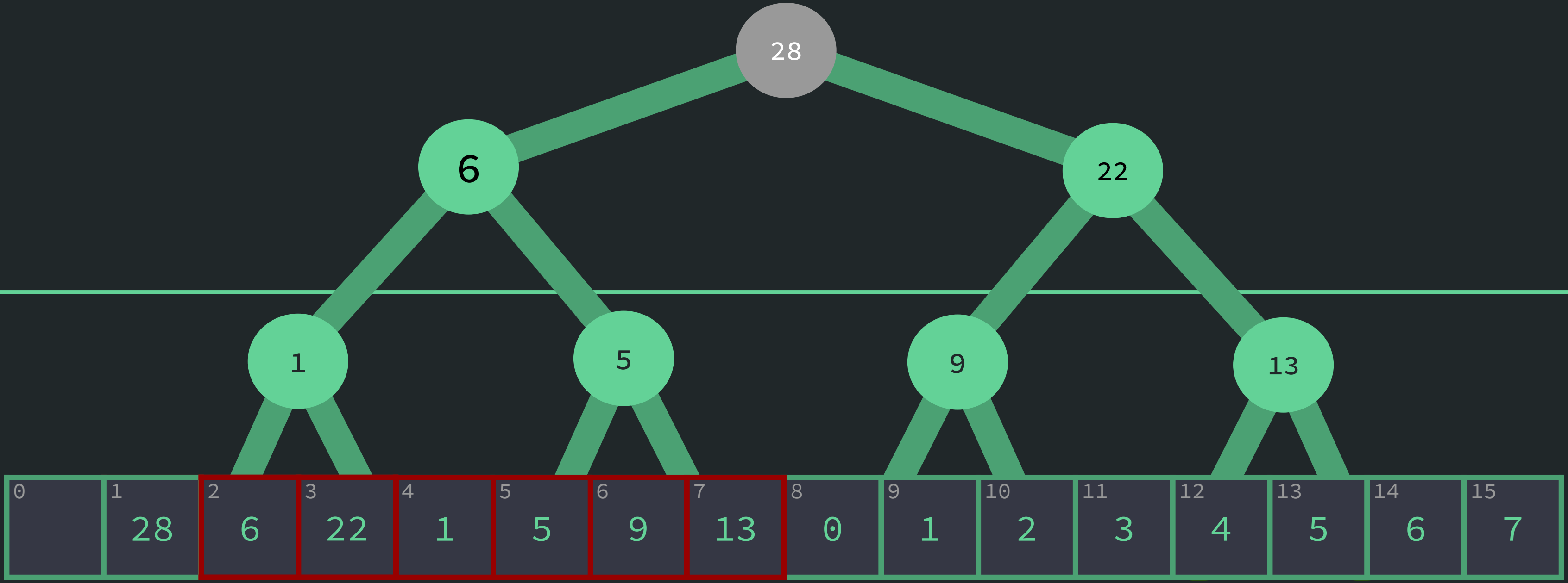


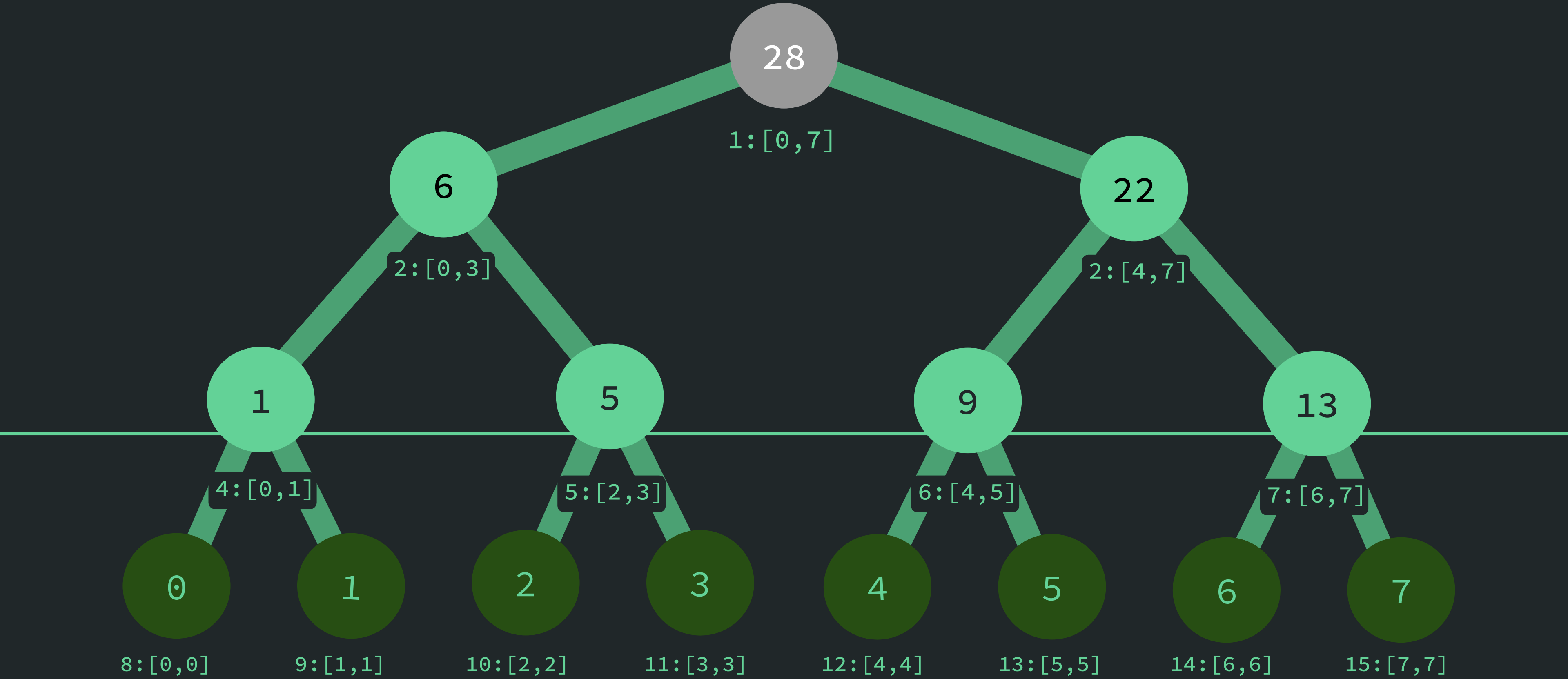
Construction - Inner Nodes





Construction - Inner Nodes




28
1:[0,7]
6
2:[0,3]
22
2:[4,7]
1
5
9
13
4:[0,1]
5:[2,3]
6:[4,5]
7:[6,7]
0
1
2
3
4
5
6
7
8:[0,0]
9:[1,1]
10:[2,2]
11:[3,3]
12:[4,4]
13:[5,5]
14:[6,6]
15:[7,7]

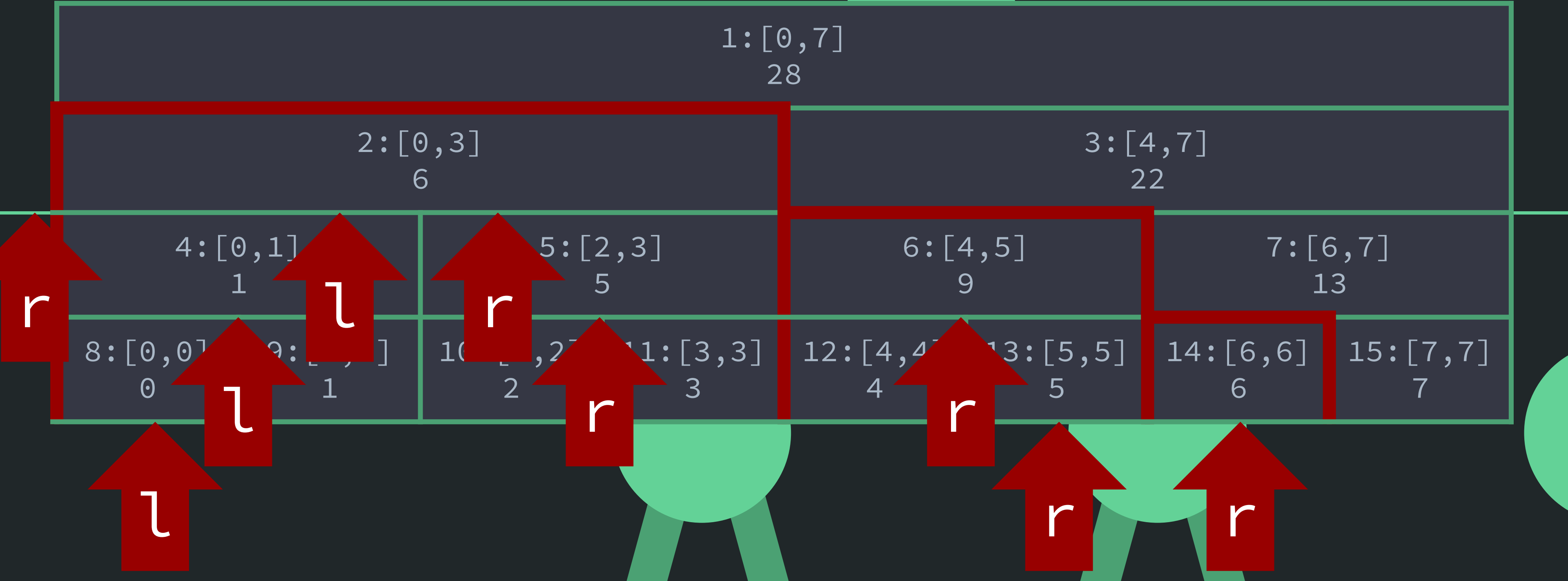
Implementation

```
void build {

    // Start at the last parent
    for (int i = arrLen-1; i>=1; --i) {

        // Parents are the sum of their children
        tree[i] = tree[2*i]+tree[2*i+1];

    }

}
```

Queries Part 1

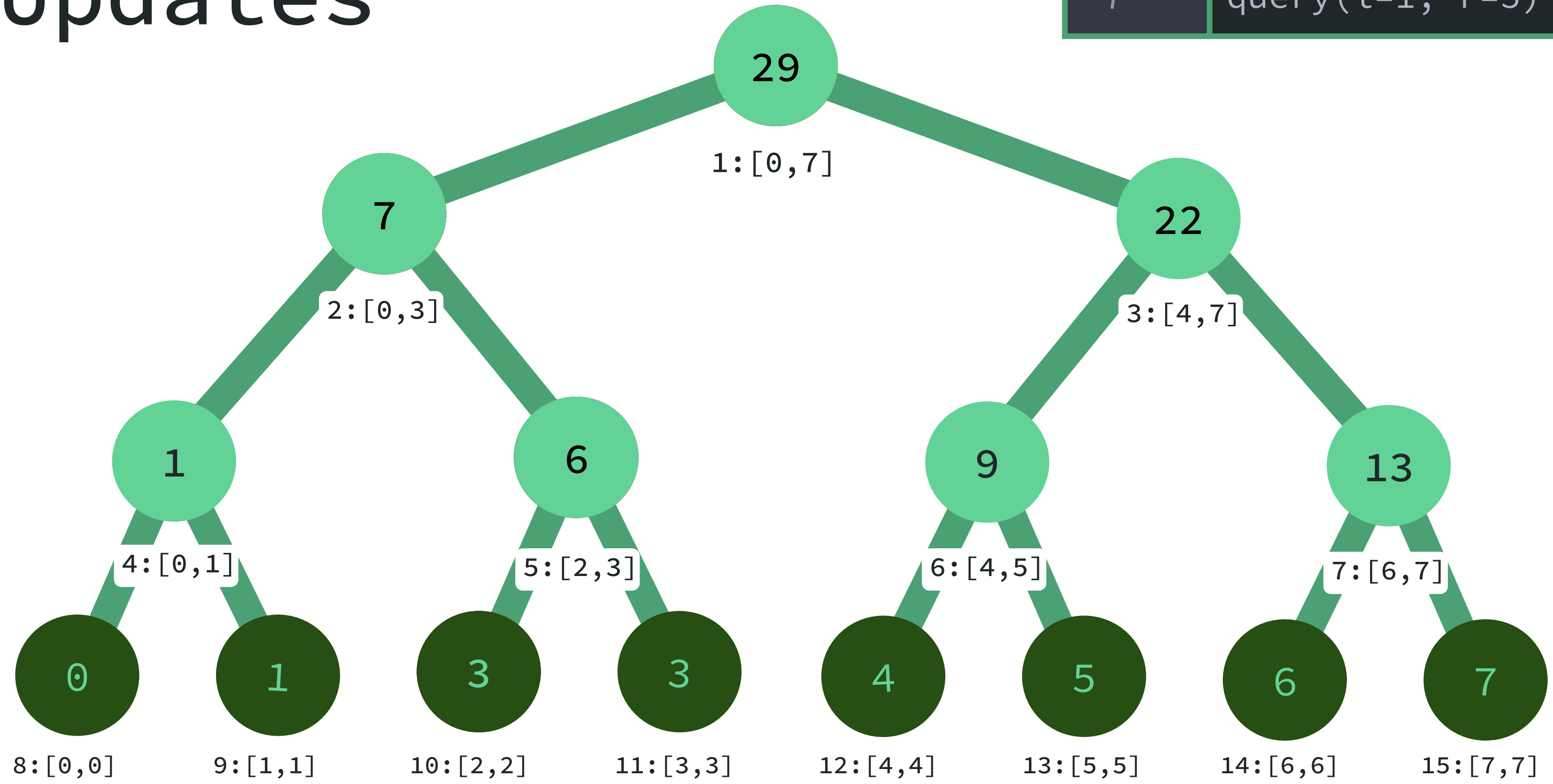
```
query(l=0, r=6)
update(i=2, val=1)
query(l=1, r=5)
```

sum = 05



Updates

```
query(l=0, r=6)
update(i=2, val=1)
query(l=1, r=5)
```



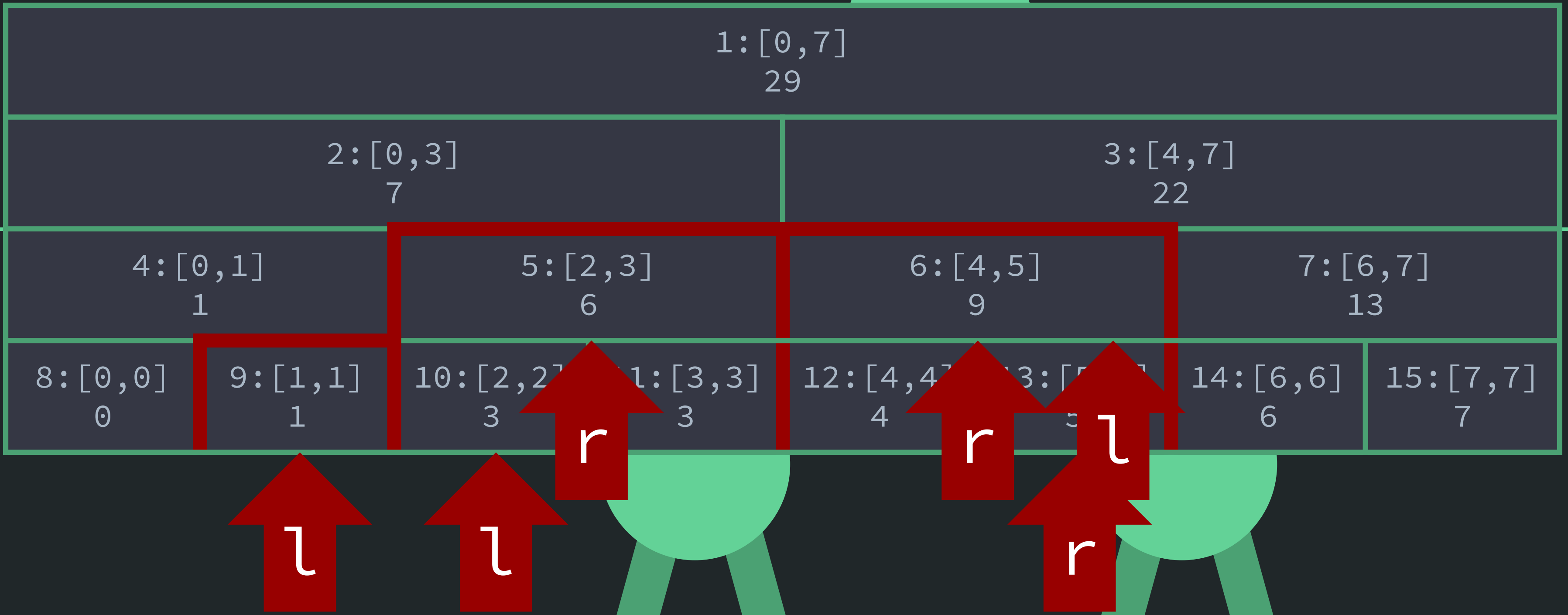
Implementation

```
void update (int i, int val) {

    i += arrLen;            // Increment i to its leaf node counterpart
    tree[i] += val;         // Update the leaf node
    i /= 2;                 // Go to i's parent

    // Bubble up to i's parents until the root node
    for (; i>=1; i /= 2) {
        // Update the parents
        tree[i] = tree[2*i]+tree[2*i+1];
    }

}
```

Queries Part 2

```
query(l=0, r=6)
update(i=2, val=1)
query(l=1, r=5)
```

sum = 16

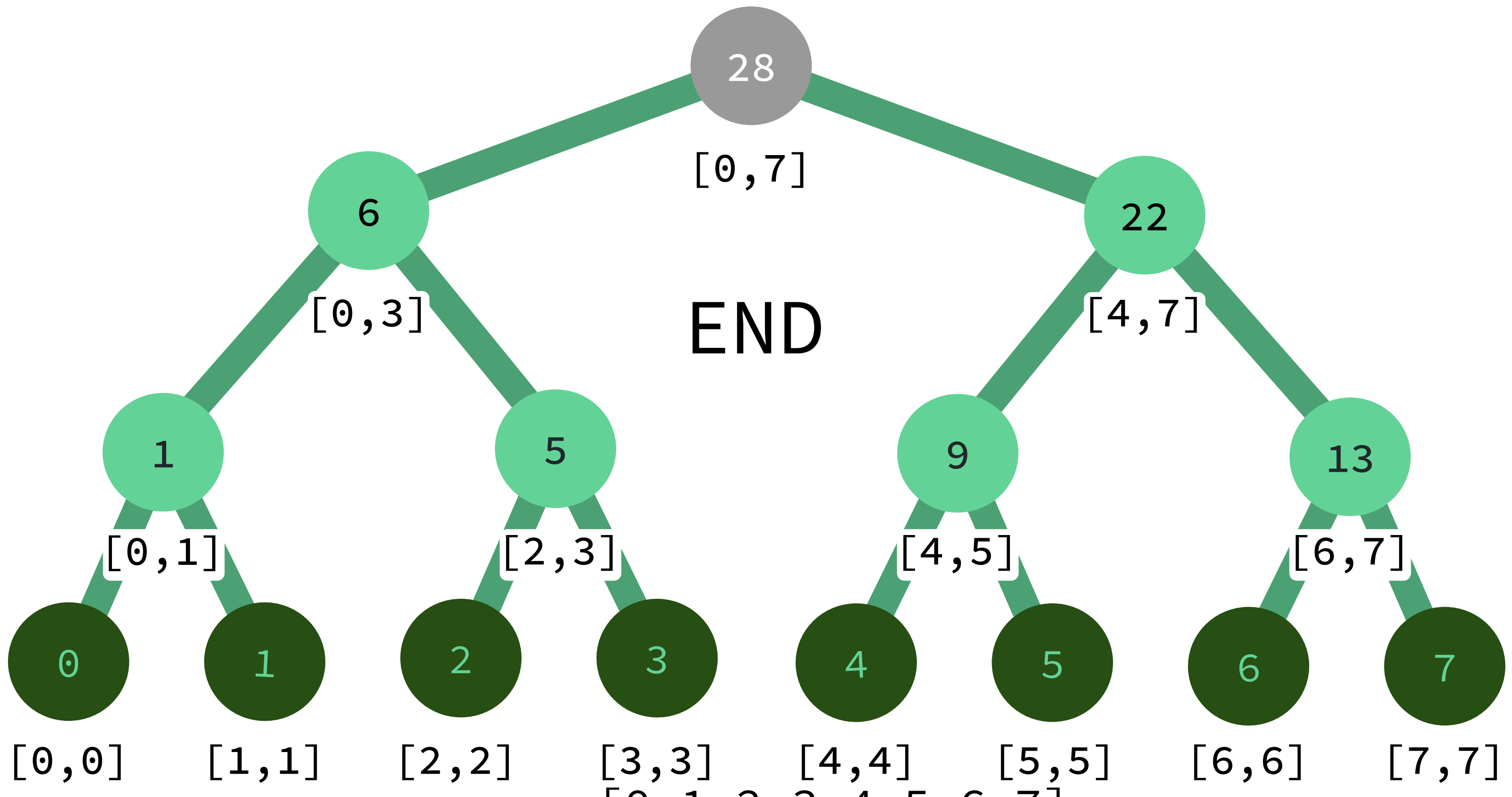


Implementation Part 1

```
int getSum (int l, int r) {

    // Increment l and r to their leaf node counterparts
    l += arrLen;
    r += arrLen;

    int sum = 0;
```

Implementation Part 2

```
    // While l and r are still left and right
    while (l<=r) {

        if (l%2==1) {               // If l is the right child of the parent
            sum += tree[l];         // Add the right child to the sum
            ++l;                    // Move l to the left child of the next branch
        }
        if (r%2==0) {               // If r is the left child of its parent
            sum += tree[r];         // Add the left child to the sum
            --r;                    // Move r to the right child of the next branch
        }
        l /= 2, r/=2;               // Traverse to their parents

    }

    return sum;

}   // End of function
```



arr = [0,1,2,3,4,5,6,7]

Further Reading

Iterative Implementation

Recursive Implementation

Lazy Propagation (Range Updates)

Get Grepper rn
Grepper - Chrome Web Store (google.com)

Problems

ICHB Selection Contest '17 Problem 3 - Parallel Universe

Dynamic Range Minimum Test

Mock CCO '18 Contest 3 Problem 4 - Roger Solves A Classic Rage Tree Problem

Challenge:
DMOPC '19 Contest 2 P3 - Selection

CCC '21 S5 - Math Homework