Heavy-Light Decomposition

COMP 4991: Advanced Problem Solving

Micah Stairs

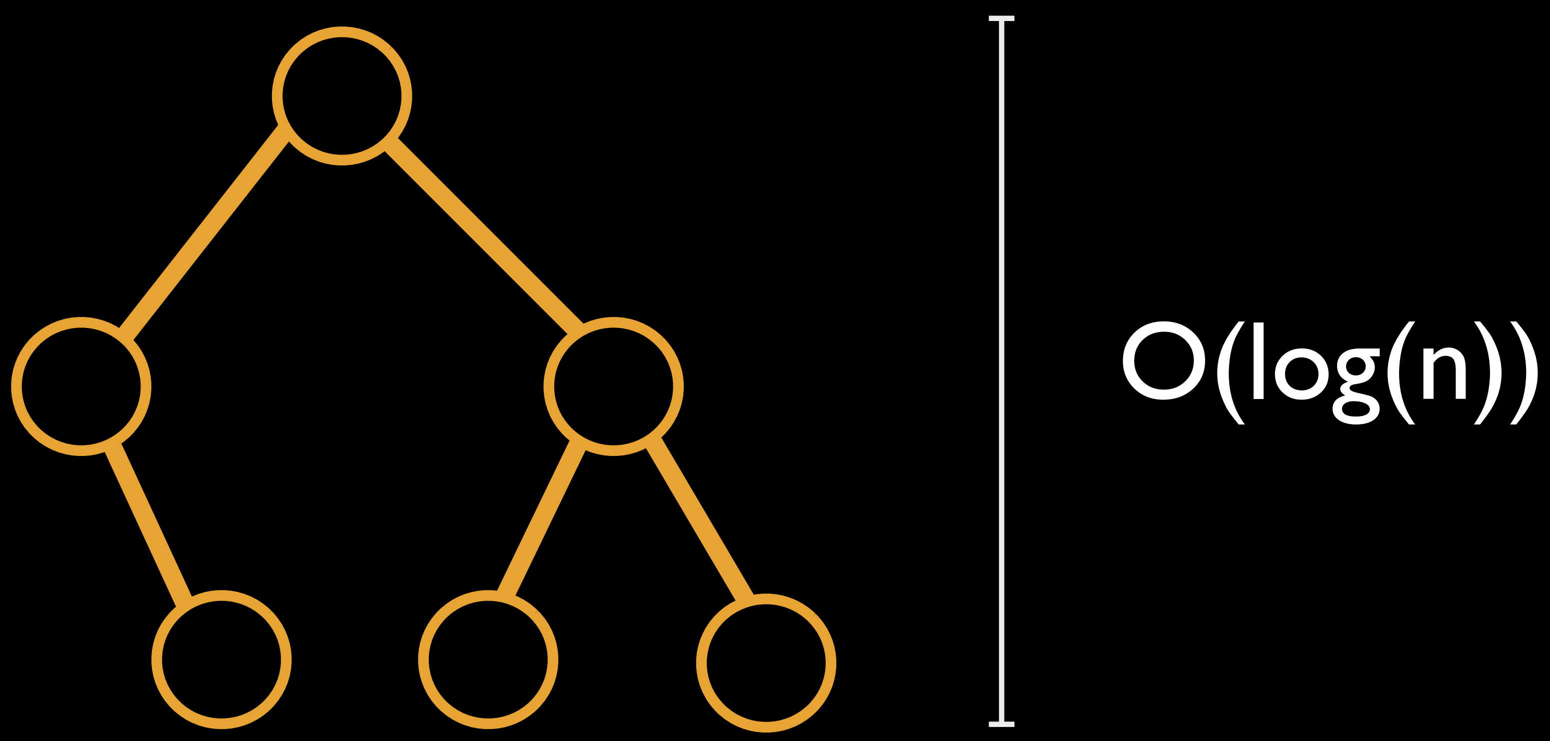
Outline

- Heavy-Light Decomposition
  - Motivation
  - Explanation
  - Construction
  - Queries
  - Kattis Problem (Tourists)

Heavy-Light Decomposition (HLD)

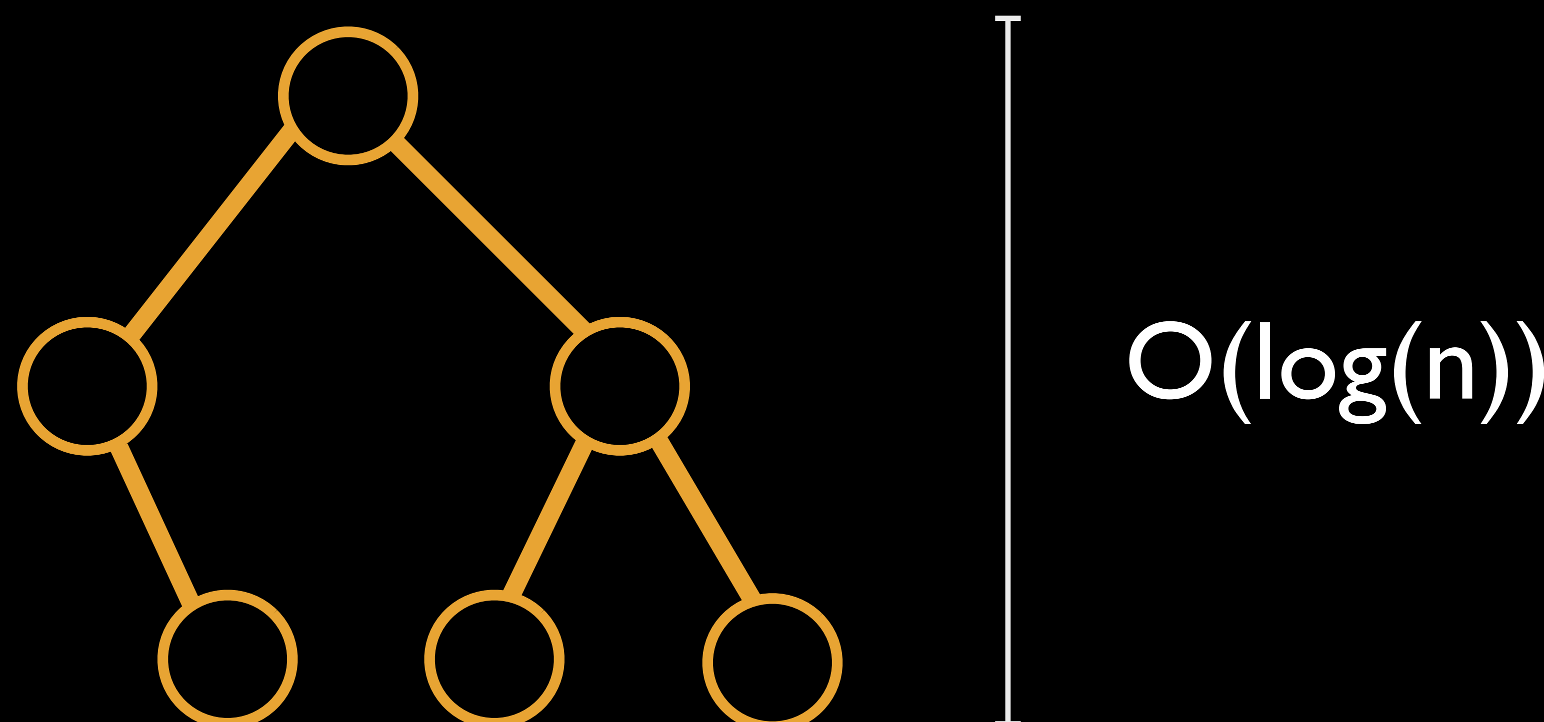
Motivation

- Trees are commonly used.
- Balanced trees have many nice O(log(n)) properties.



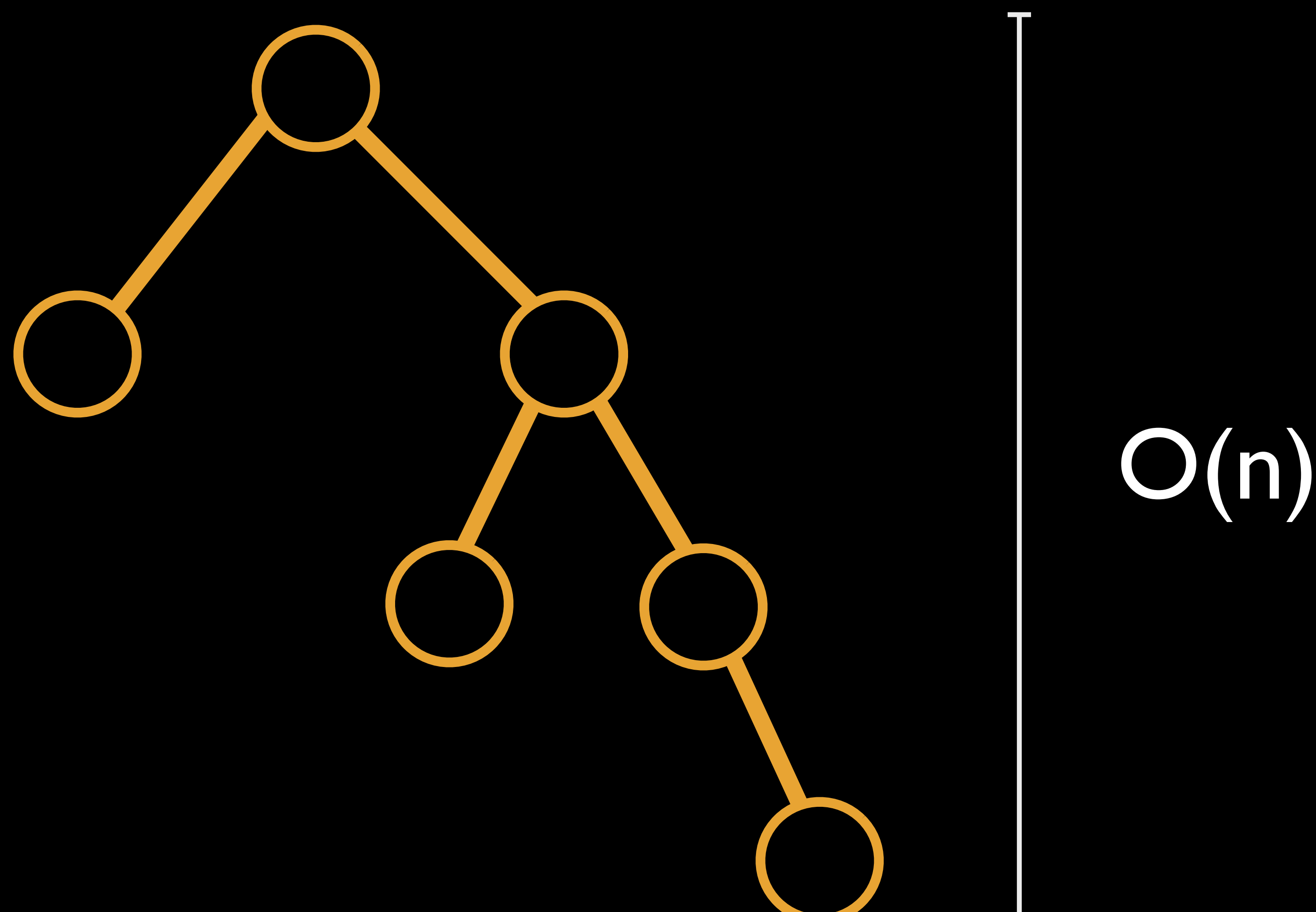
Motivation

- We may want to do queries such as determining the *Least Common Ancestor* of or the *distance* between two nodes.



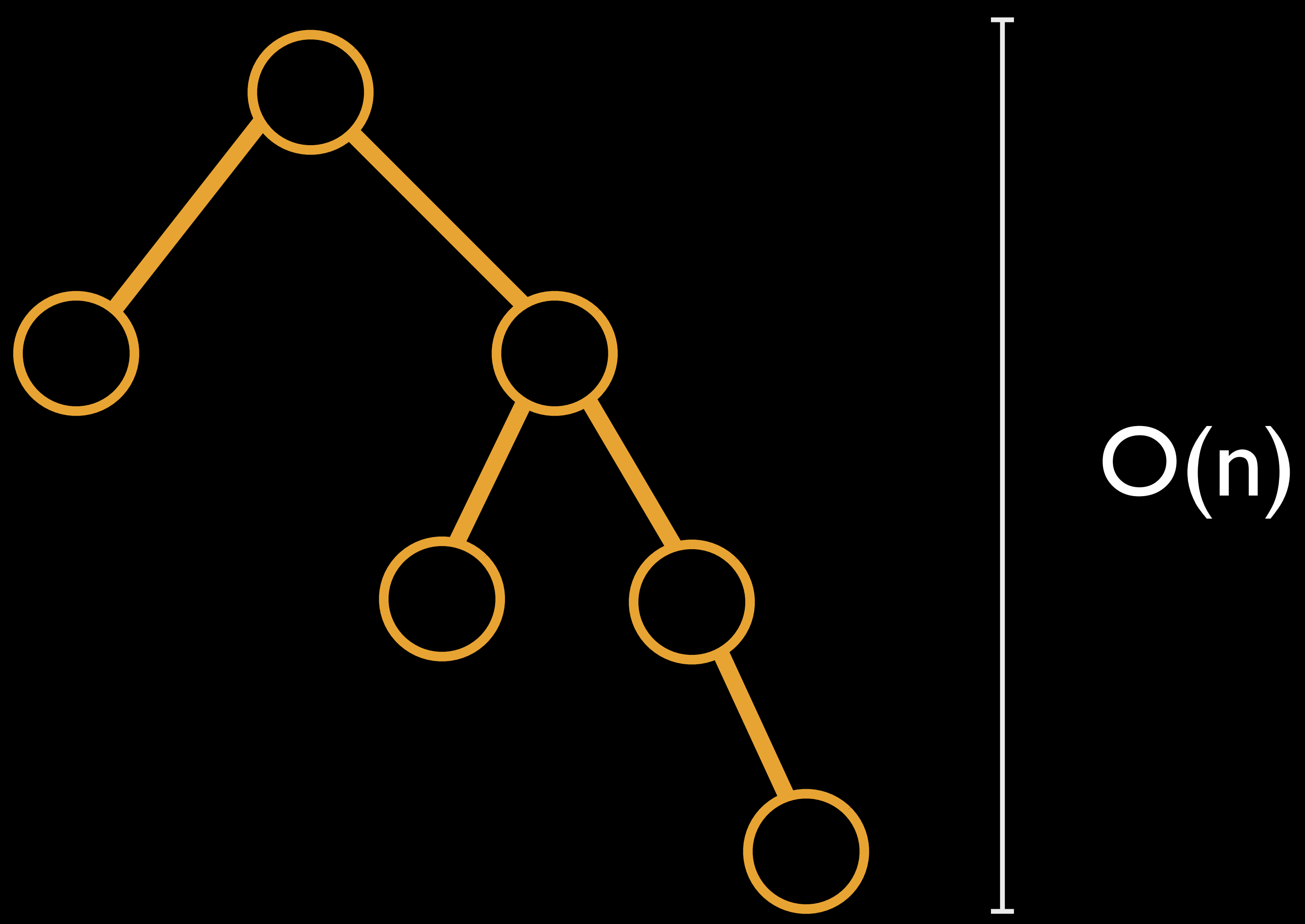
Motivation

- Unbalanced trees can degenerate to a linked list, giving O(n) properties.



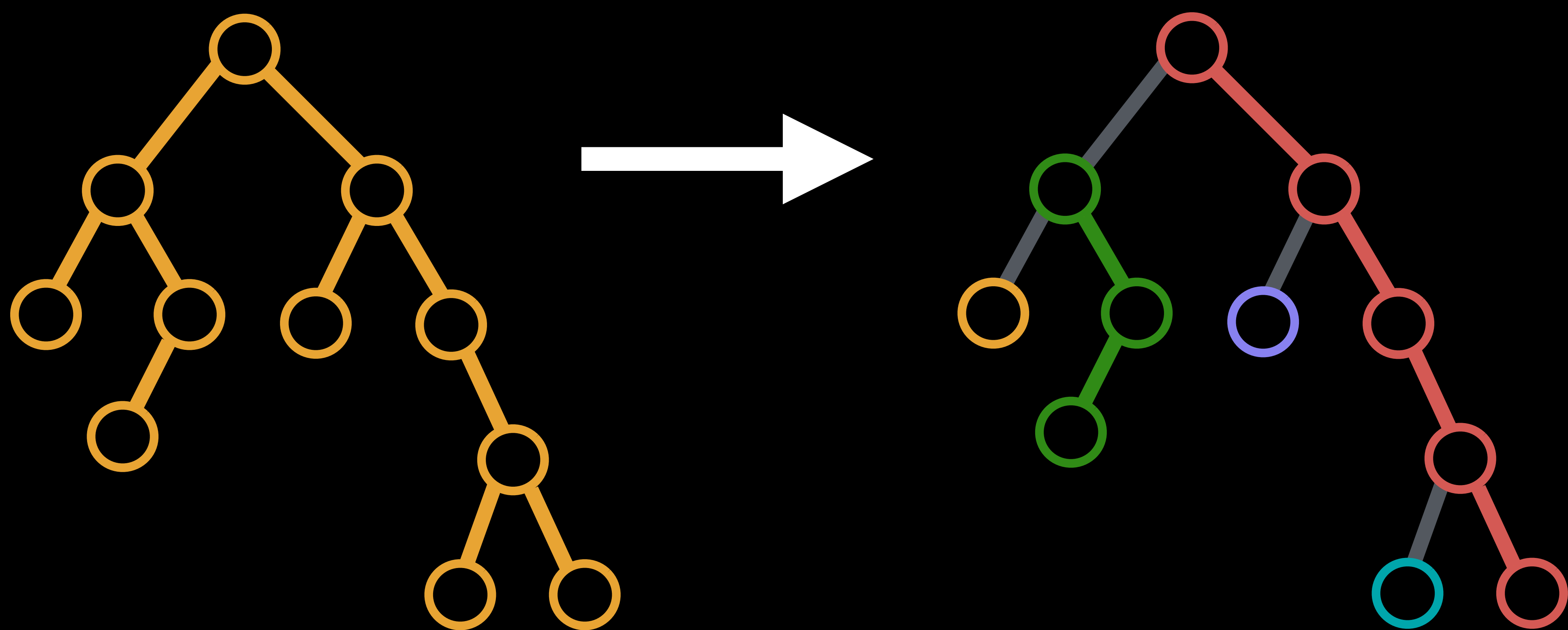
Motivation

- We want to give unbalanced trees logarithmic properties.



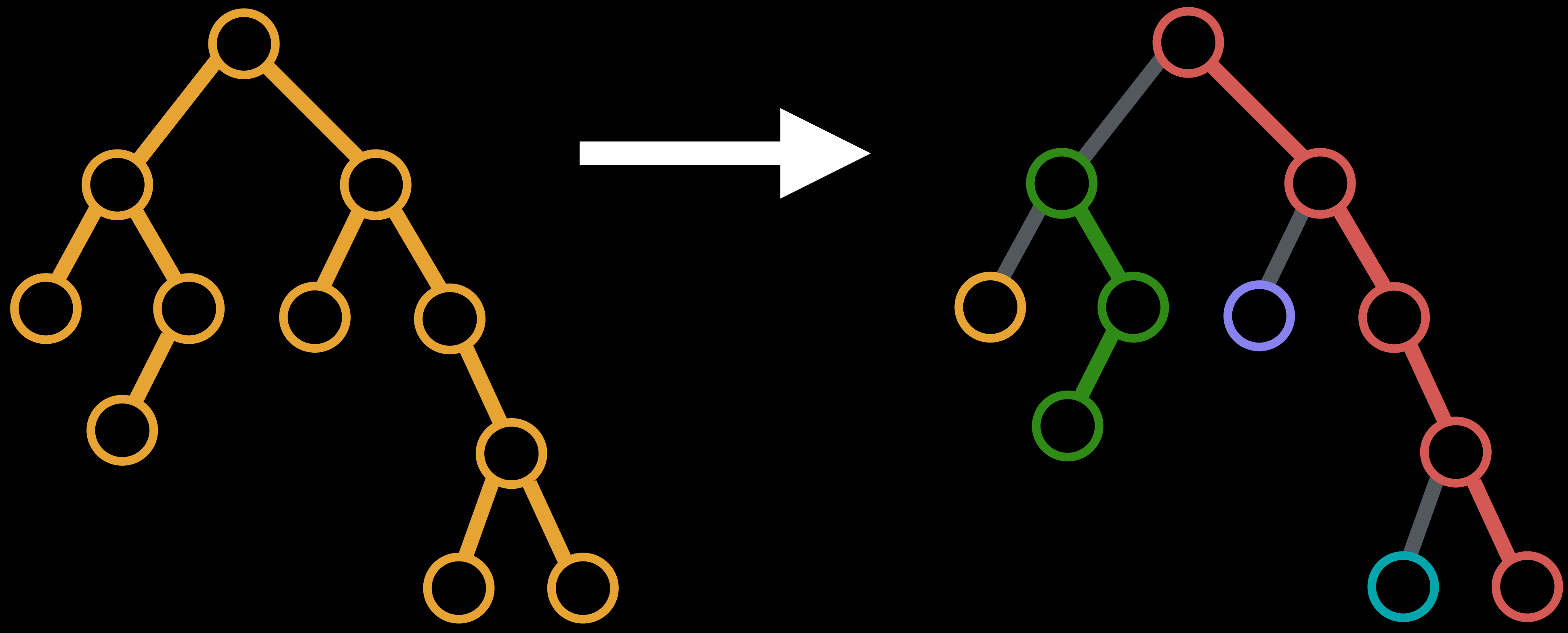
Explanation

- To do this we will split up an unbalanced tree into chains.



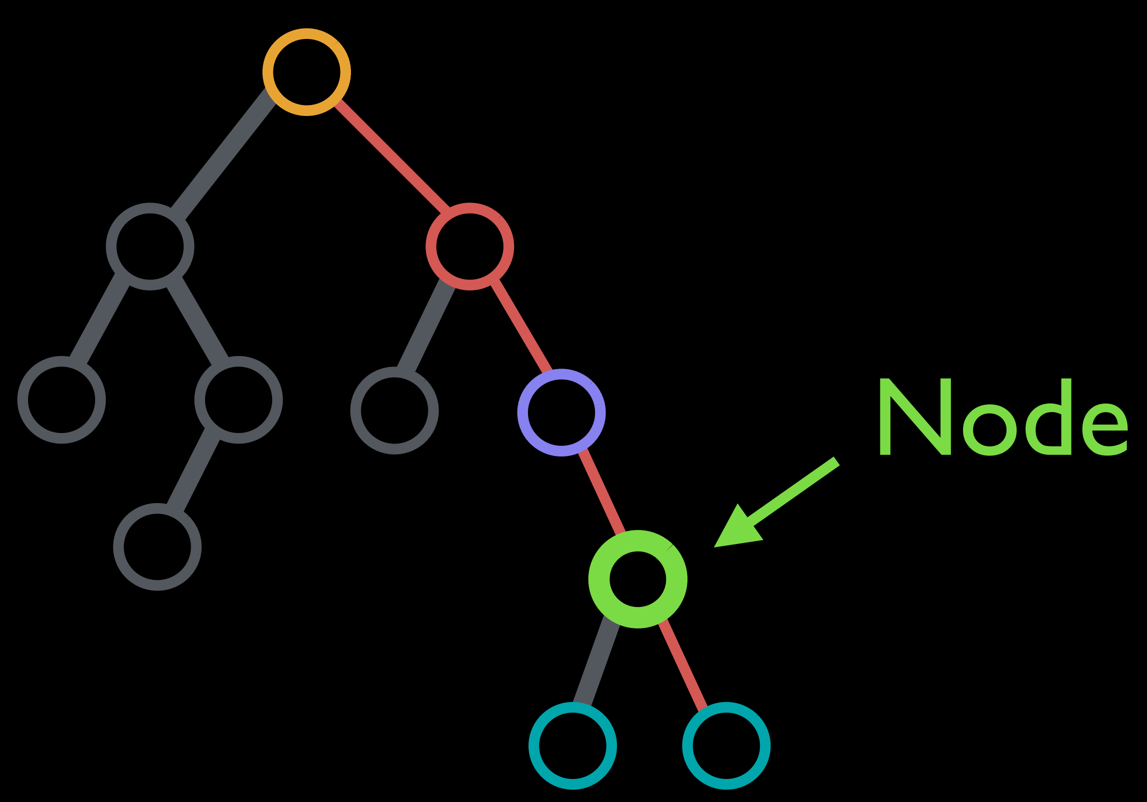
Explanation

- This chains will be constructed so that there are O(log(n)) chains in any path from the root to a leaf.



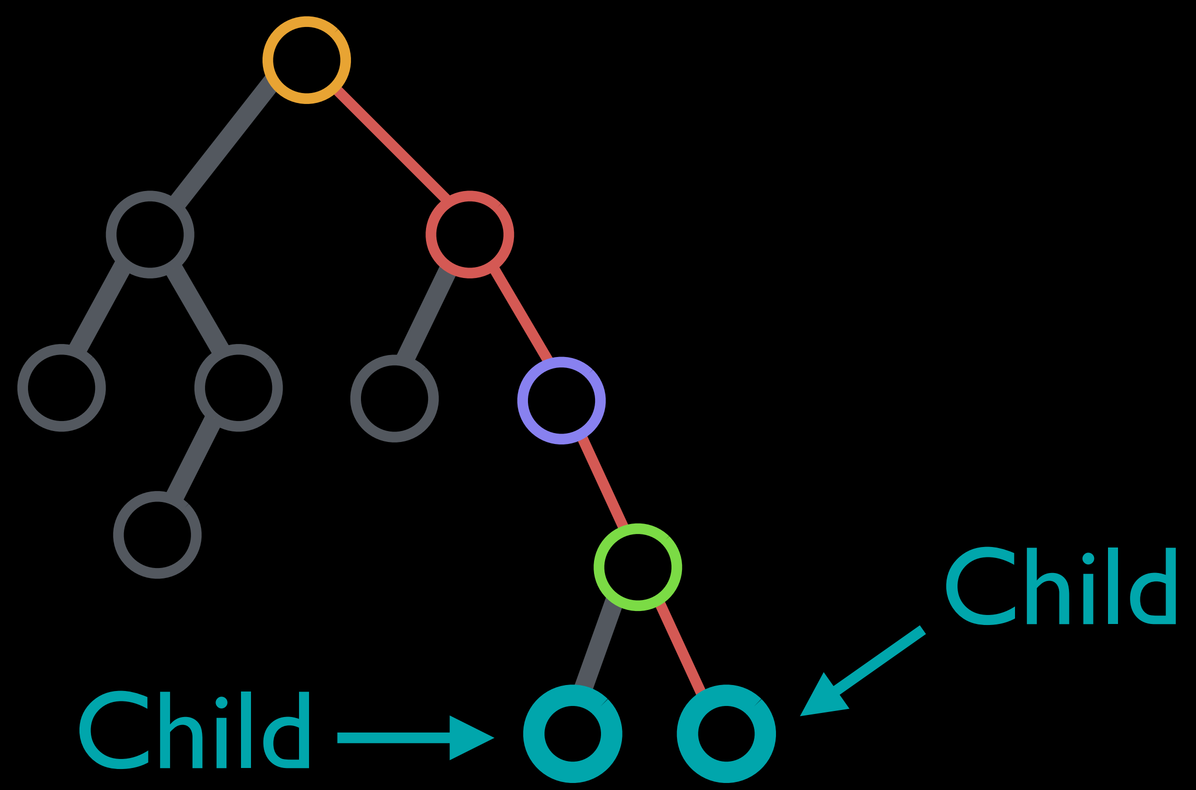
Explanation

- Each **node** has a reference to its list of children, its parent, and its head of the chain.



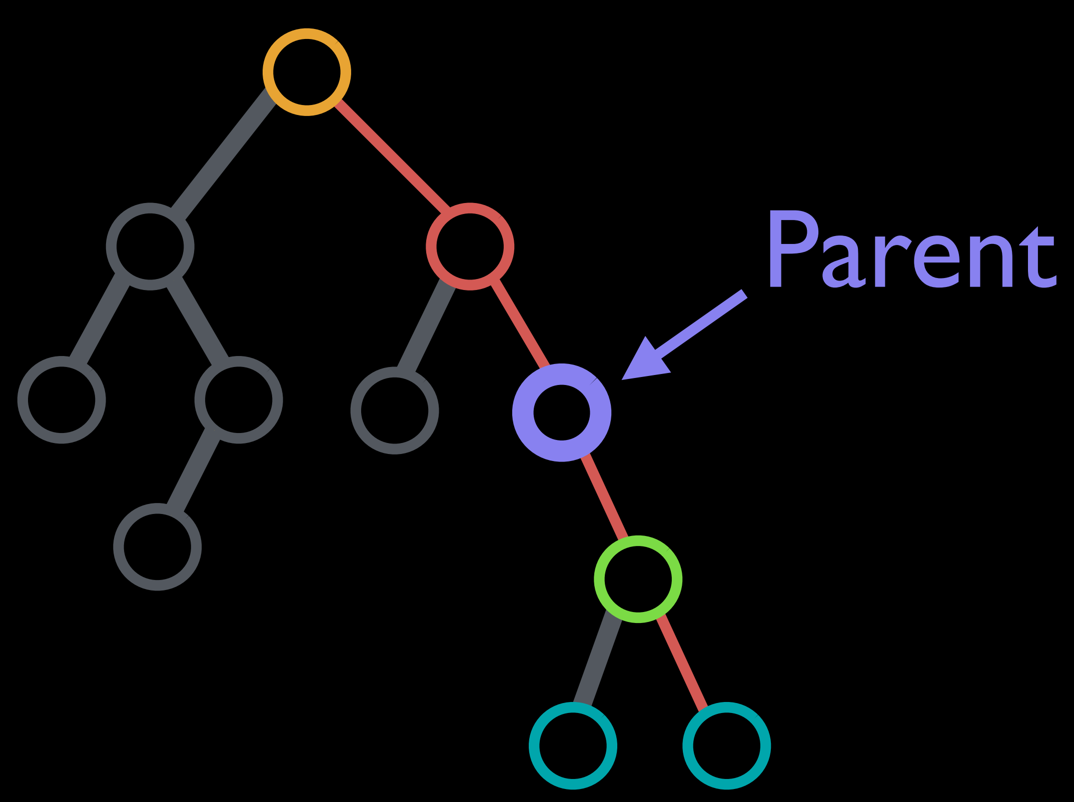
Explanation

- Each node has a reference to its list of **children**, its parent, and its head of the chain.



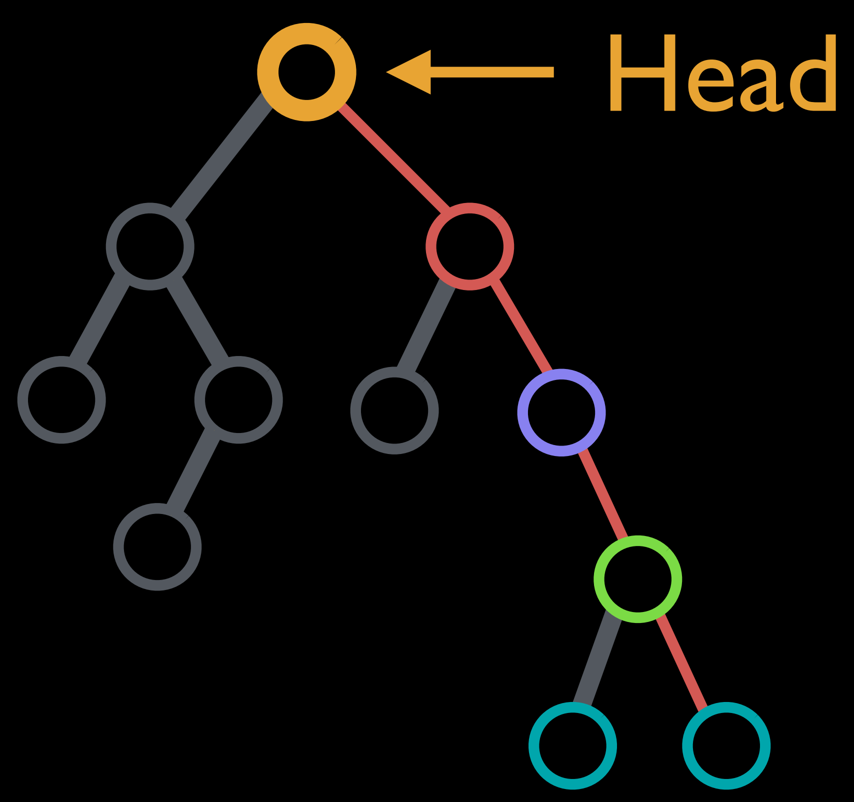
Explanation

- Each node has a reference to its list of children, its **parent**, and its head of the chain.



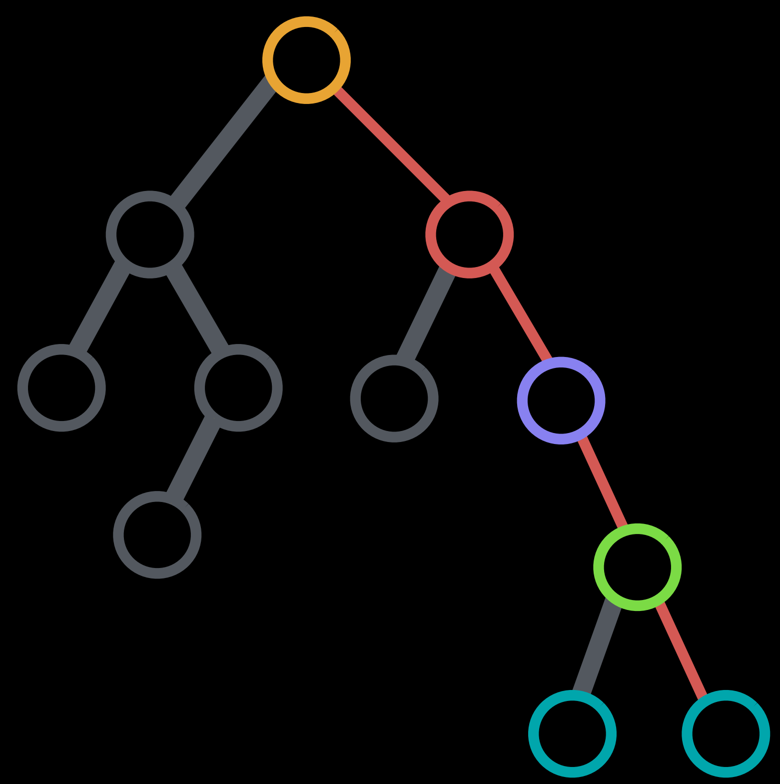
Explanation

- Each node has a reference to its list of children, its parent, and its **head** of the chain.



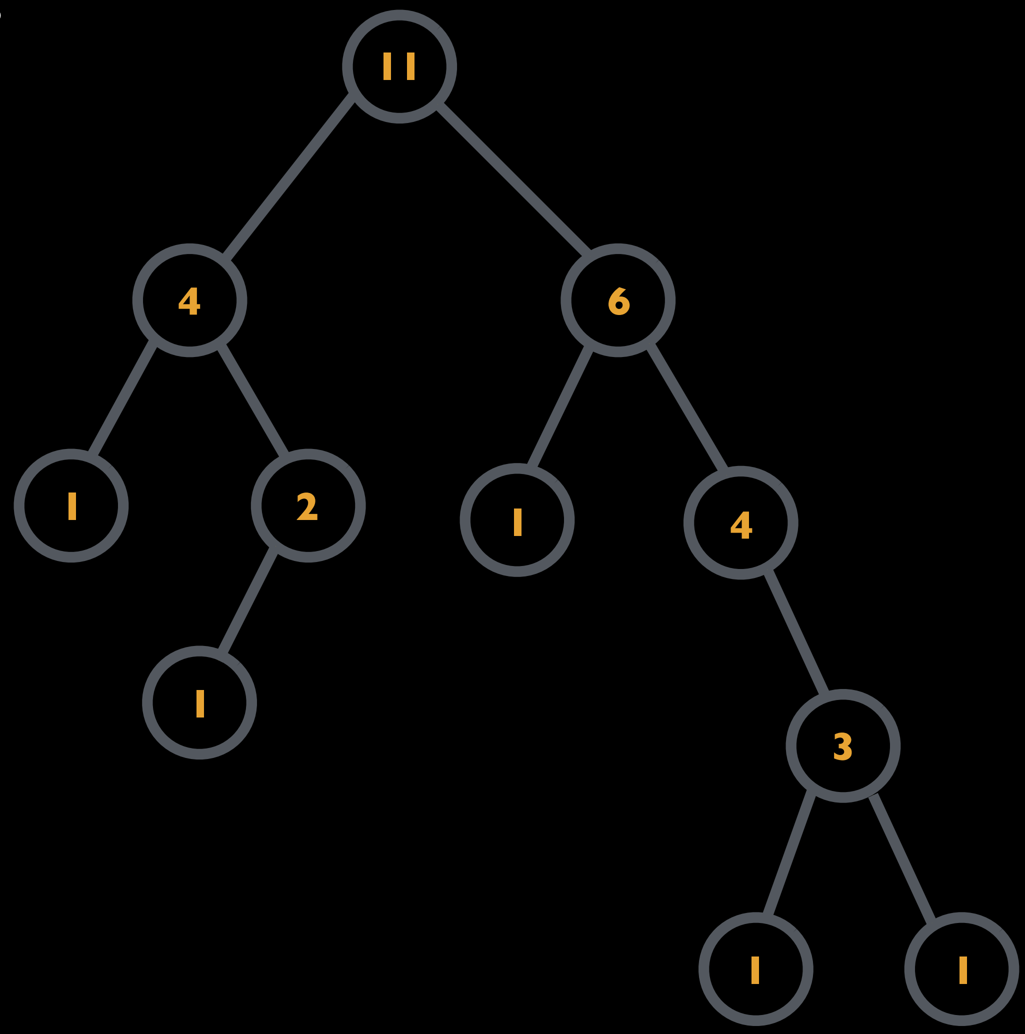
Explanation

- Each node has a reference to its list of children, its parent, and its head of the chain.

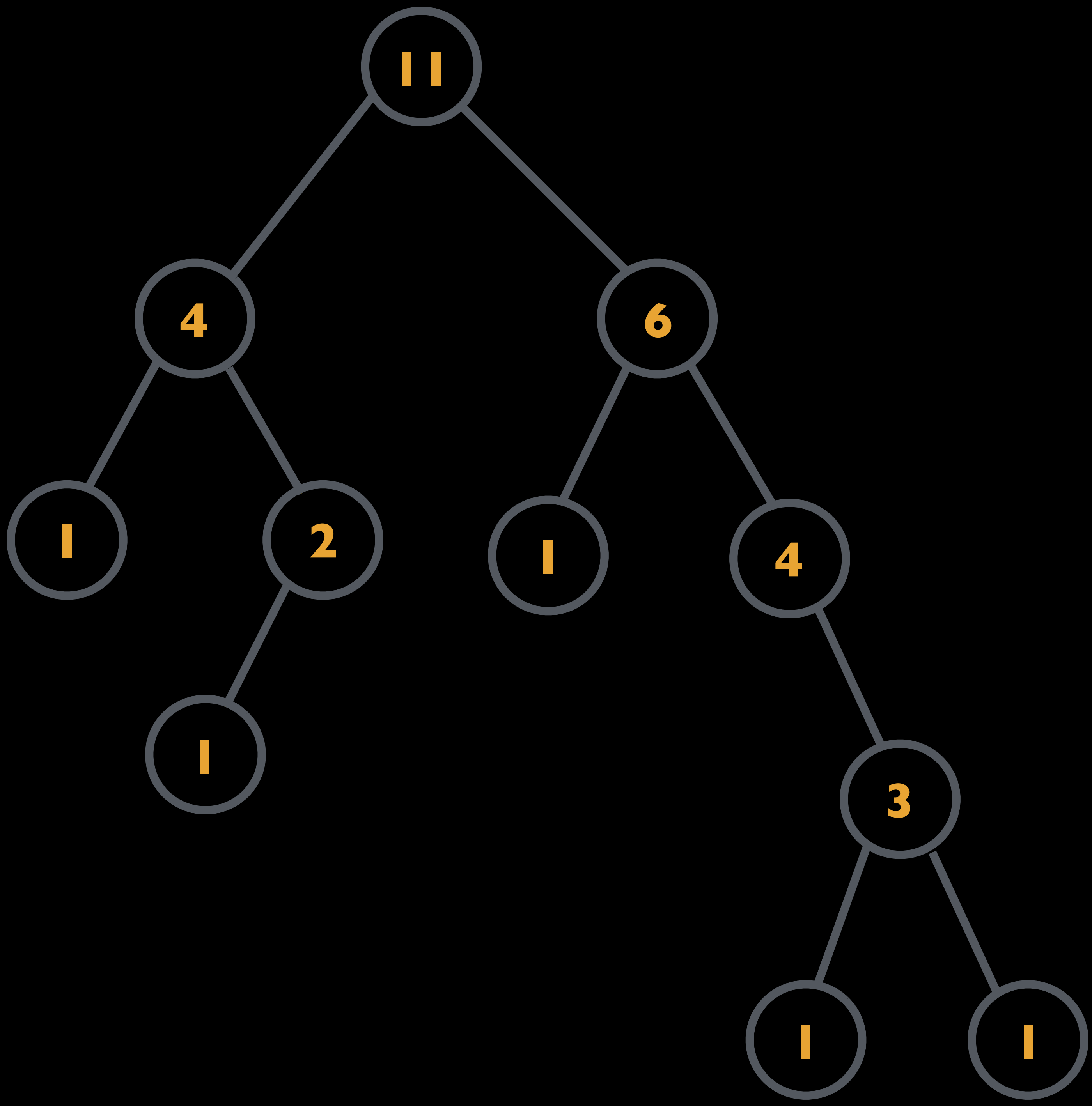


Construction

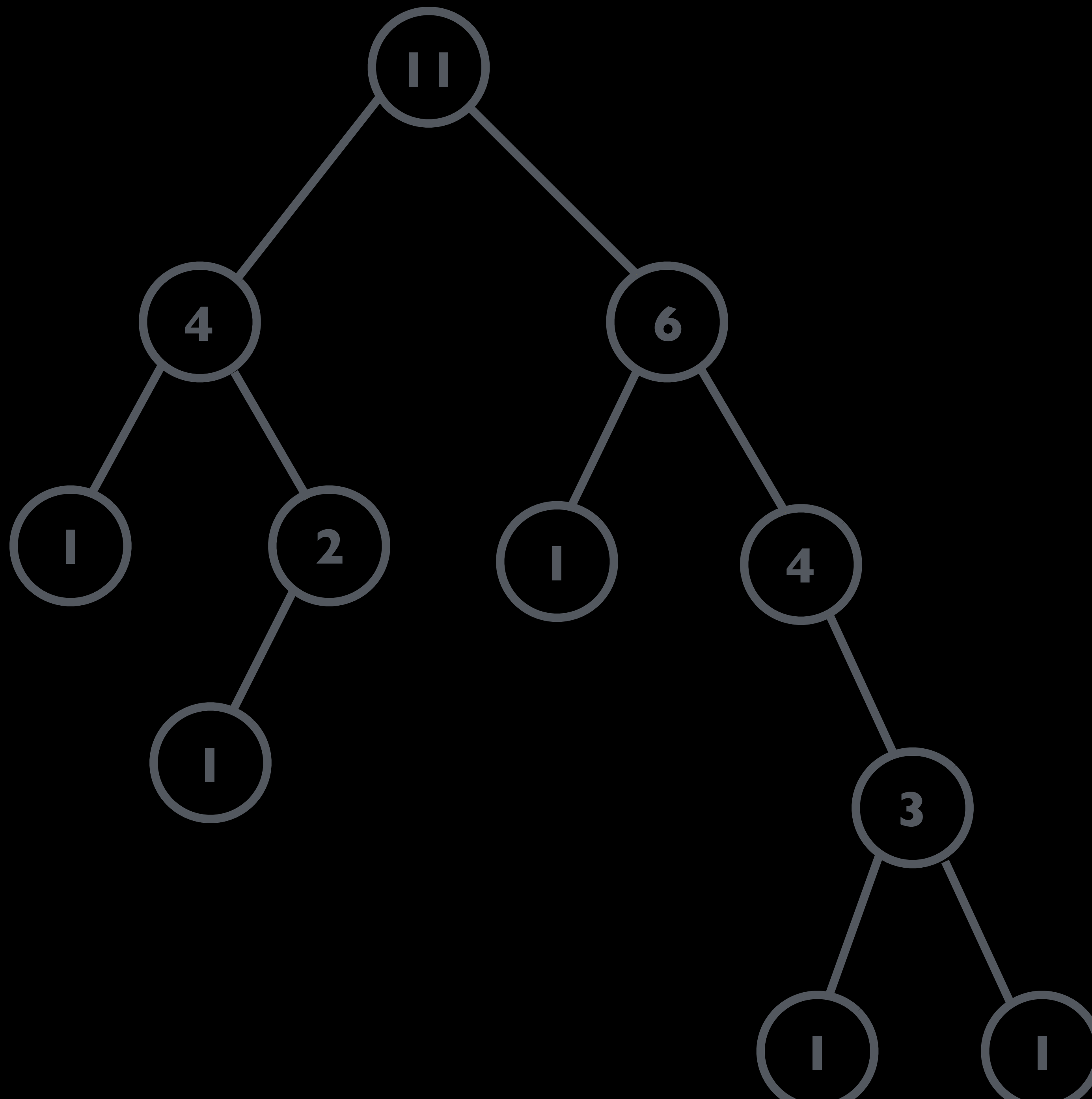
- Chains follow the heaviest child, so we must first pre-compute the size of each subtree.



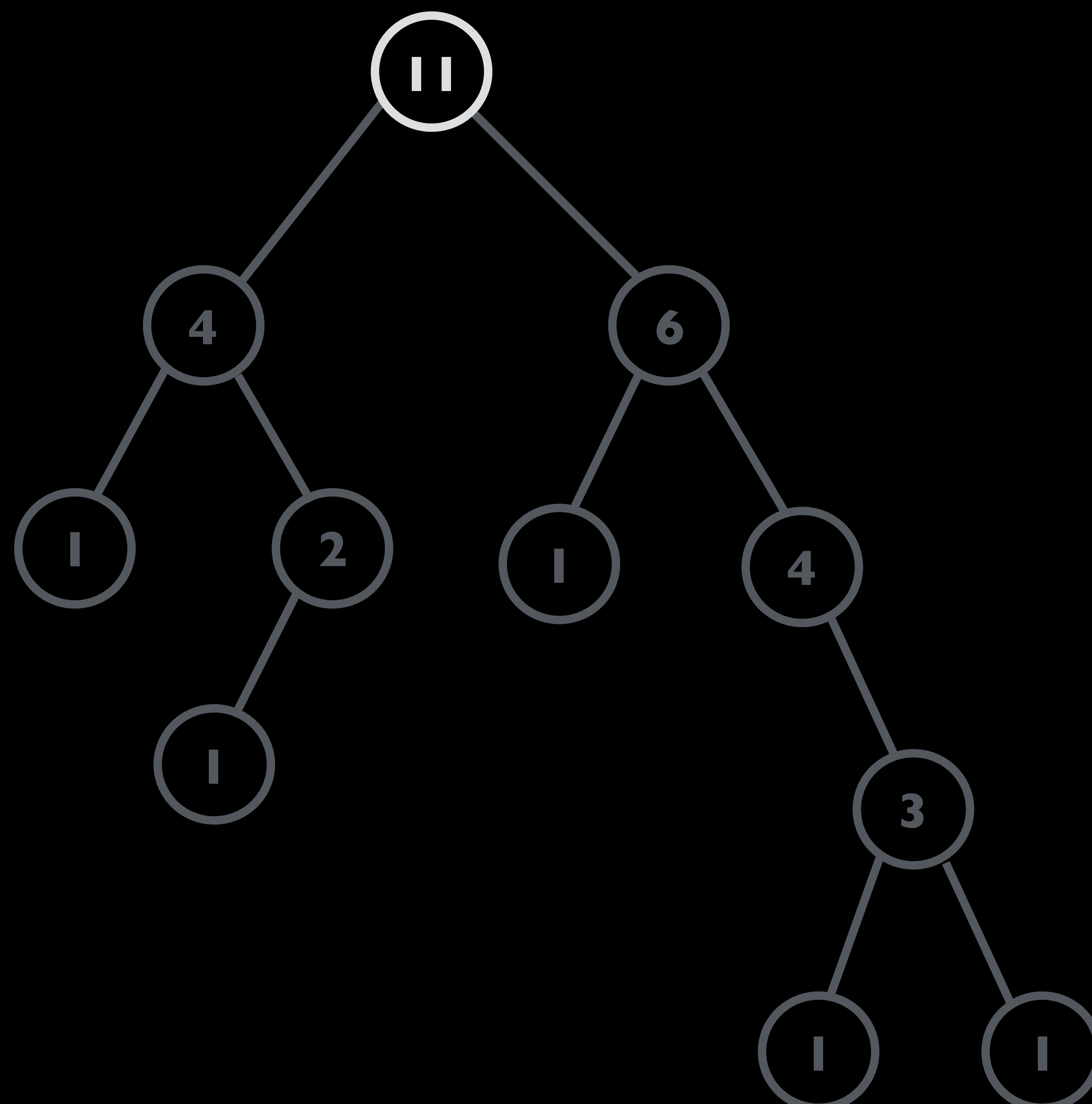
Construction



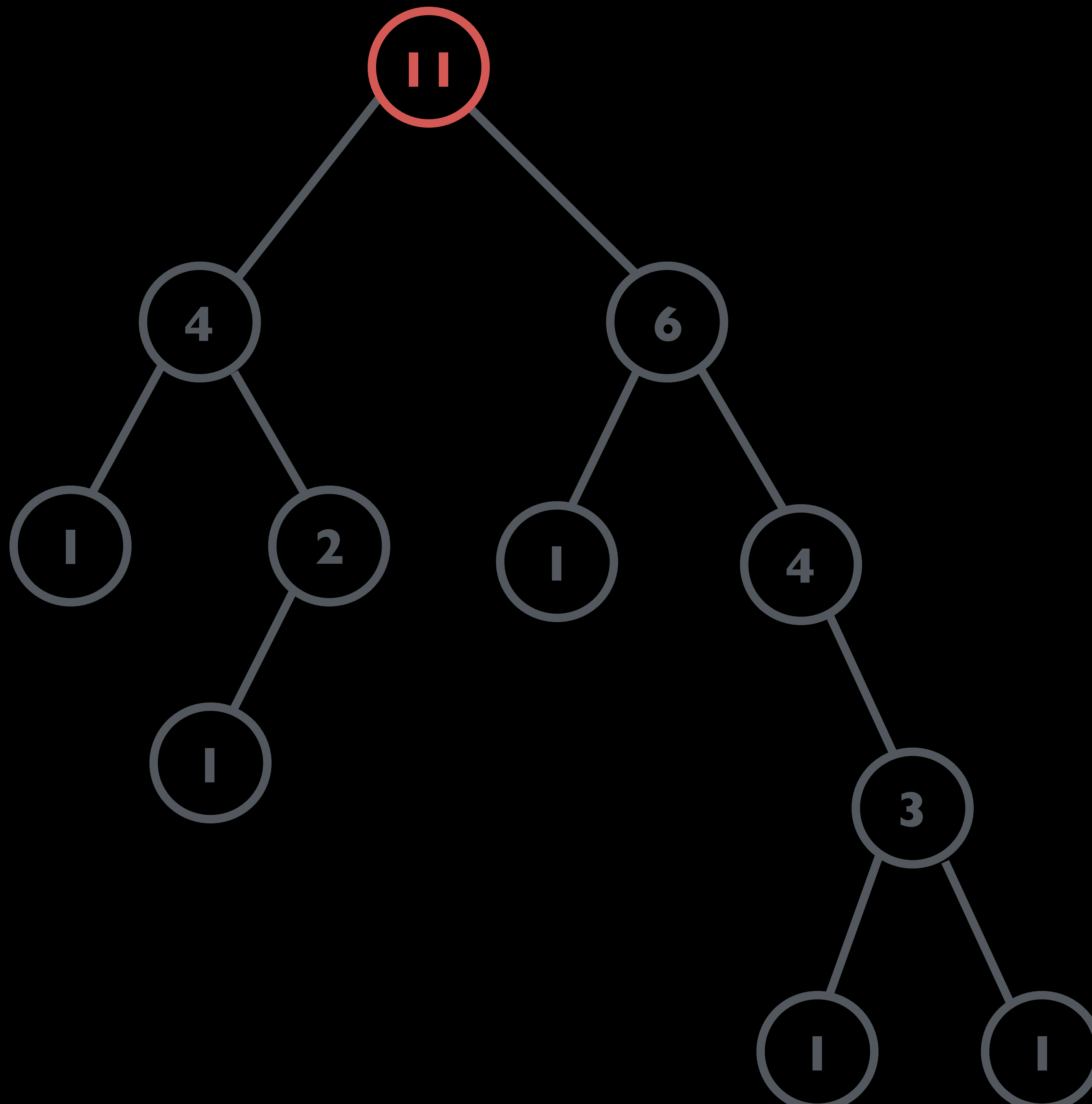
Construction



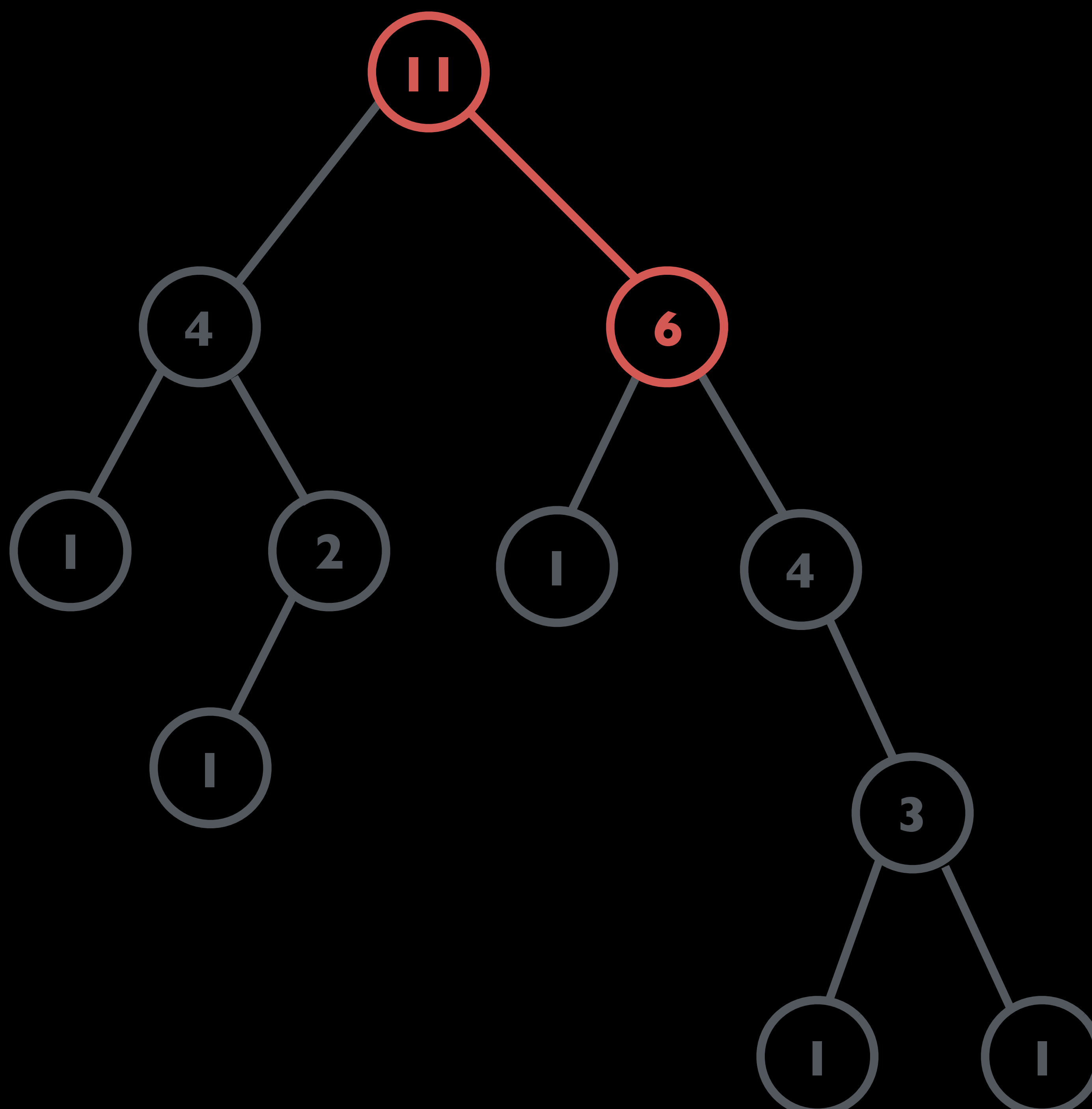
Construction



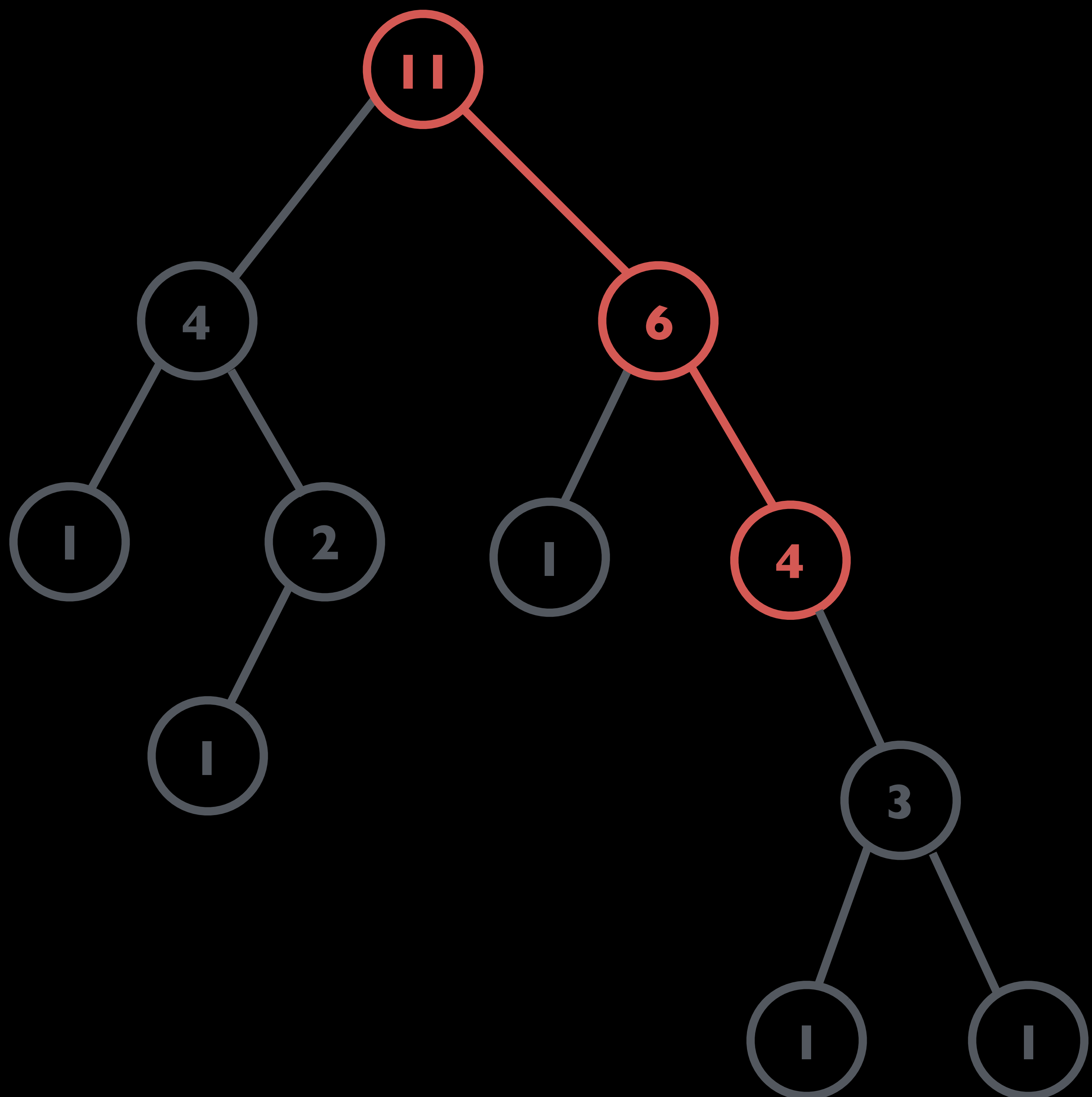
Construction



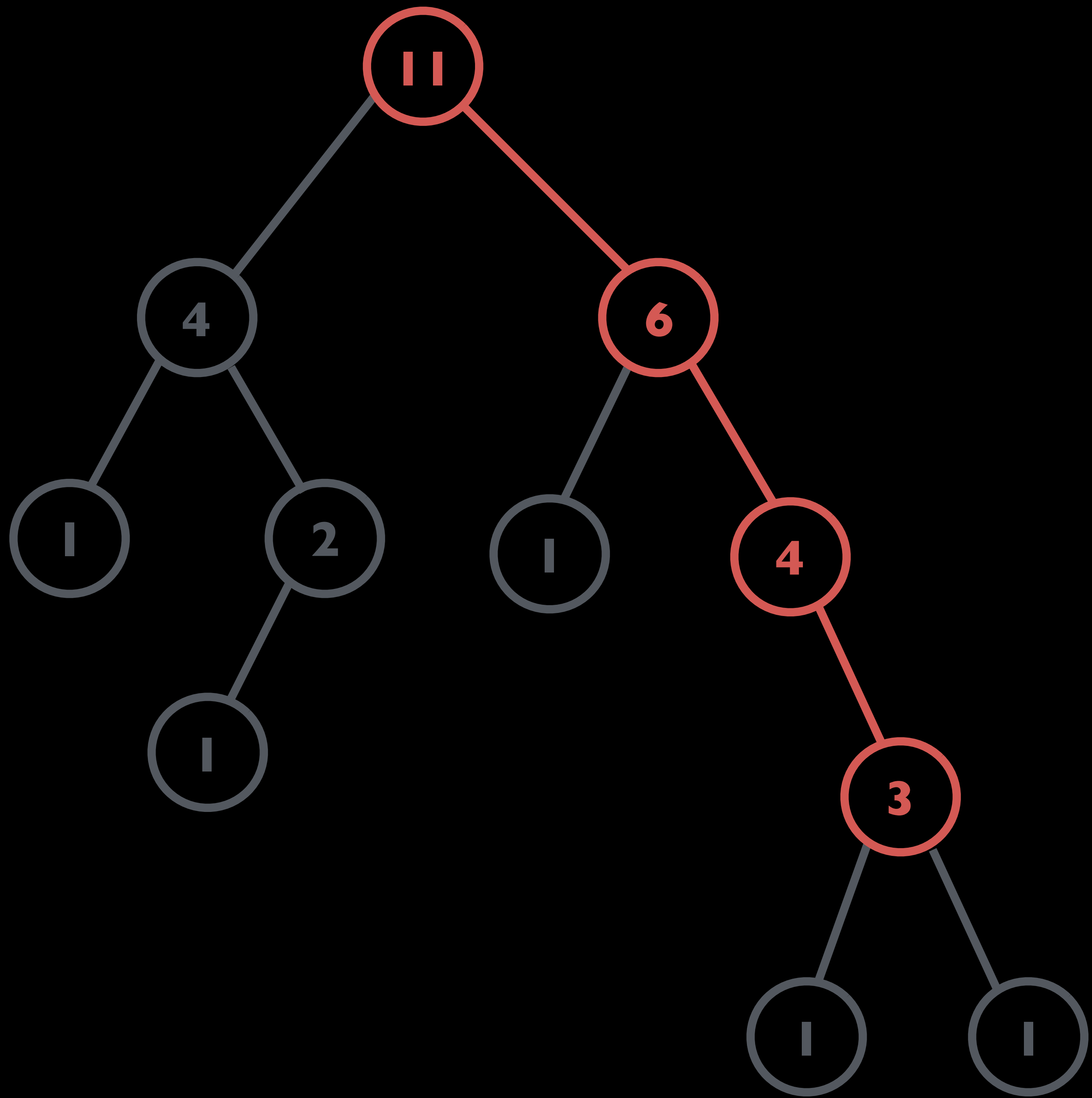
Construction



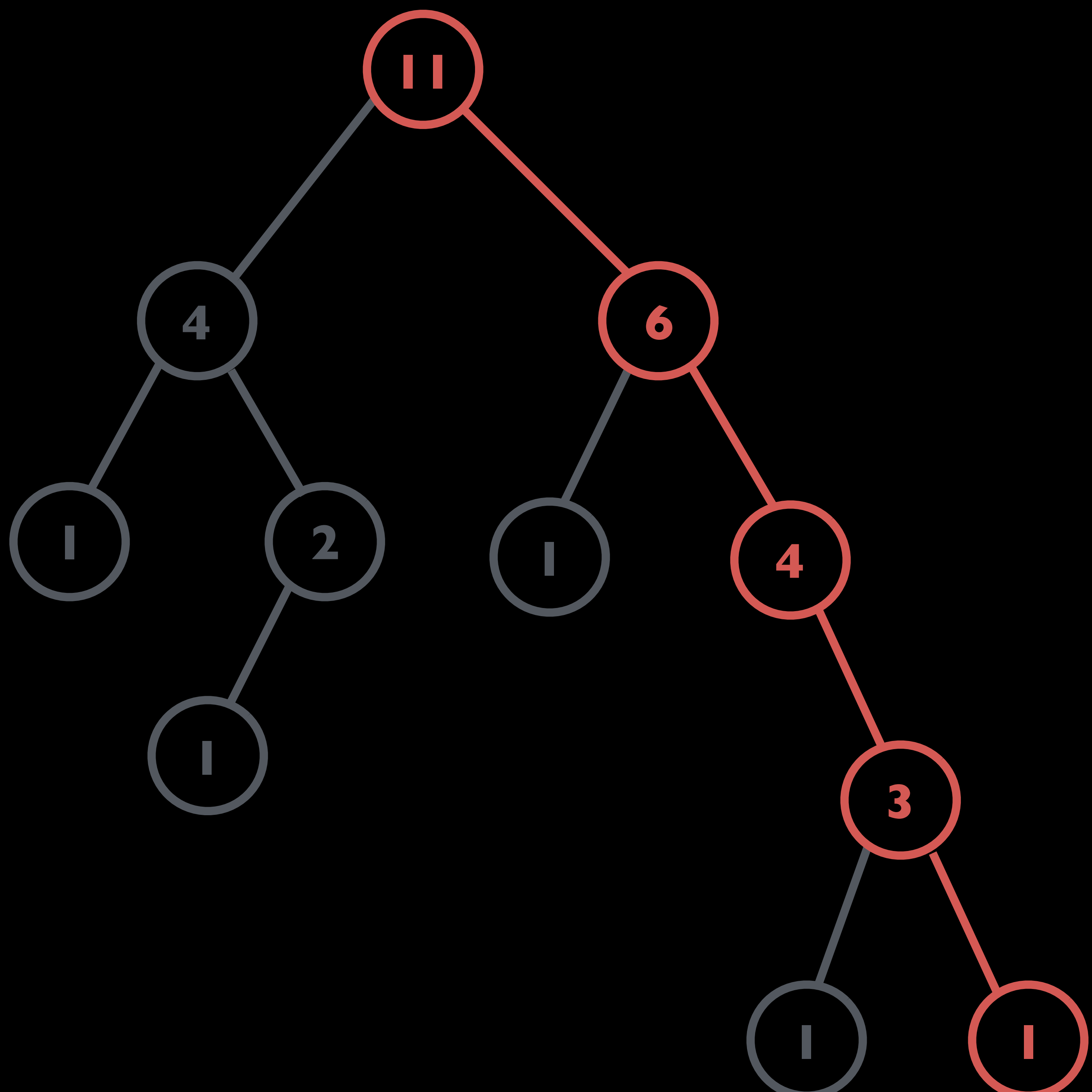
Construction



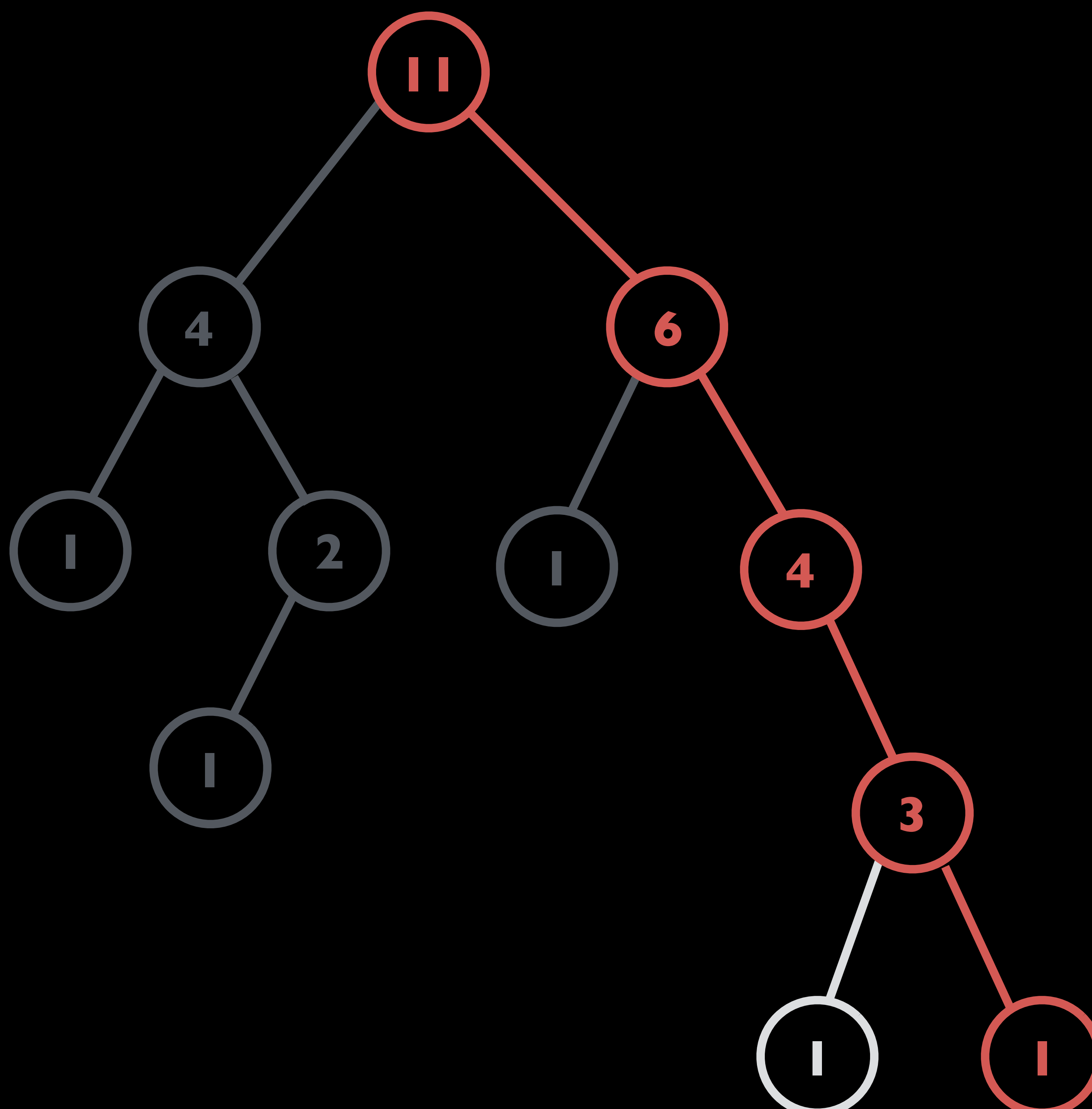
Construction



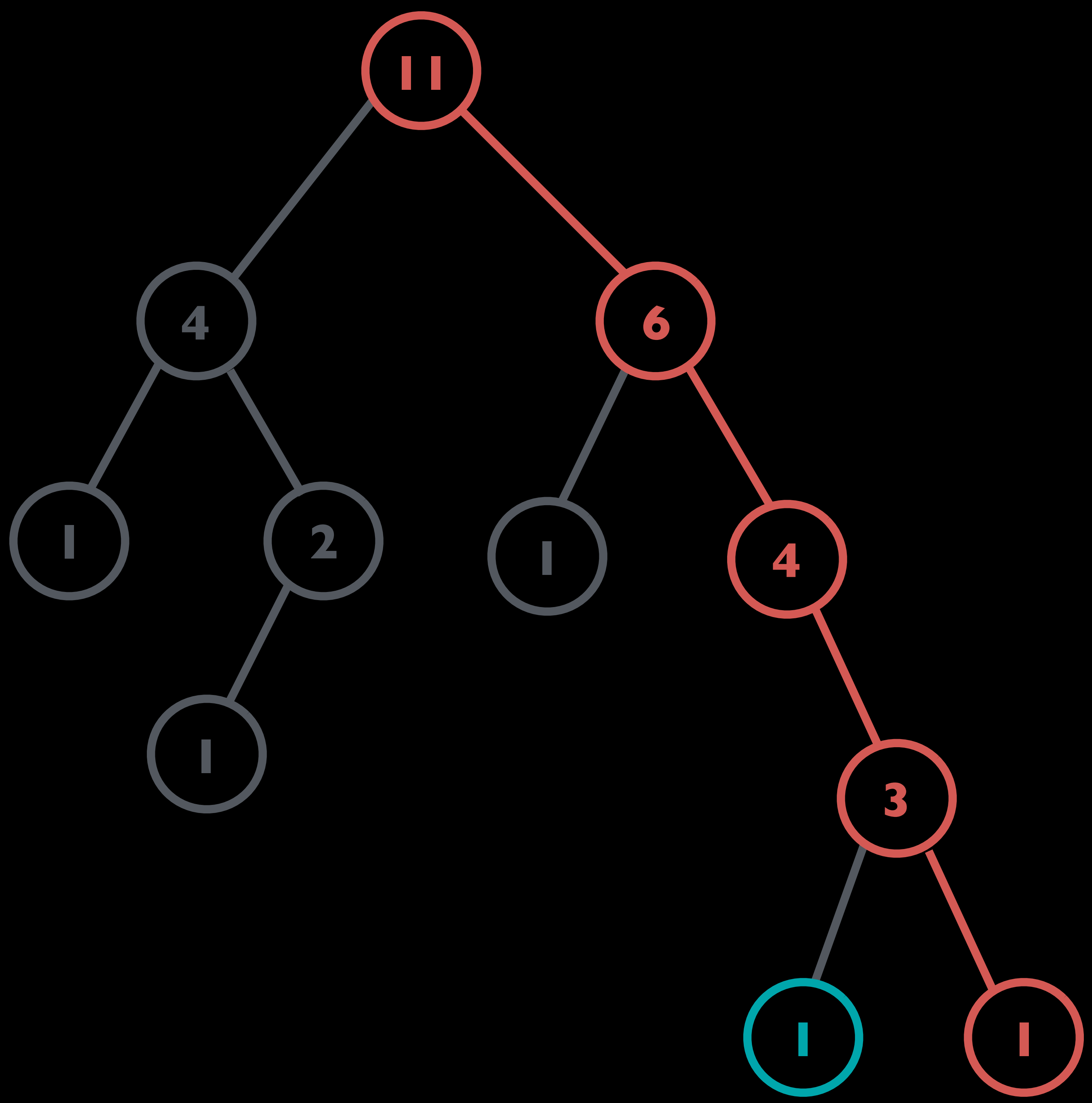
Construction



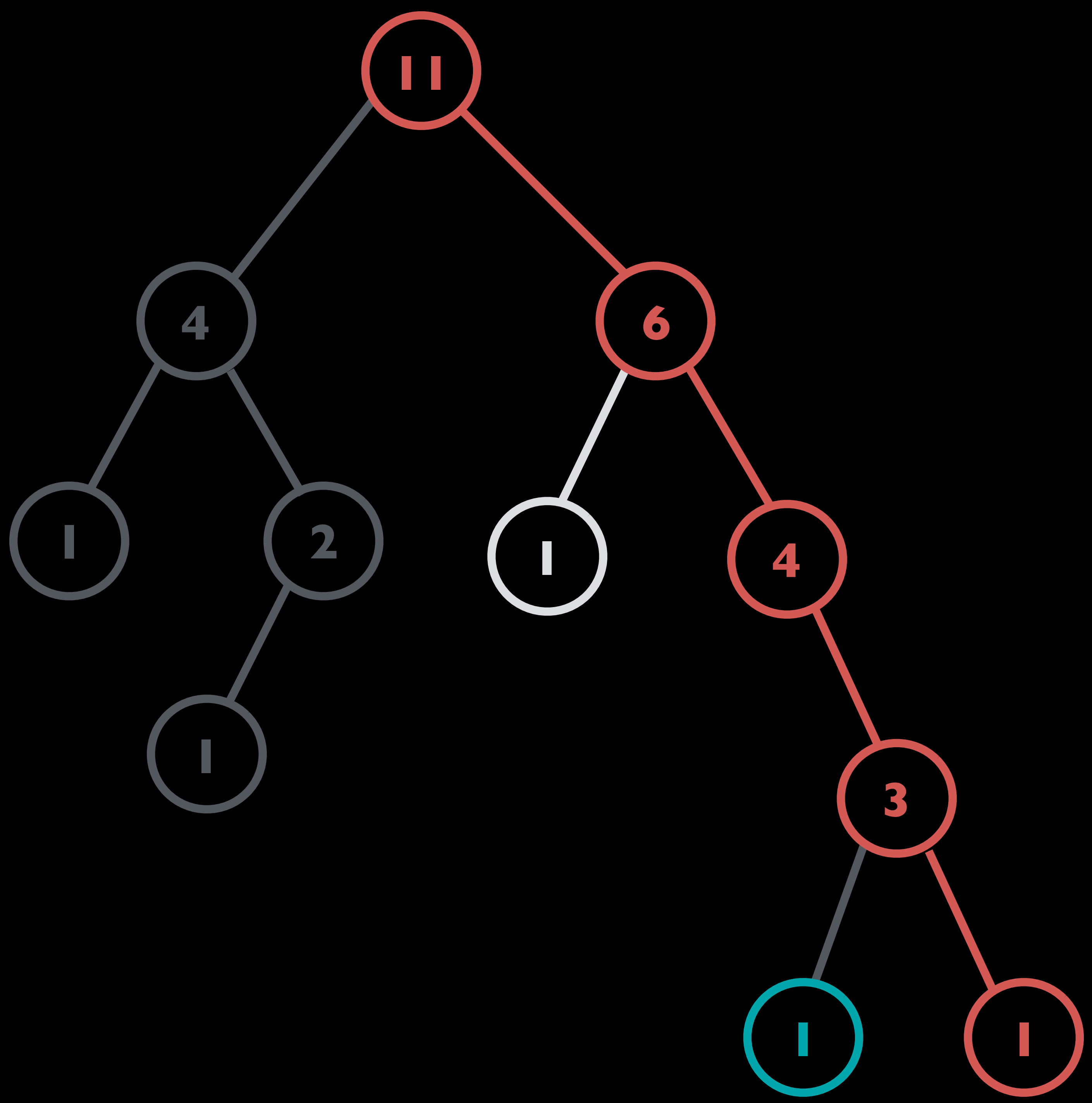
Construction



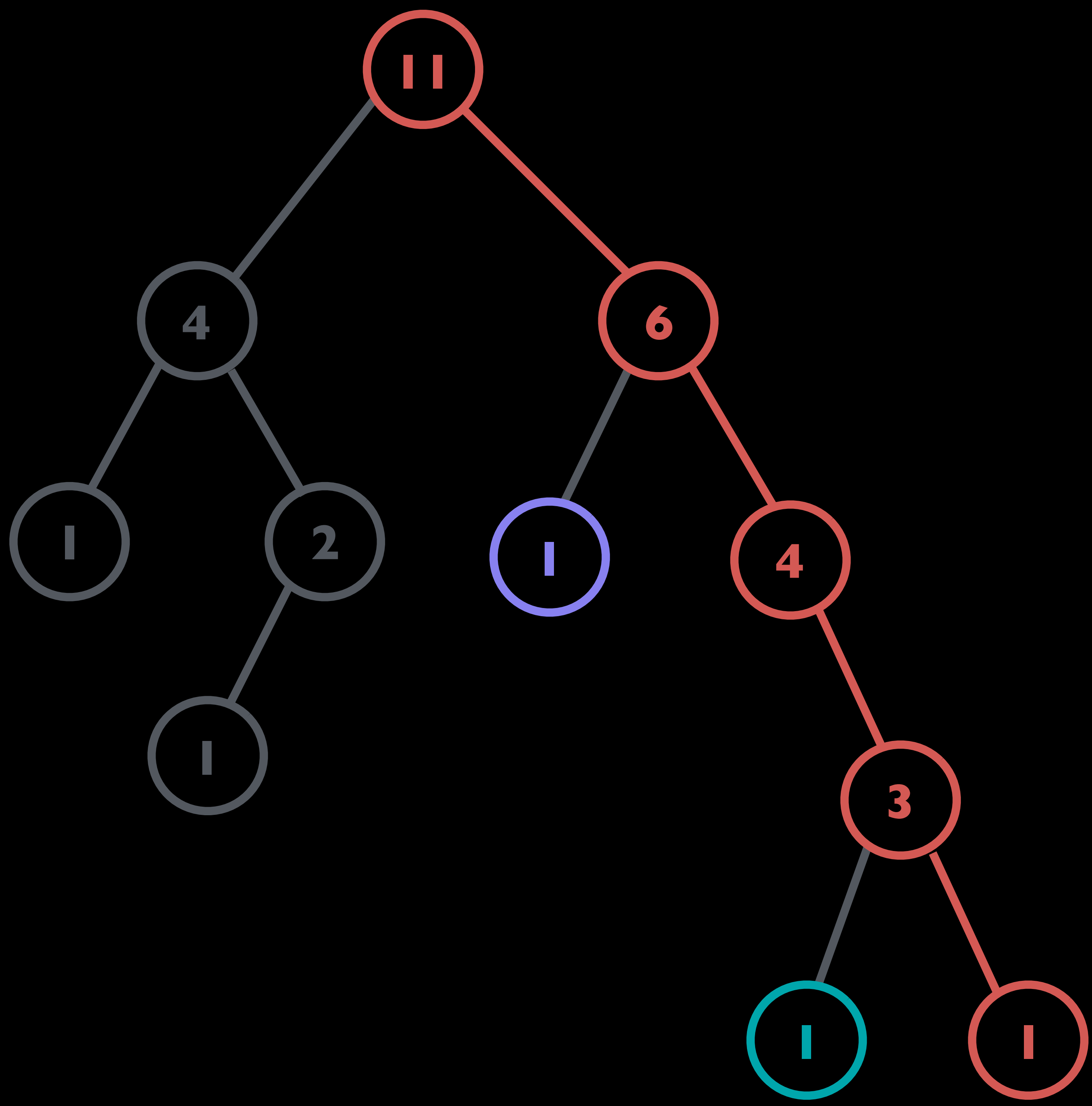
Construction



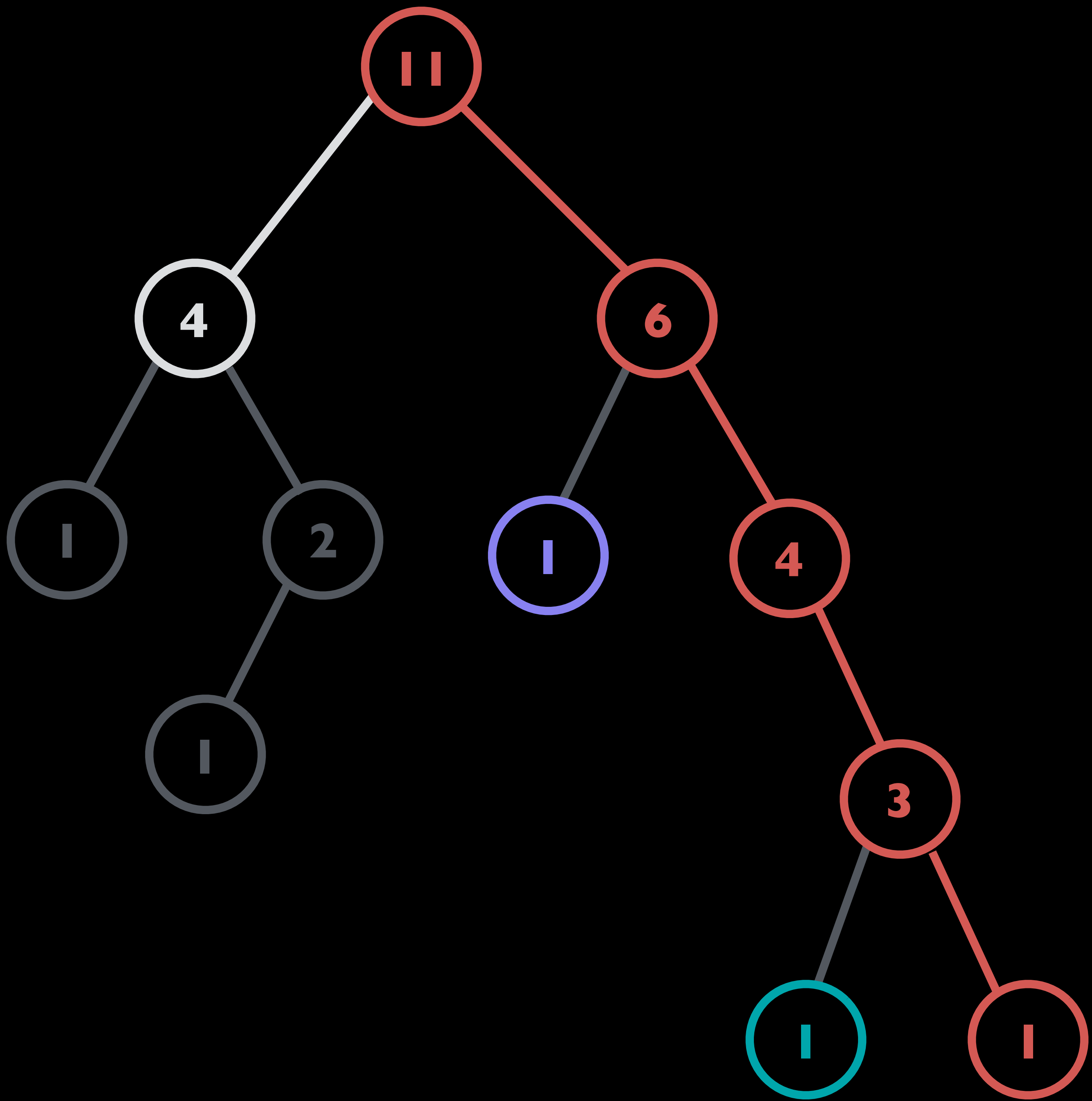
Construction



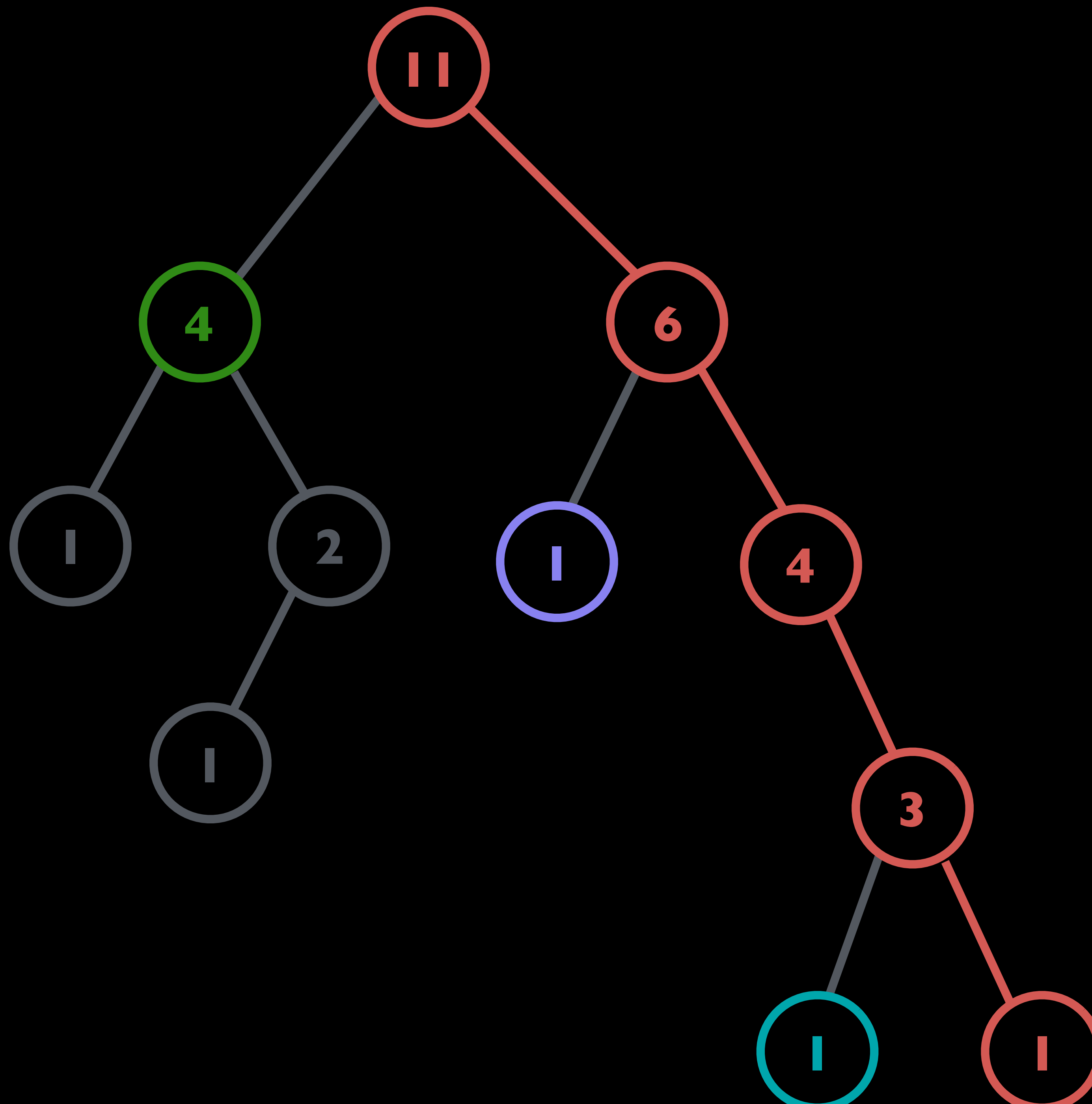
Construction



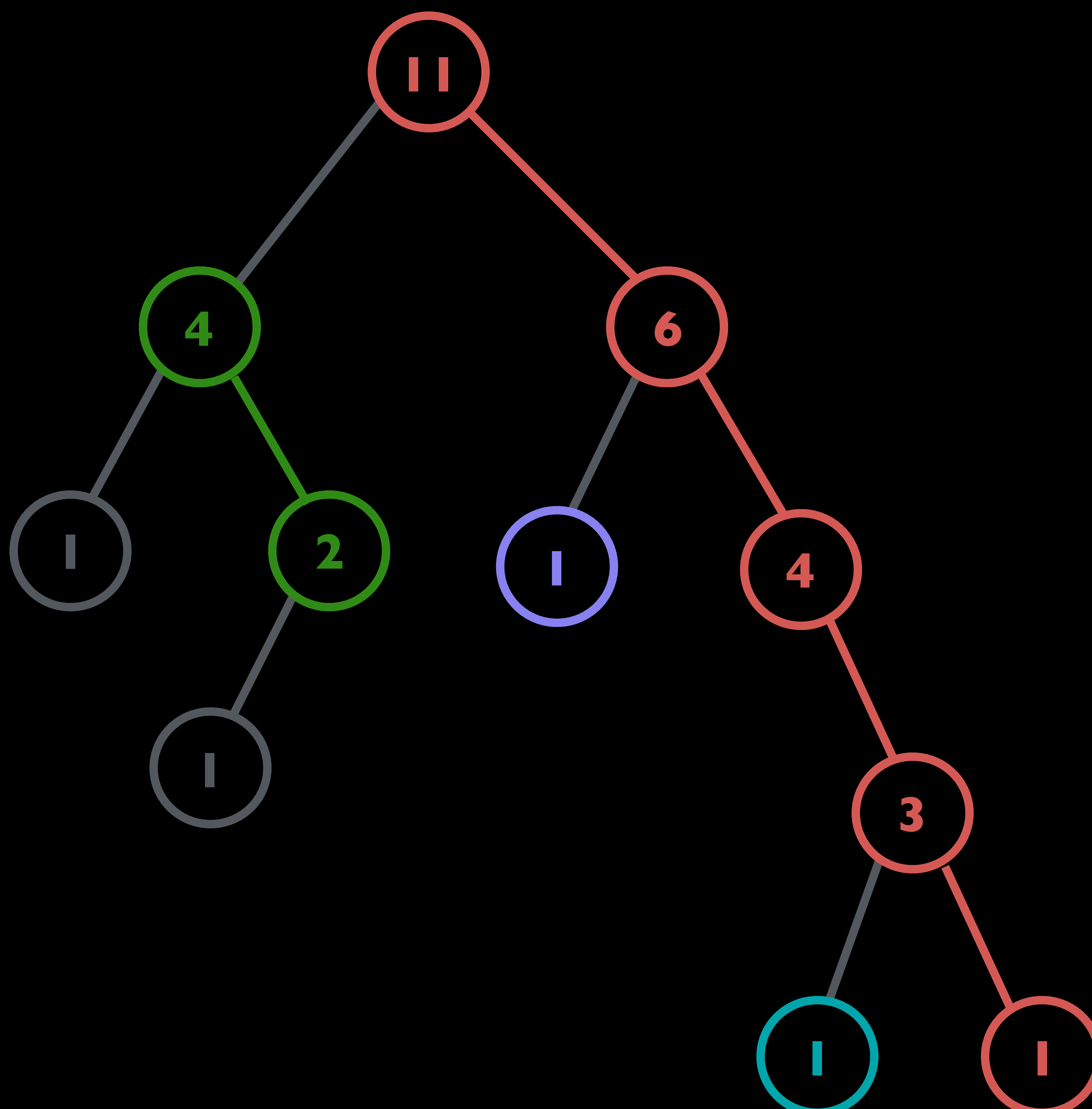
Construction



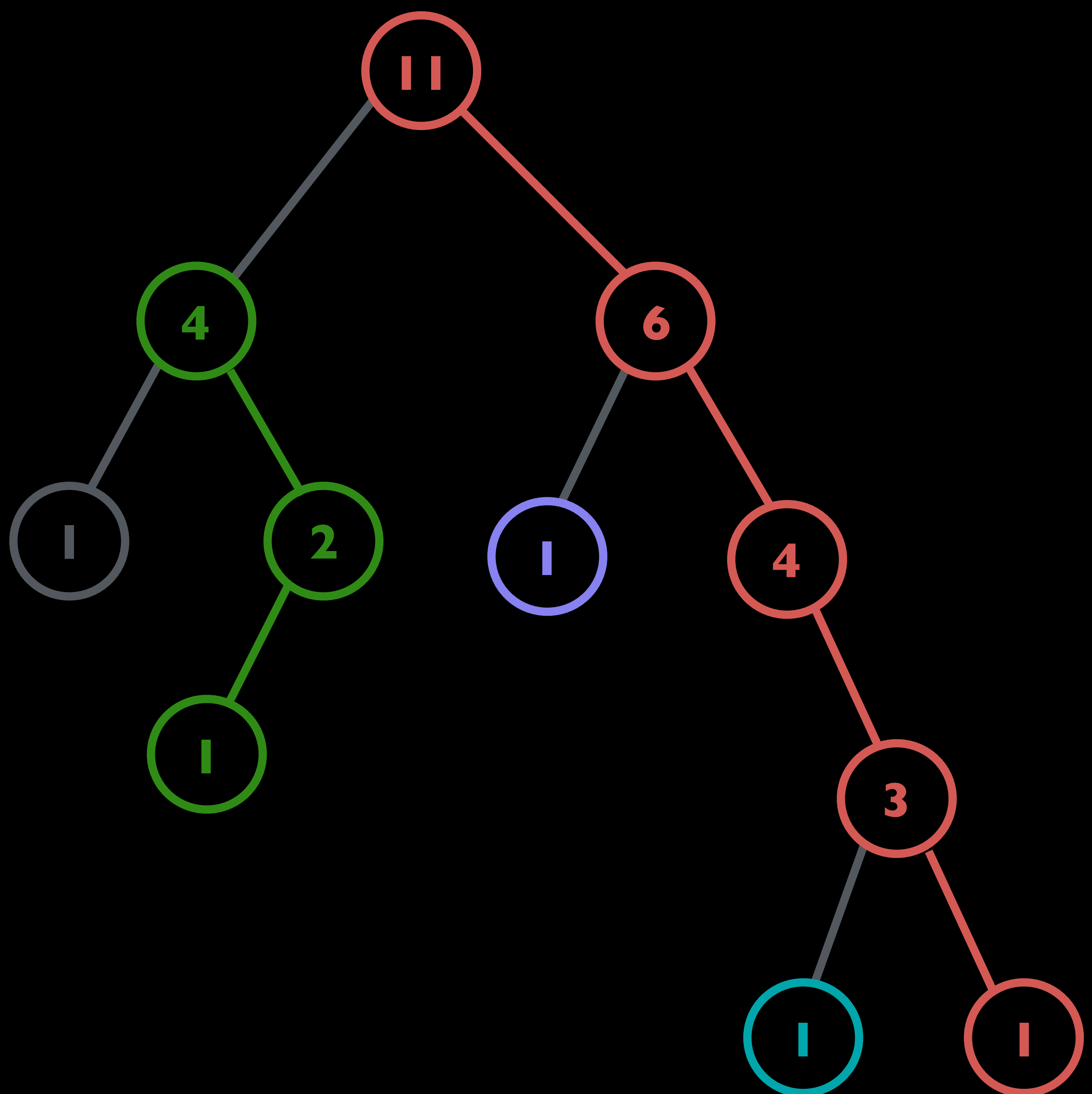
Construction



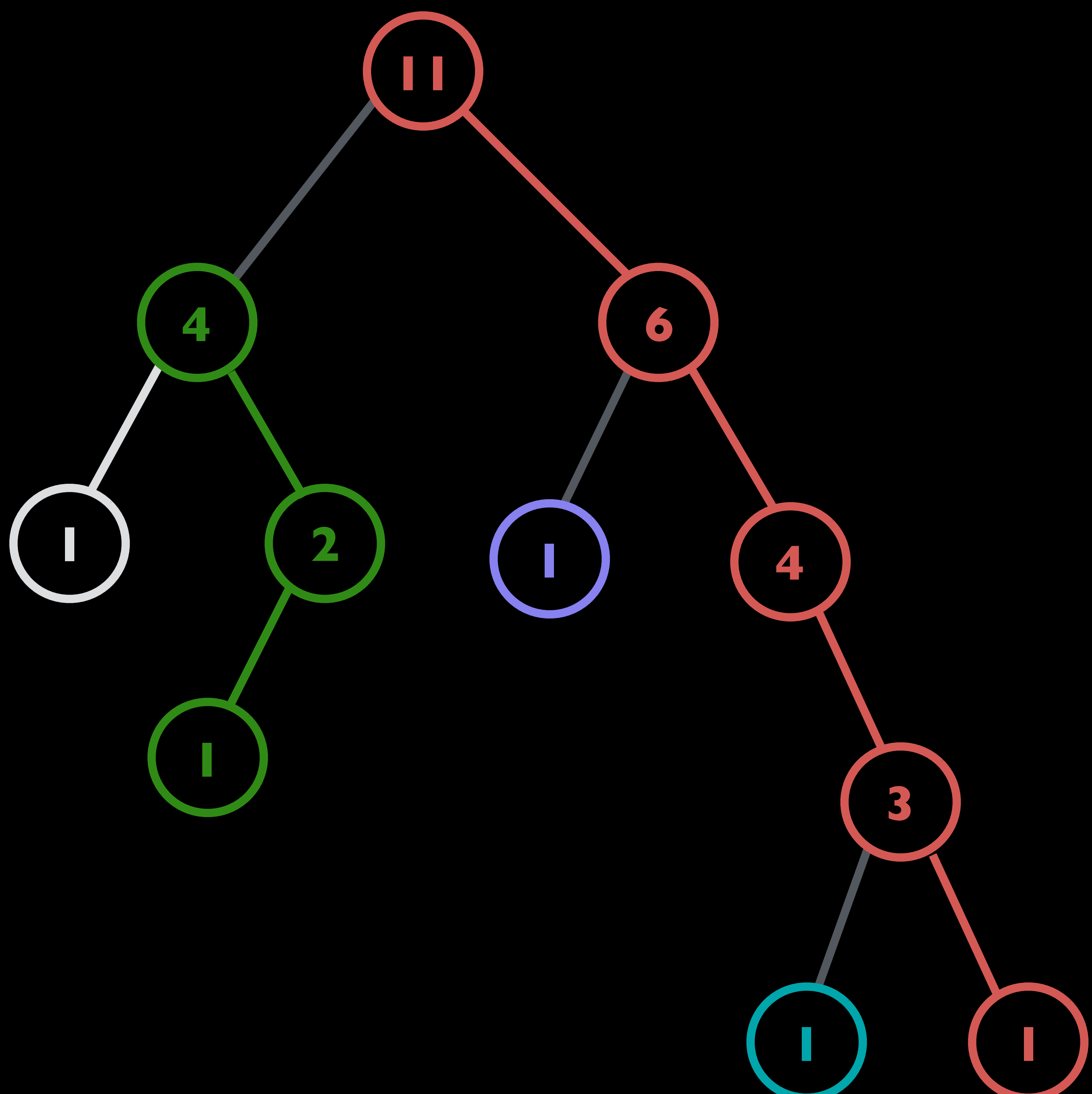
Construction



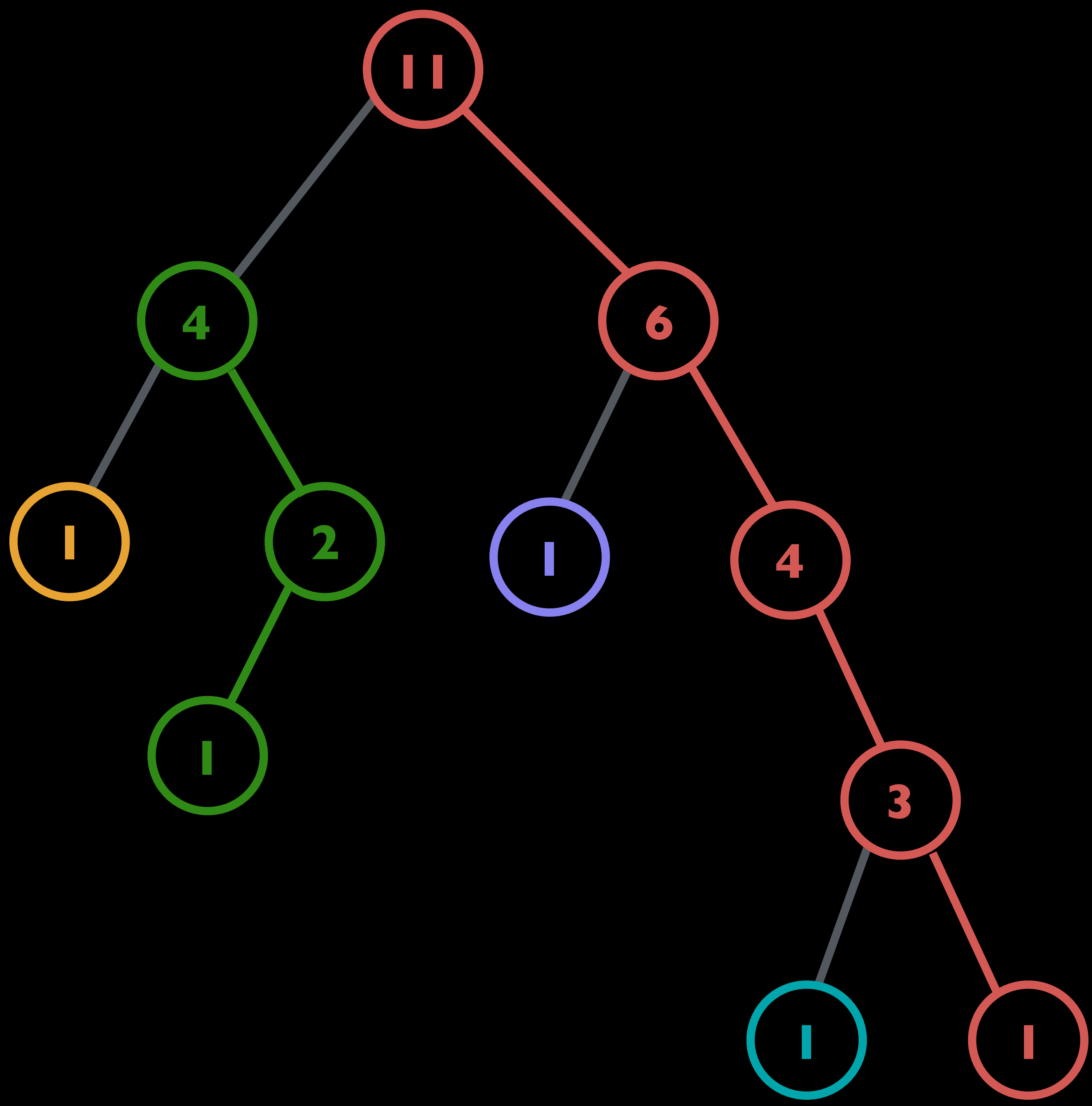
Construction



Construction

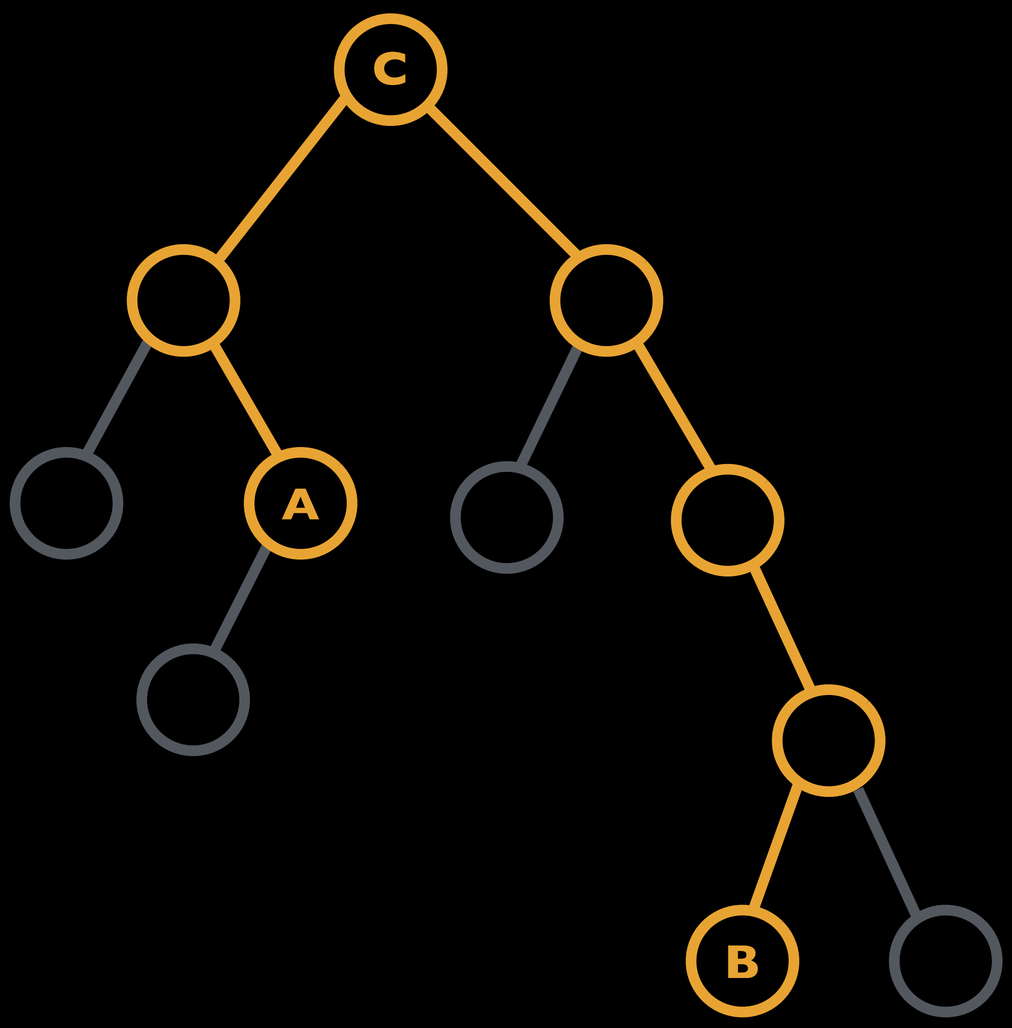


Construction



LCA Query

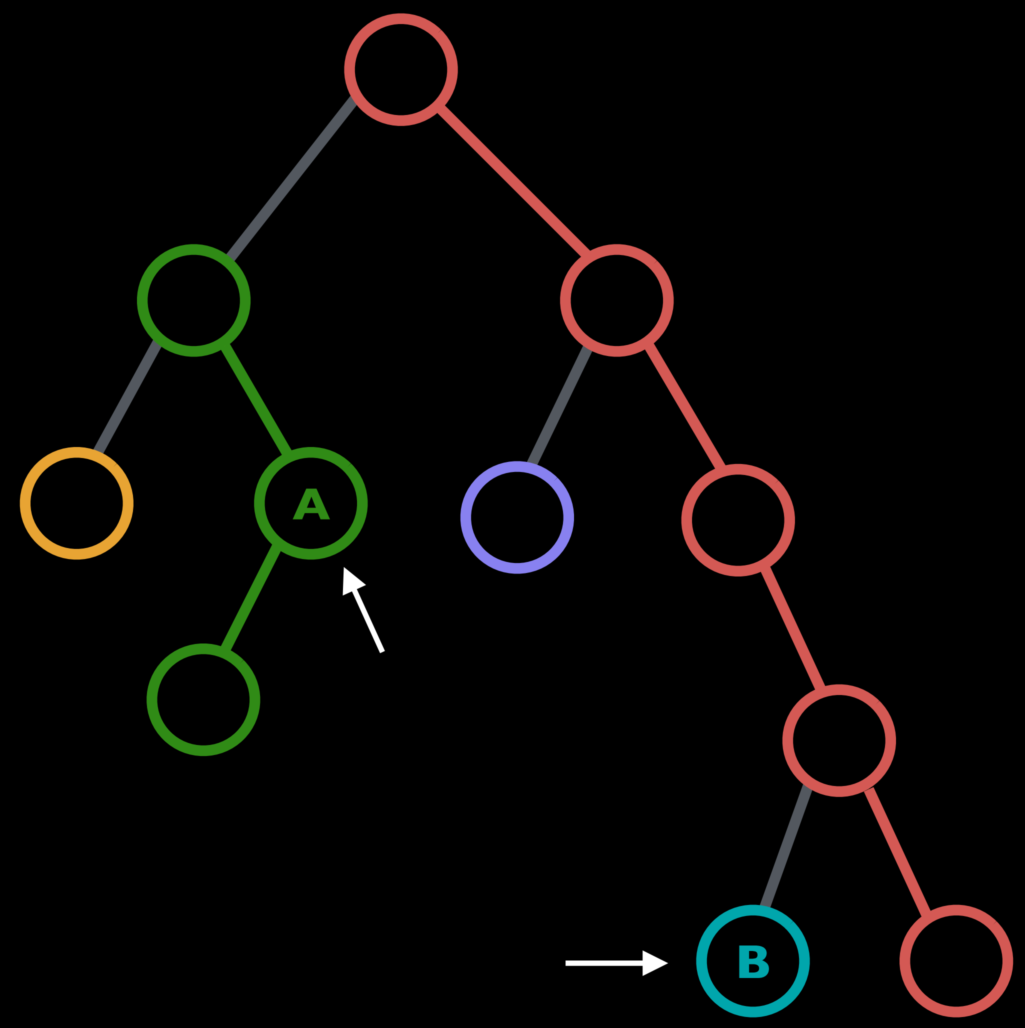
- We can use this to find the *Least Common Ancestor (LCA)* in $O(\log^2(n))$.



LCA(A, B) = C

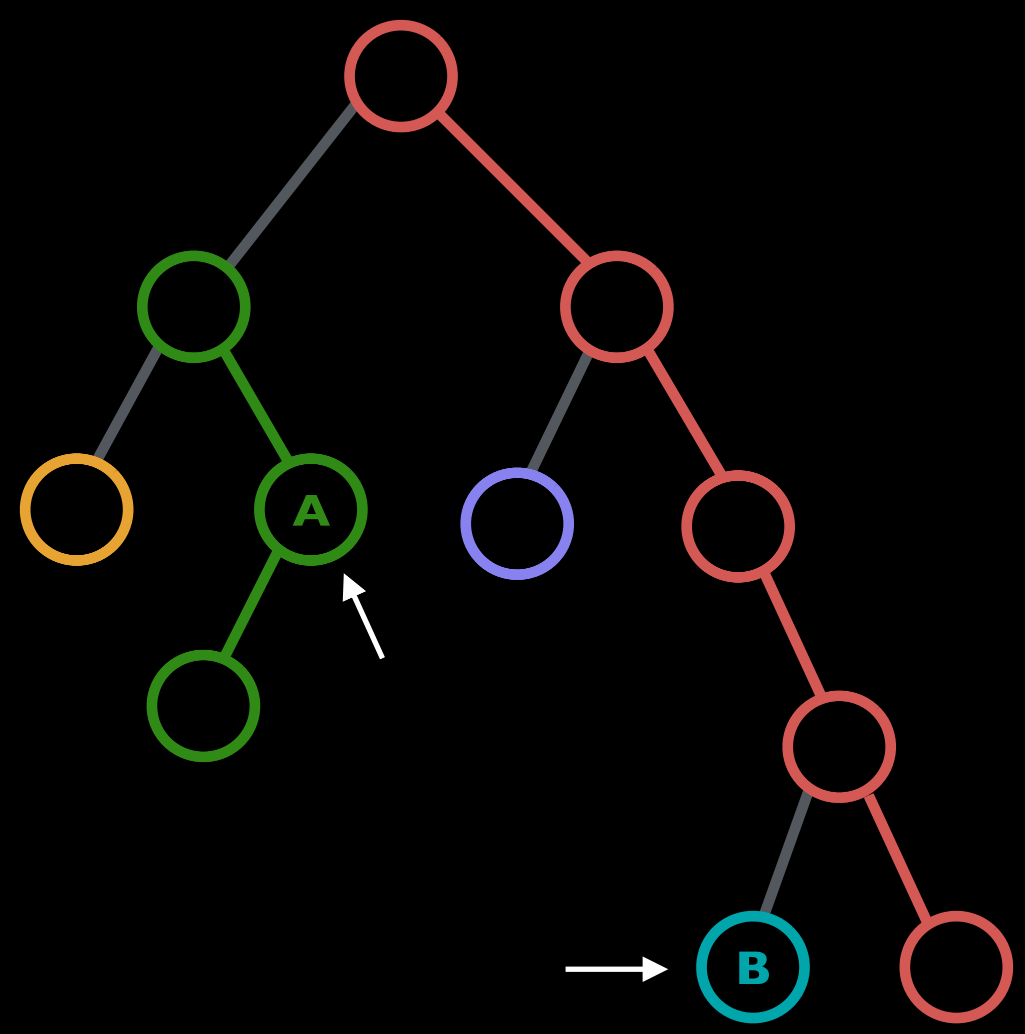
LCA Query

- We start with pointers at A and B.



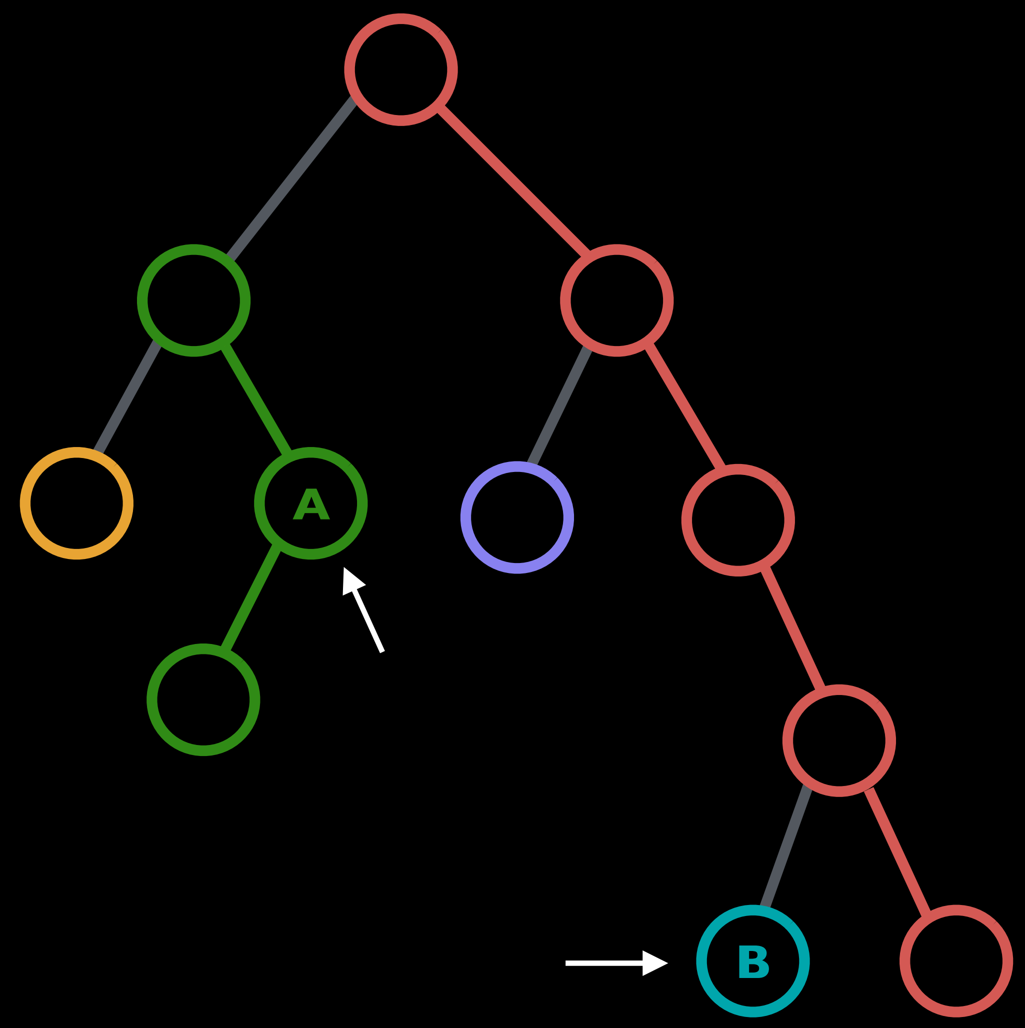
LCA Query

- We will be walking up the tree using the chains.



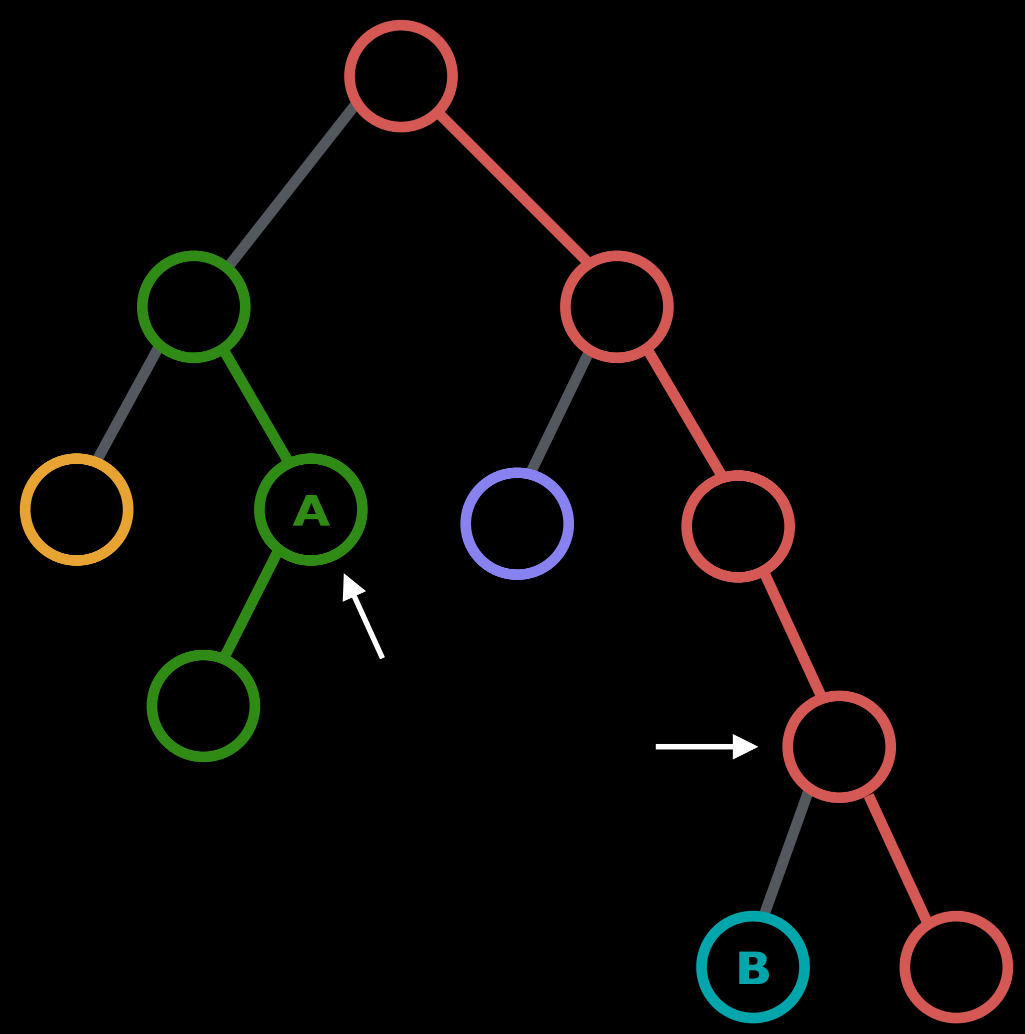
LCA Query

- For each step in the walk of one pointer, the other pointer will do an entire walk.



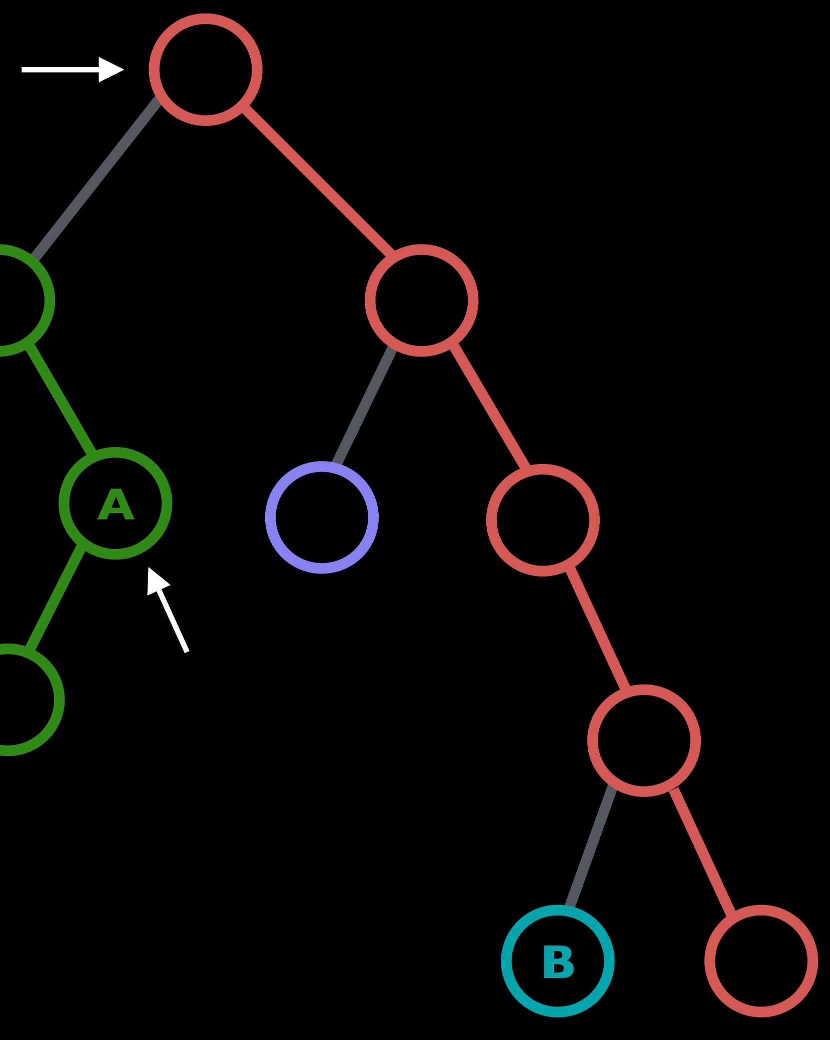
LCA Query

- When walking up we either move to the parent or to the head of the chain.



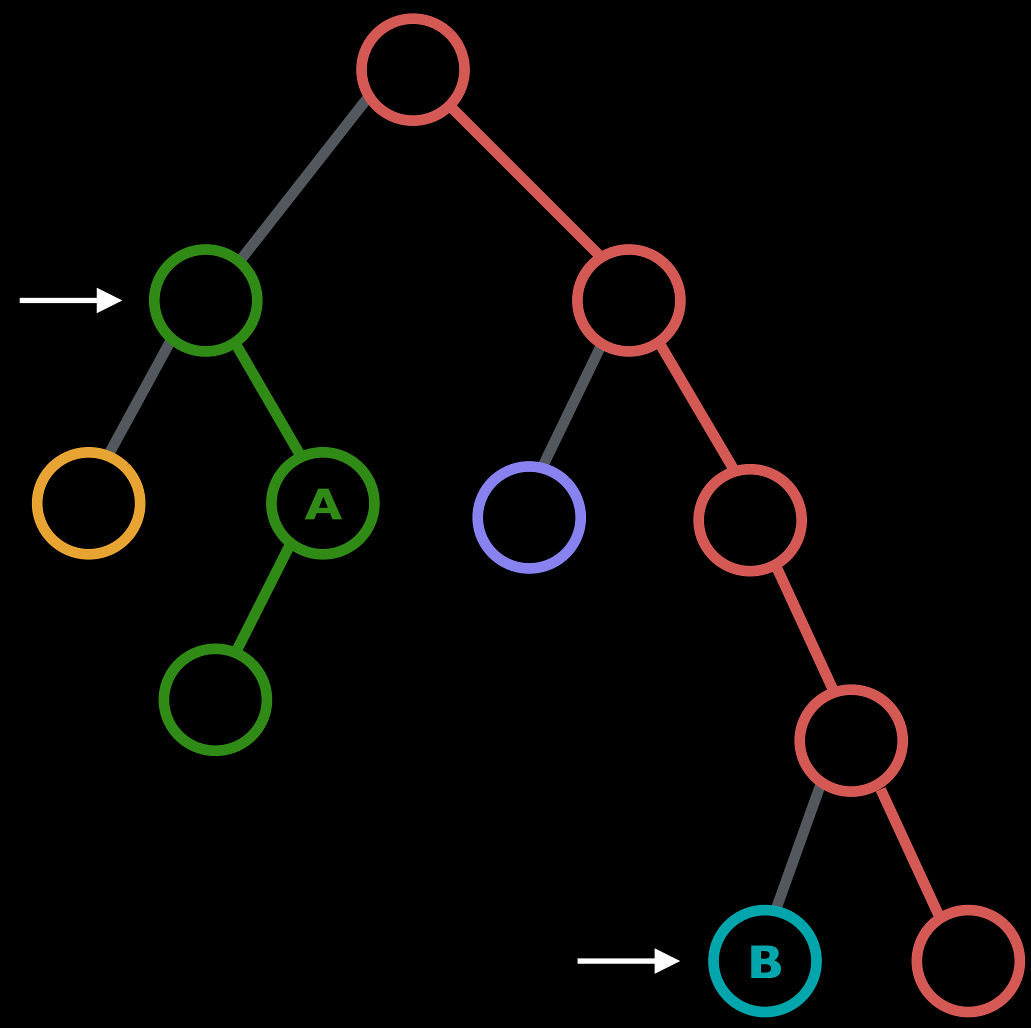
LCA Query

- When walking up we either move to the parent or to the head of the chain.



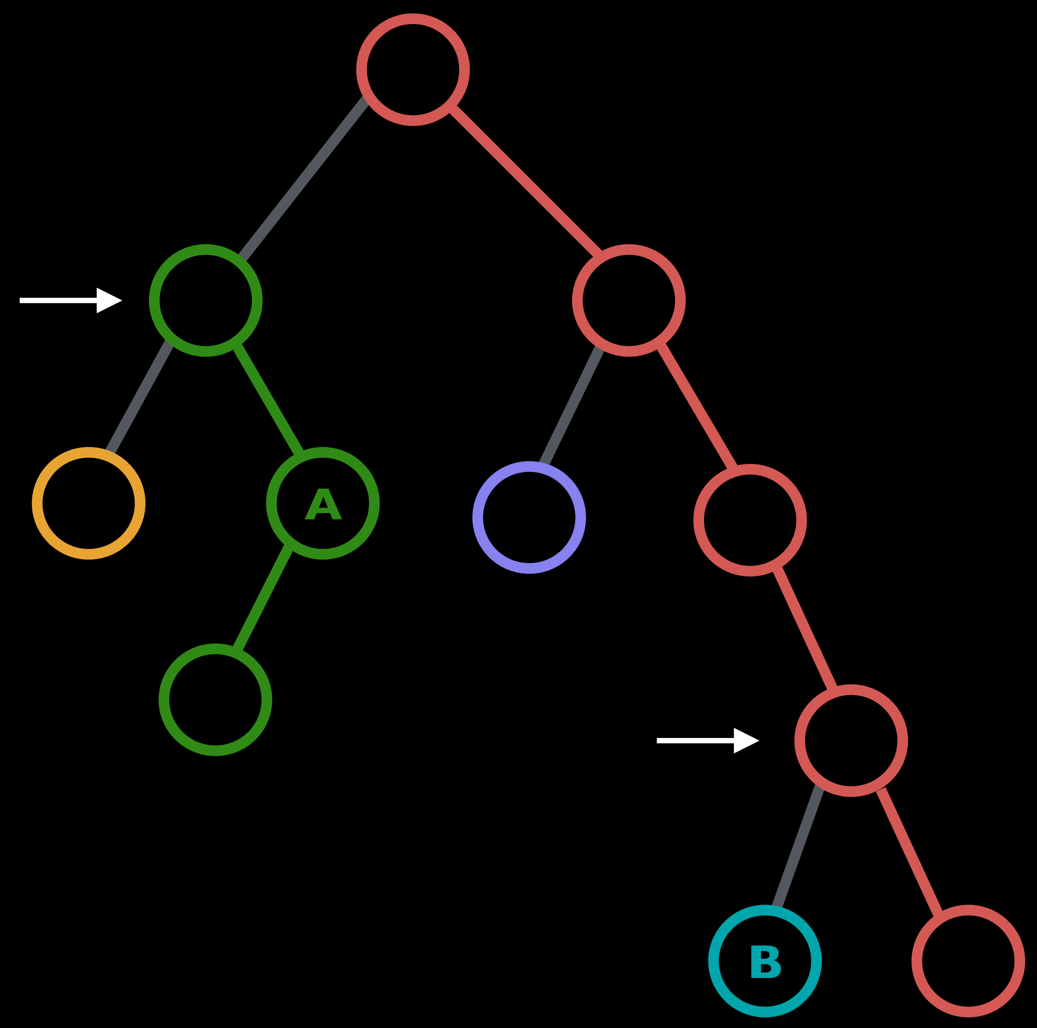
LCA Query

- When the top is reached, the first pointer is advanced and the second one is reset.



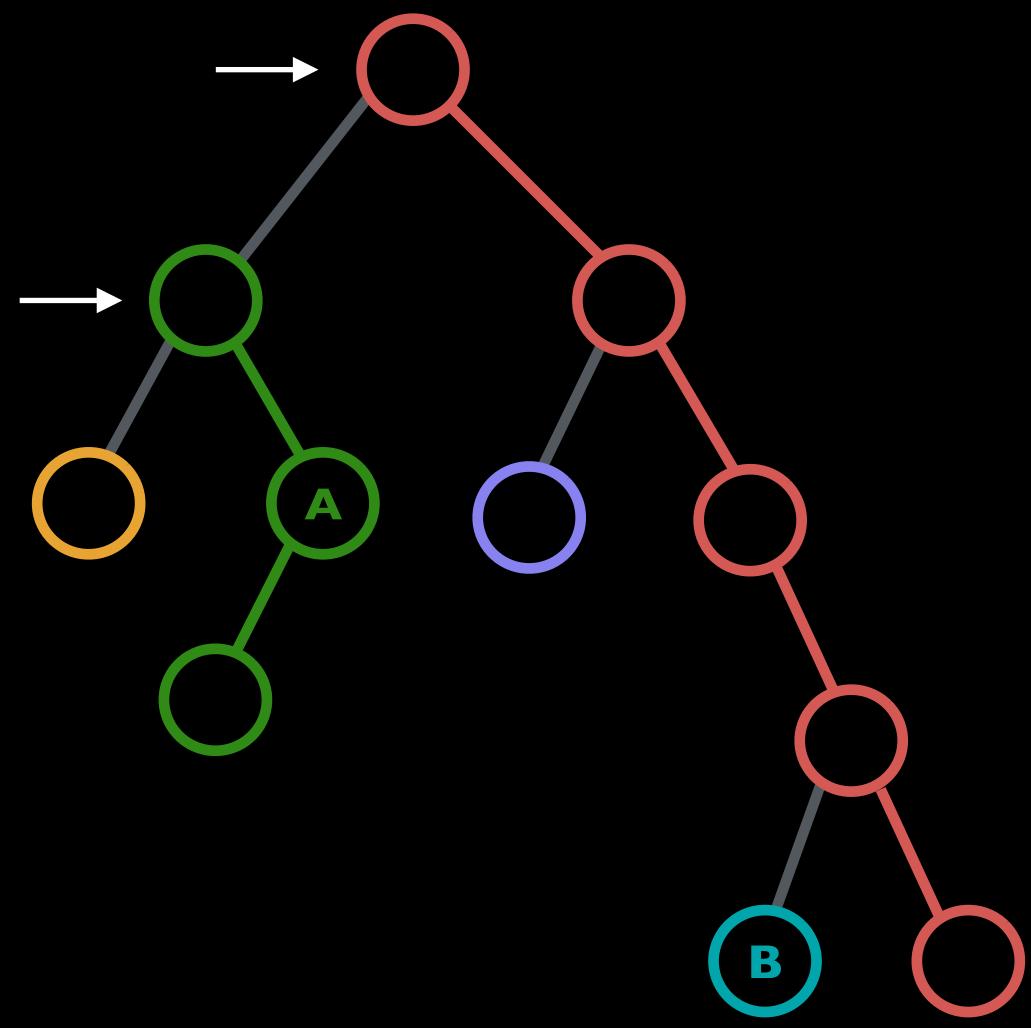
LCA Query

- And the process is continued.



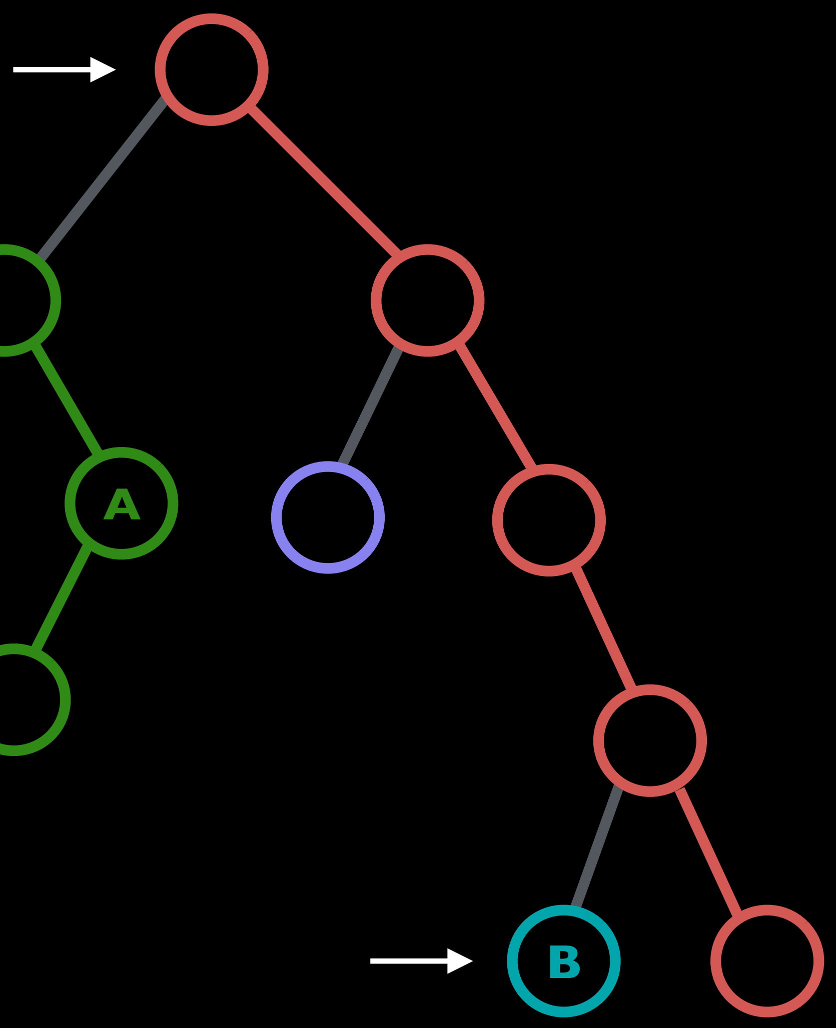
LCA Query

- And the process is continued.



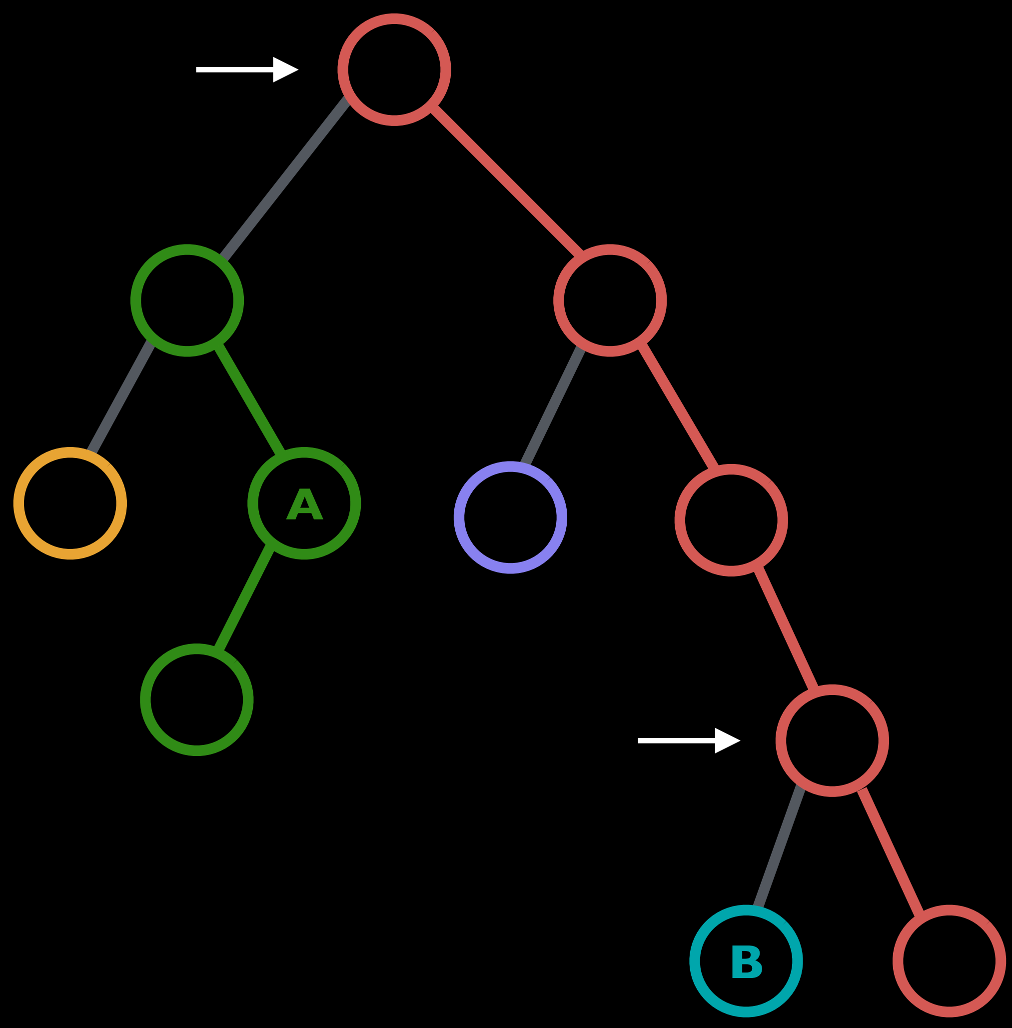
LCA Query

- And the process is continued.



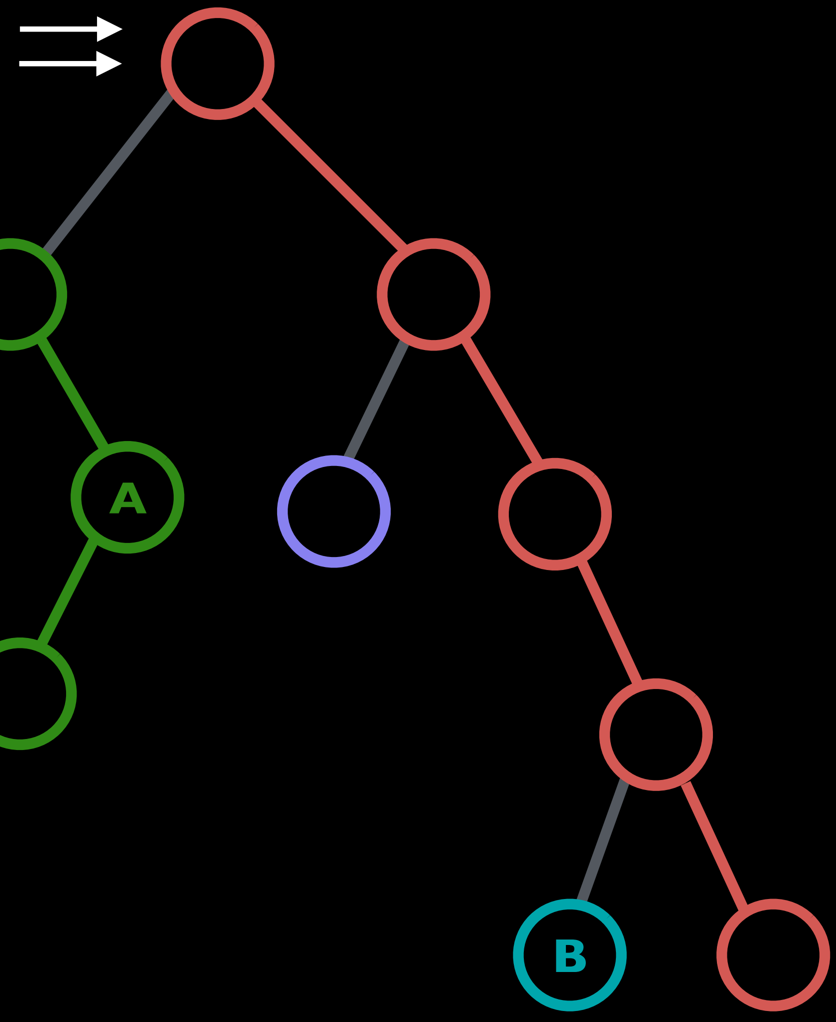
LCA Query

- And the process is continued.



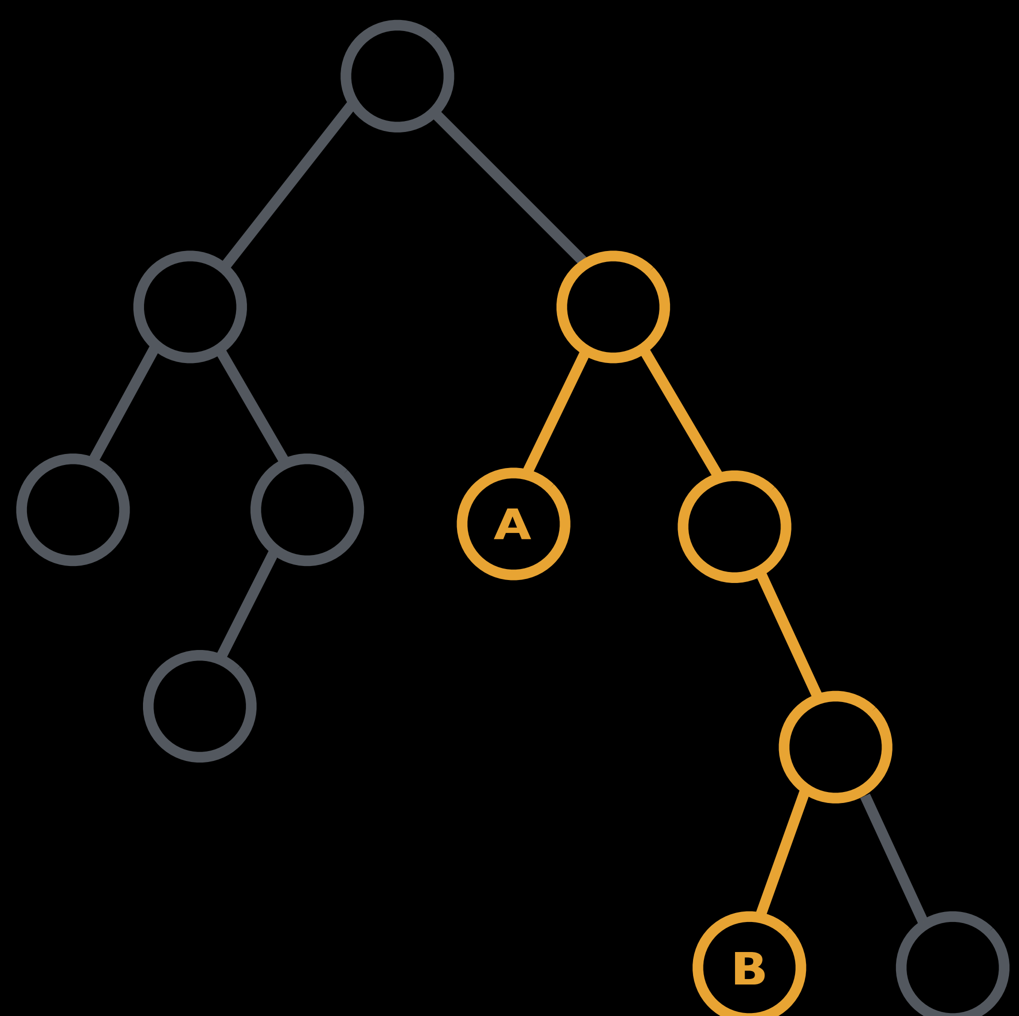
LCA Query

- We have found the LCA when both pointers are on the <u>same chain</u>.



Distance Query

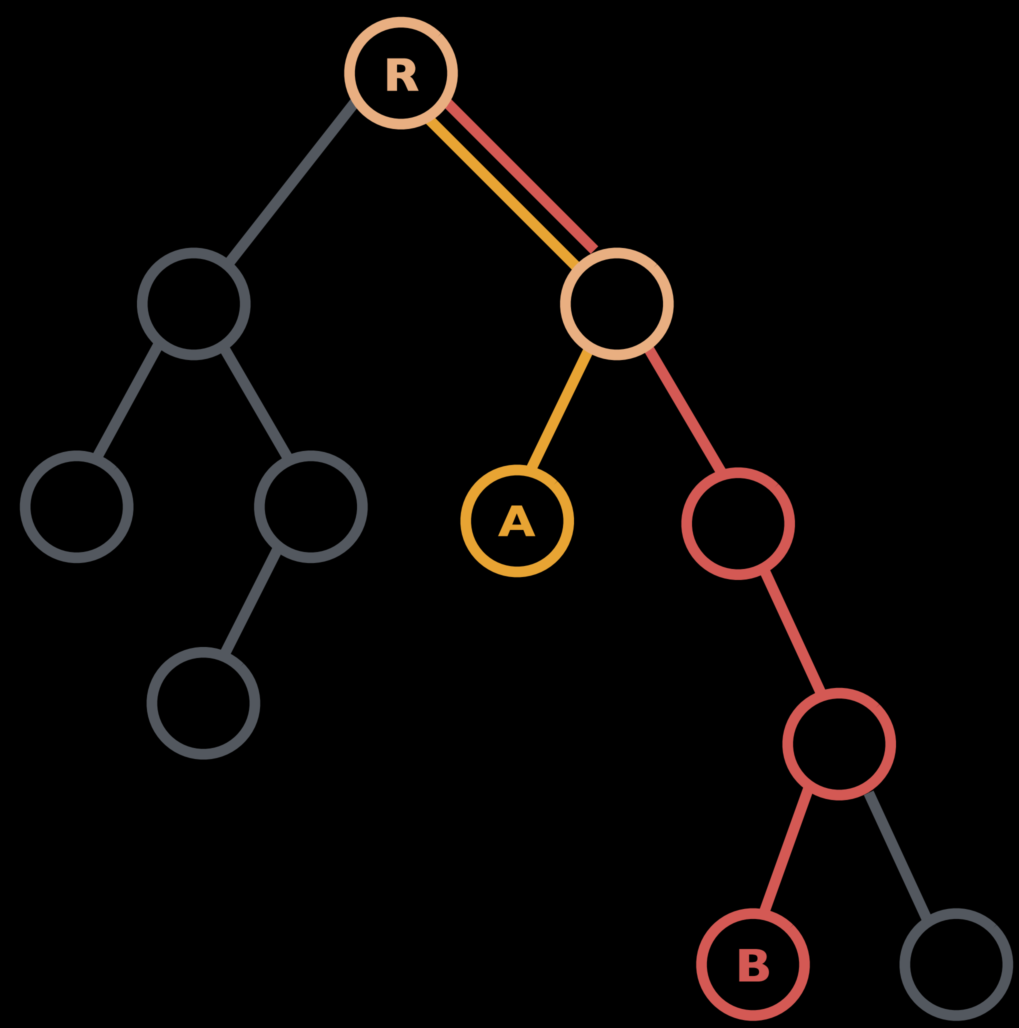
- We can use easily find the distance between two nodes if we know their LCA.



Dist(A, B) = 4

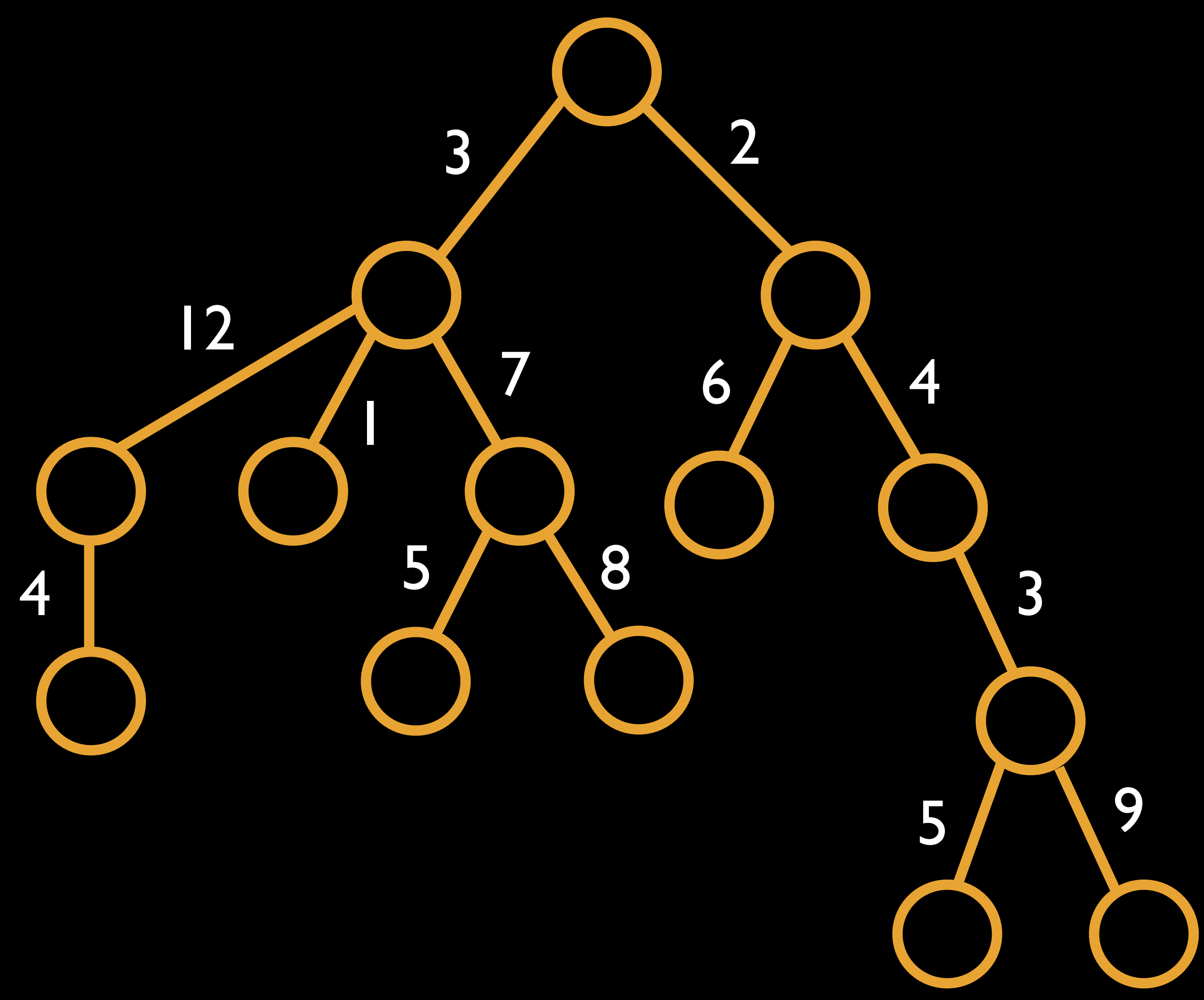
Distance Query

$$Dist(A,B) = Dist(R,A) + Dist(R,B) - 2 * Depth(LCA(A,B))$$



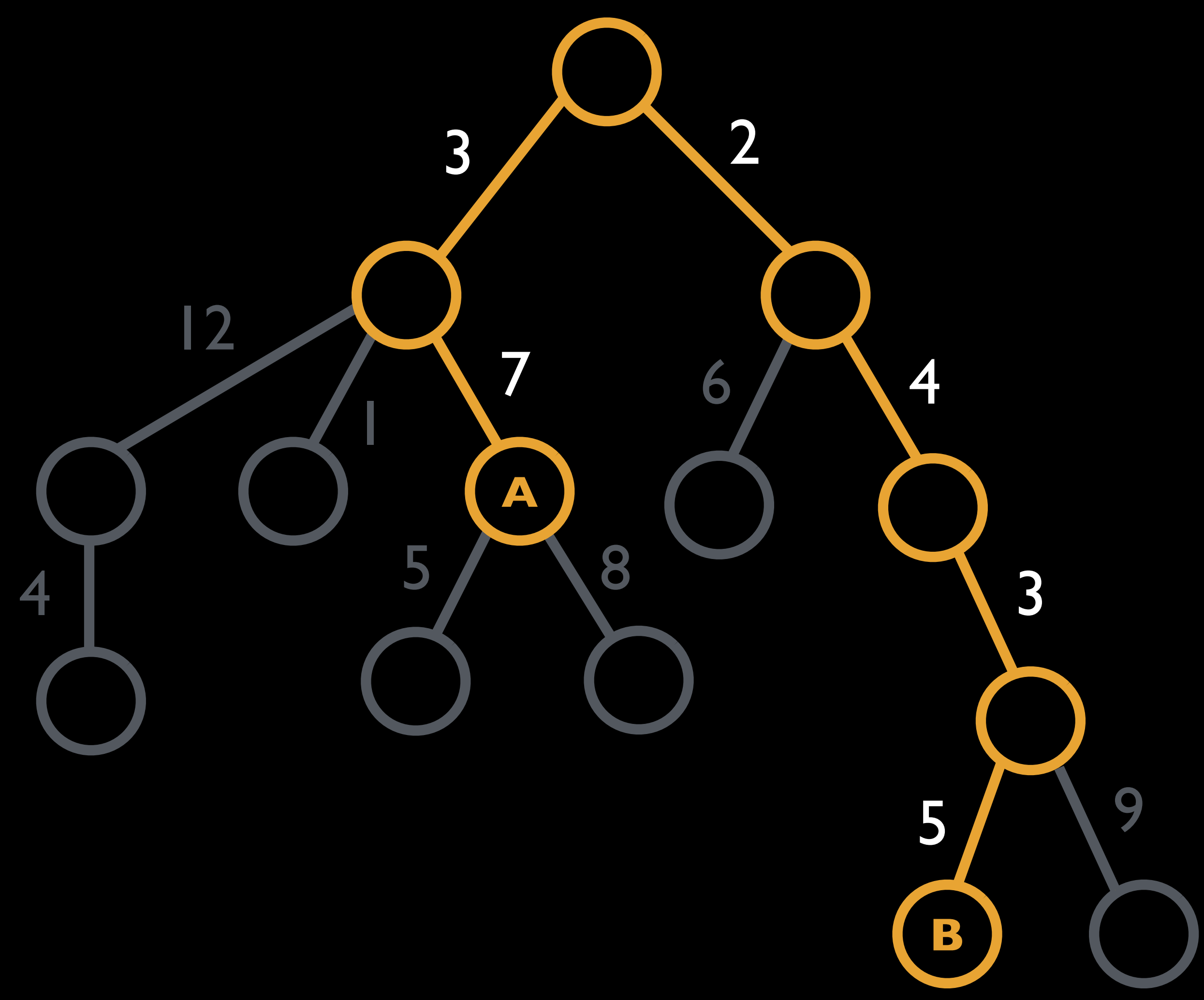
Max Edge Query

- Suppose our tree is weighted. We may want to find the largest edge weight on the path between two nodes.



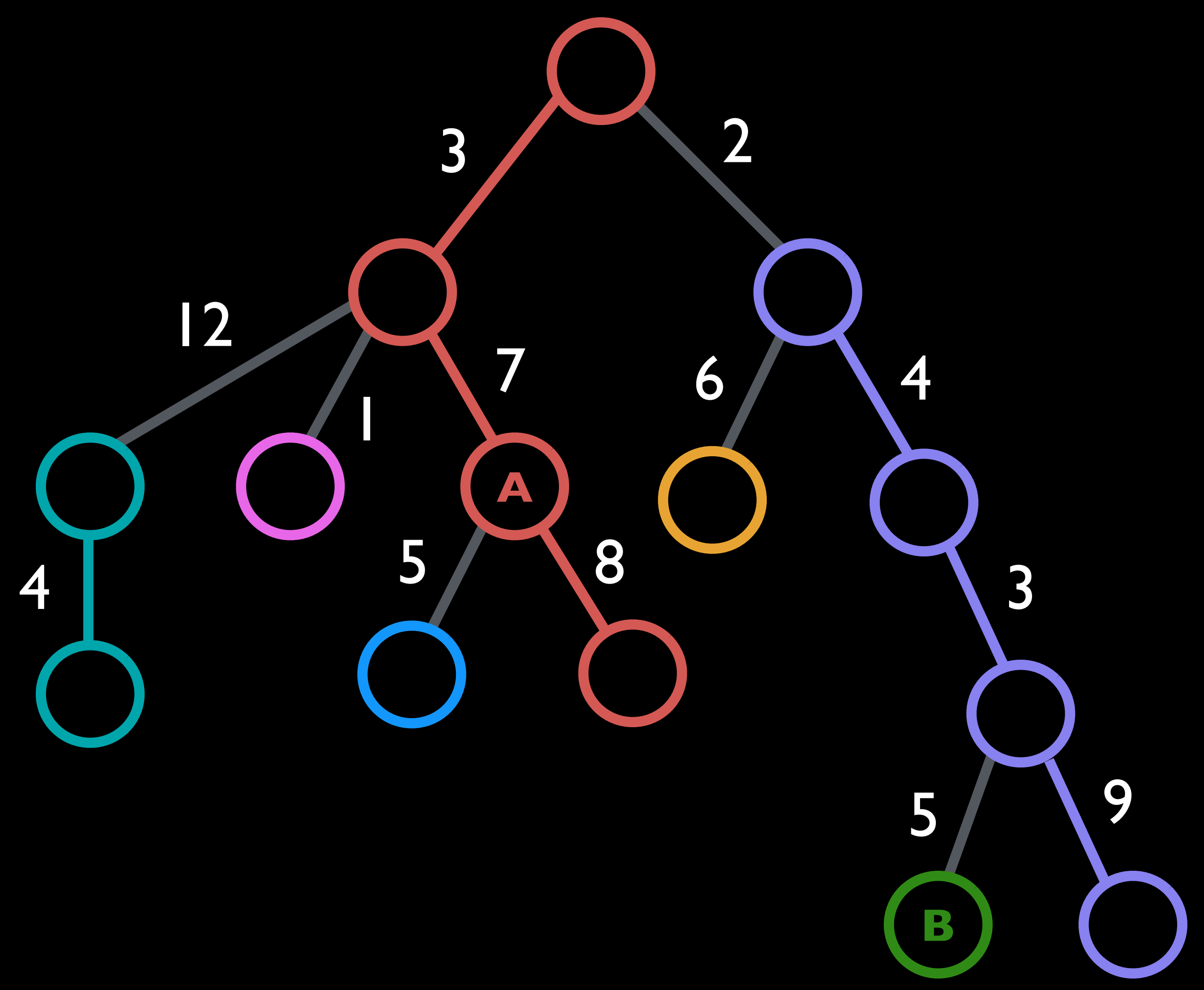
Max Edge Query

- Suppose our tree is weighted. We may want to find the largest edge weight on the path between two nodes.



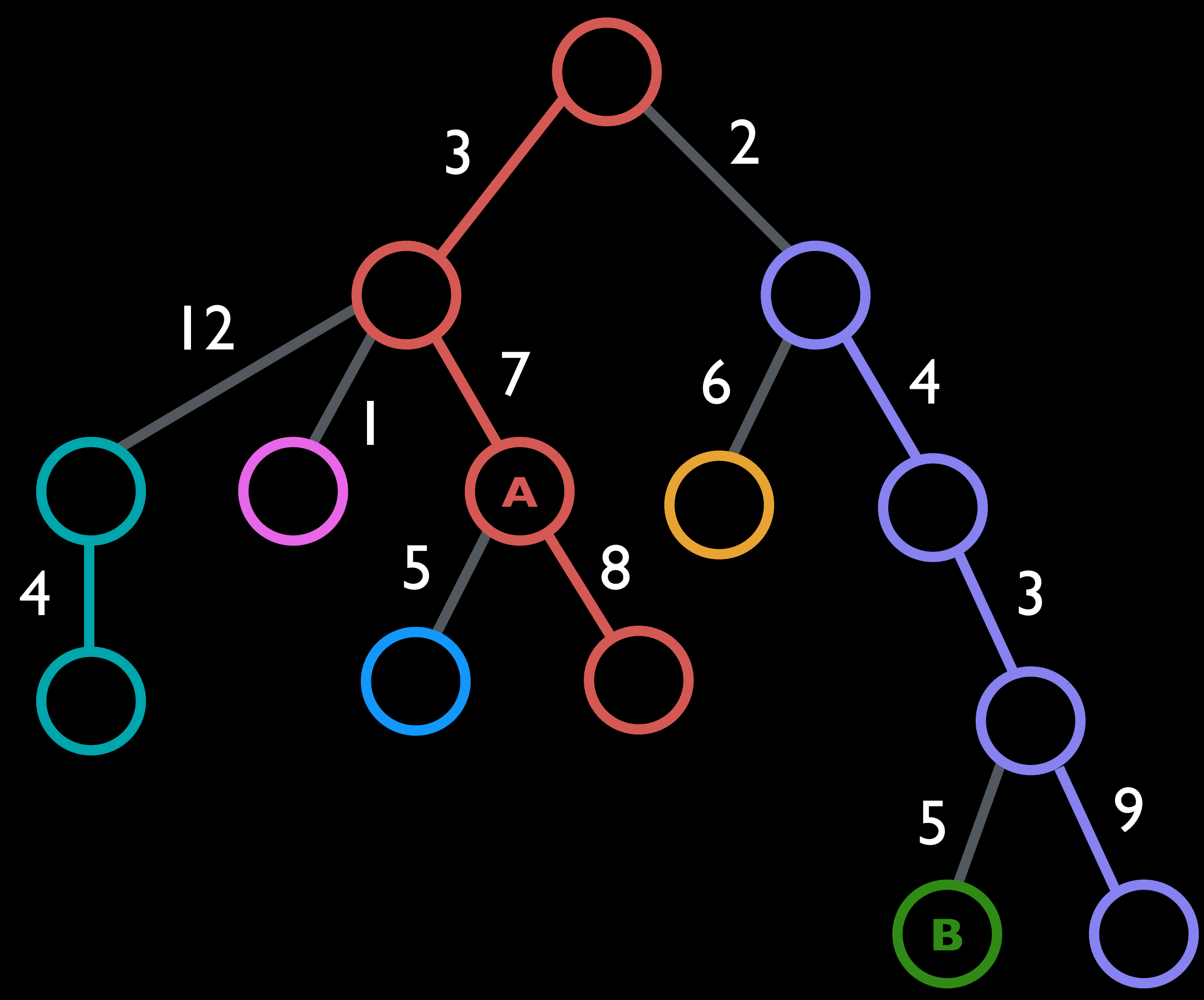
Max Edge Query

- Suppose our tree is weighted. We may want to find the largest edge weight on the path between two nodes.



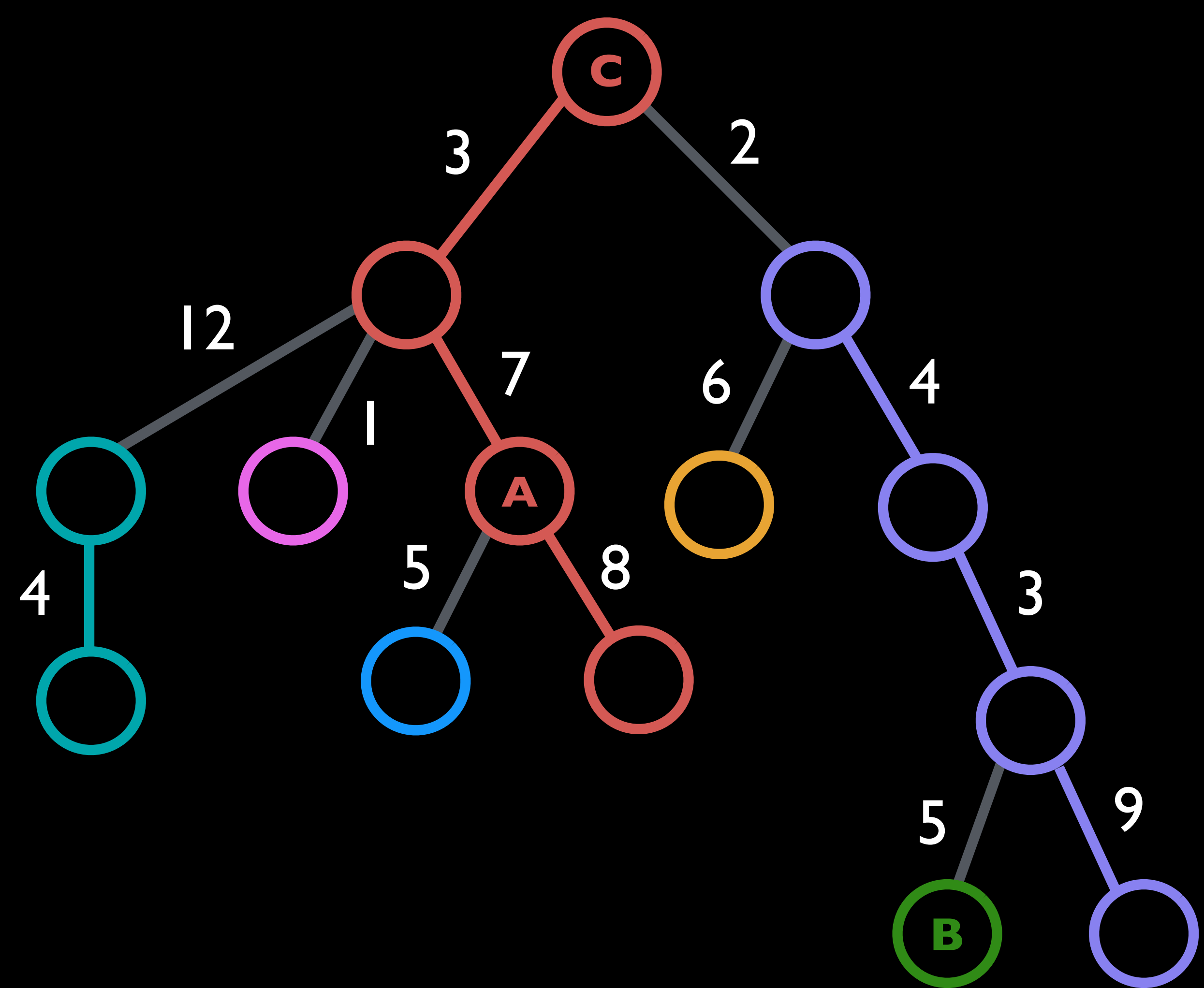
Max Edge Query

- We can use a Fenwick Tree for each of the chains in the tree (using the relative depth to the head as the index).



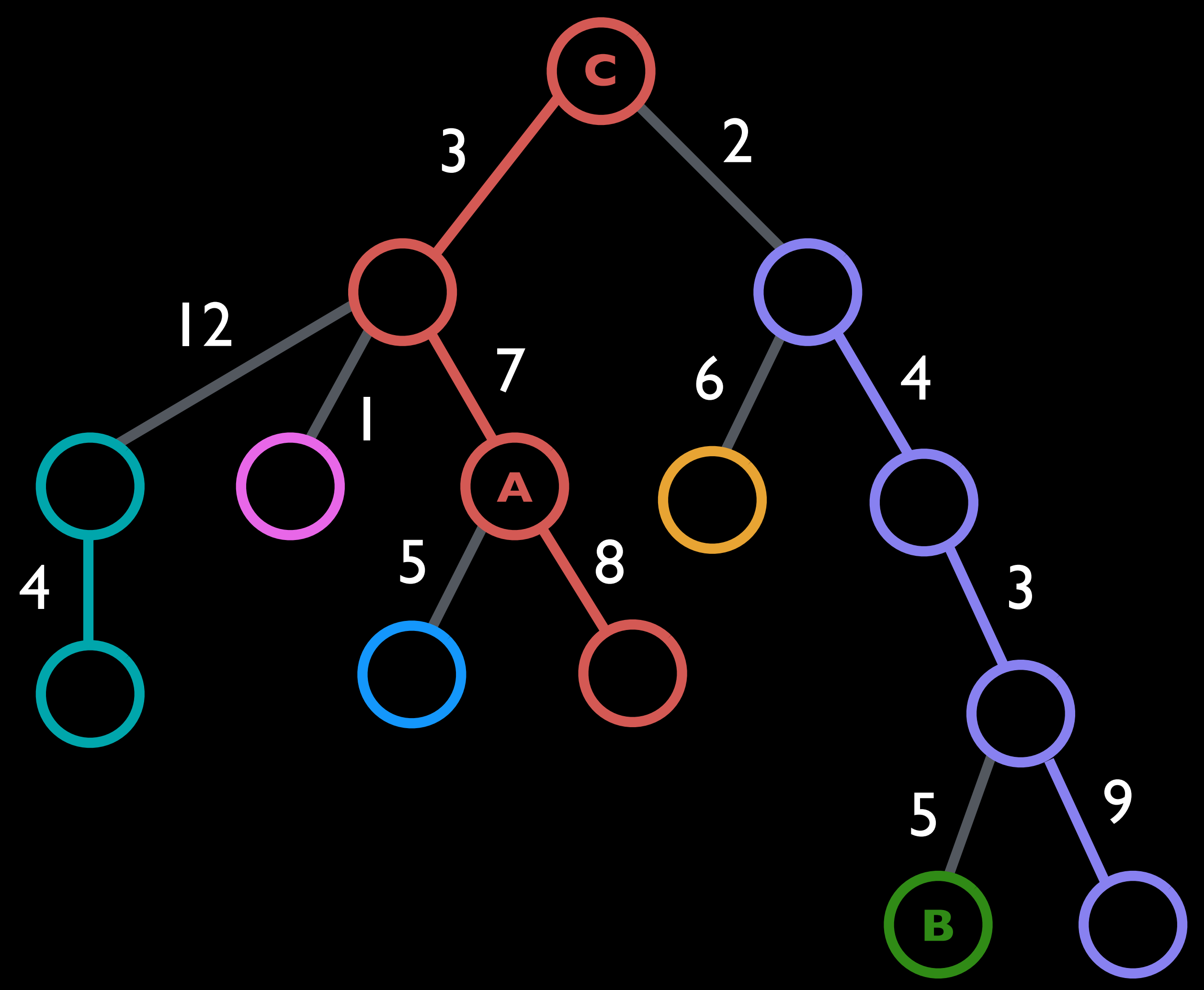
Max Edge Query

- We first find *LCA(A, B)*, calling it C.



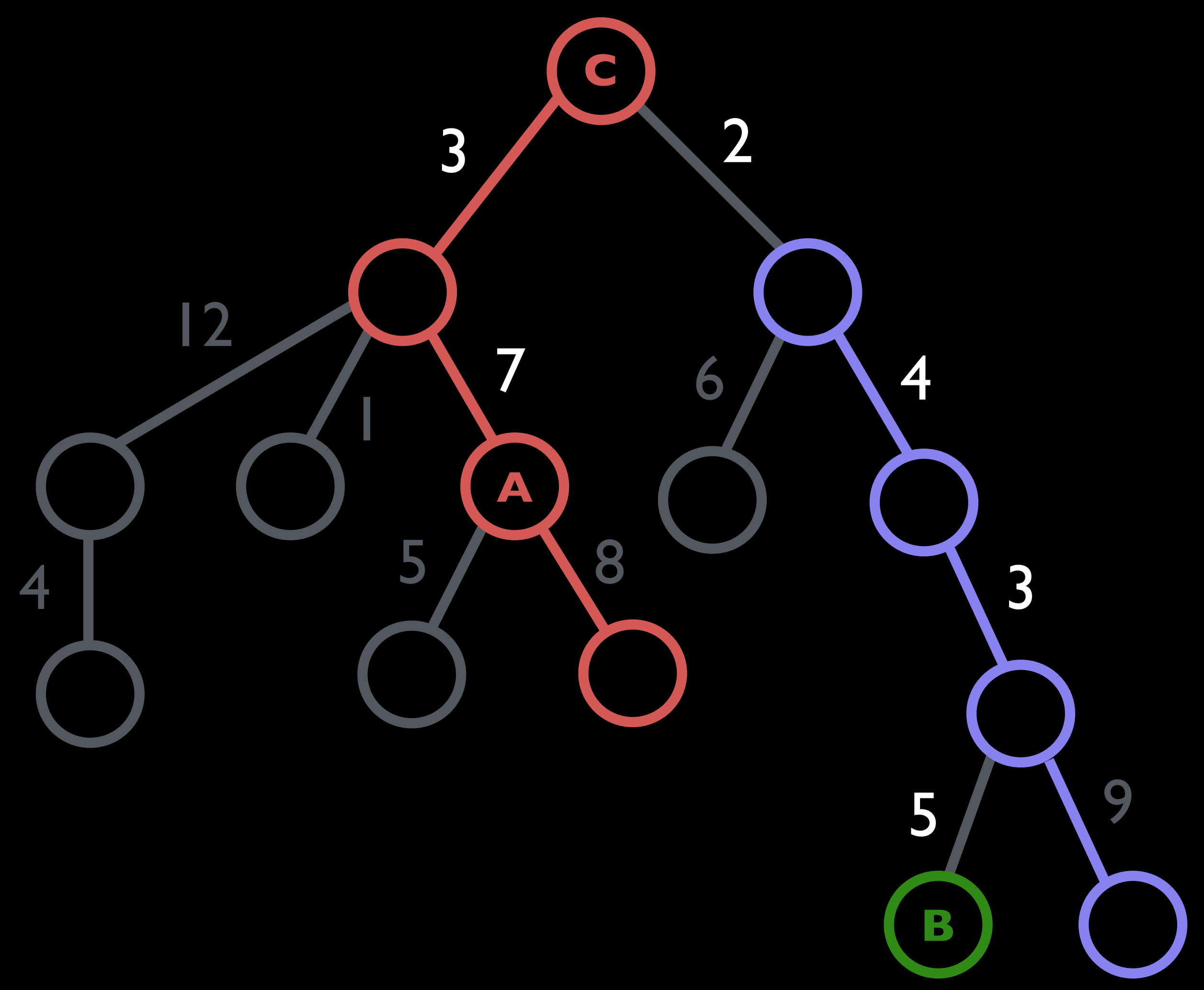
Max Edge Query

- We can find *maxEdge(A,C)* and *maxEdge(B,C)* separately and take the maximum as our answer.



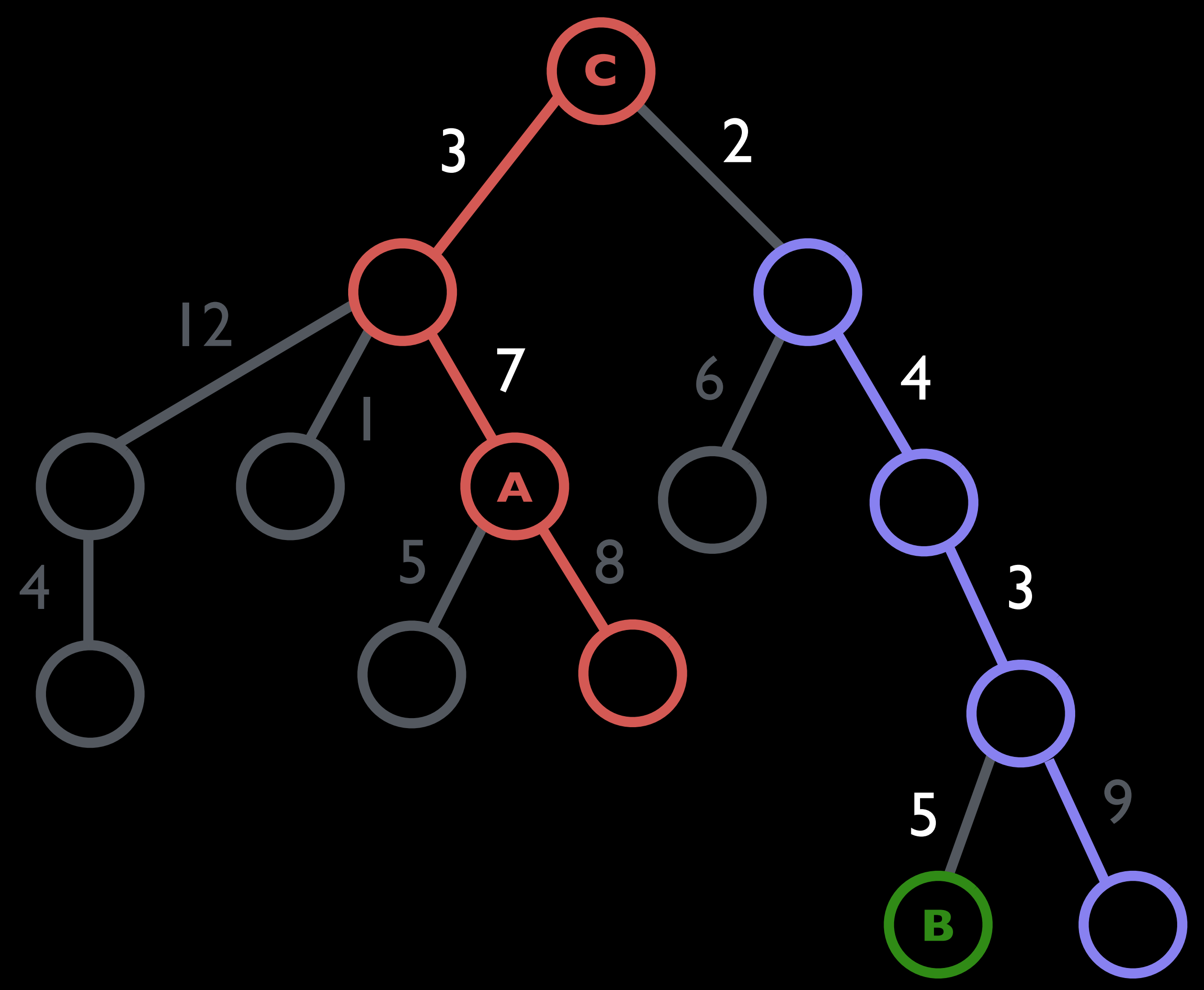
Max Edge Query

- We can find *maxEdge(A,C)* and *maxEdge(B,C)* separately and take the maximum as our answer.



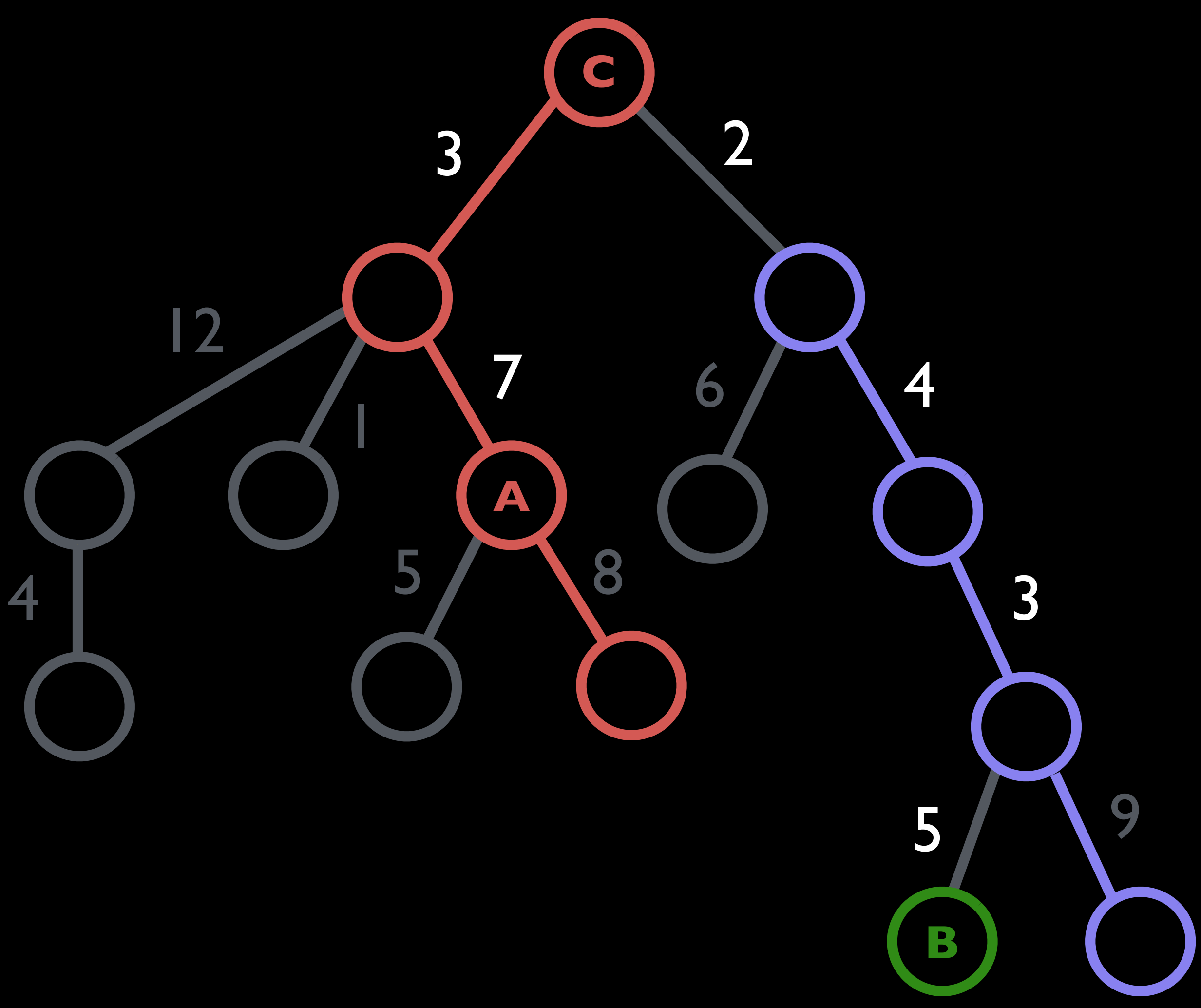
Max Edge Query

- This is done by doing max queries on the Fenwick Trees and also considering the light edges connecting chains.



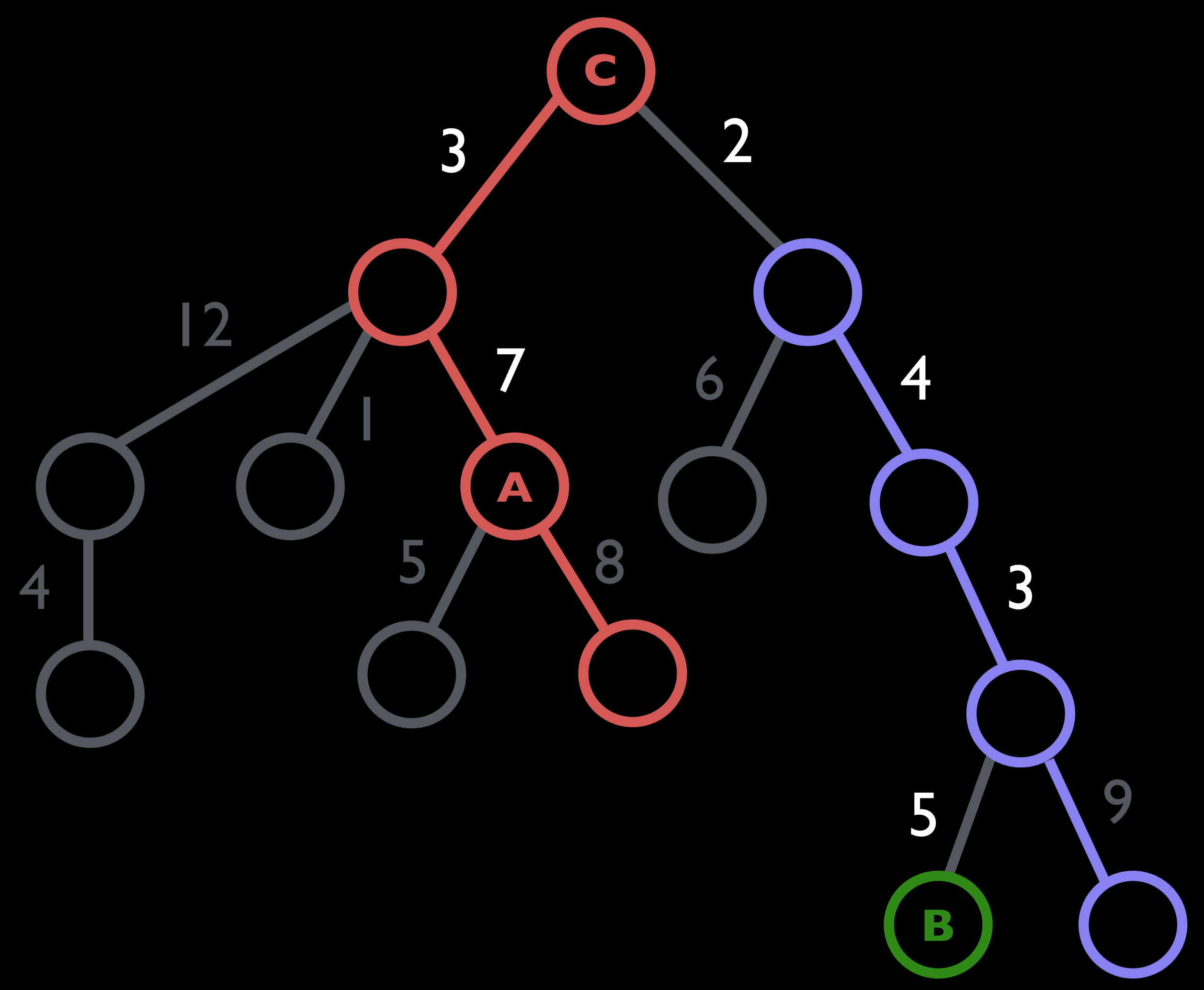
Max Edge Query

- This is also $O(\log^2(n))$ since there are $O(\log(n))$ chains and each query on a Fenwick Tree is $O(\log(n))$.



Max Edge Query

- Since Fenwick Trees are being used, we can efficiently do dynamic updates to the edge weights.



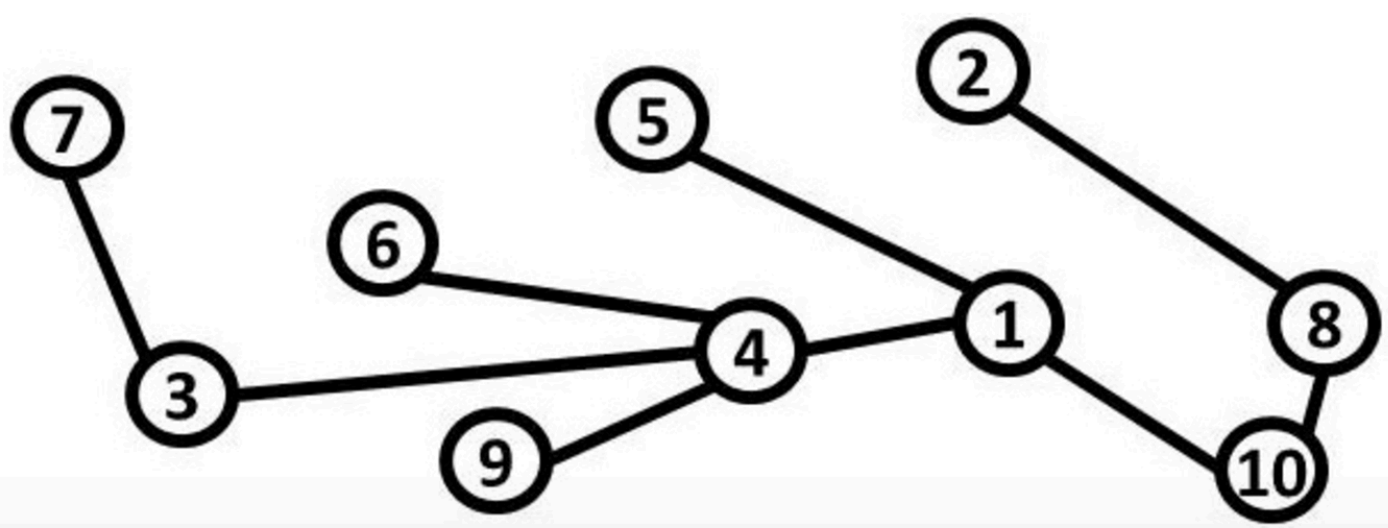
Kattis Problem

Tourists

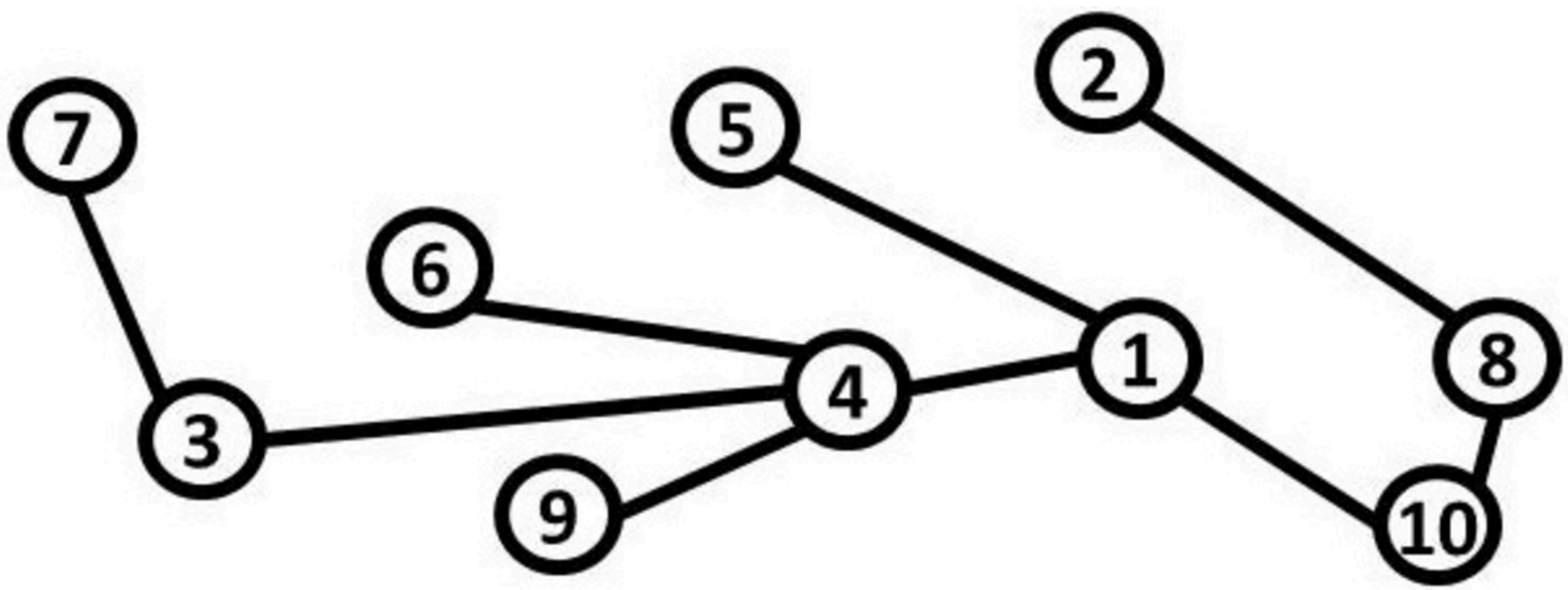
In Tree City, there are n tourist attractions uniquely labeled 1 to n. The attractions are connected by a set of $n-1$ bidirectional roads in such a way that a tourist can get from any attraction to any other using some path of roads.

You are a member of the Tree City planning committee. After much research into tourism, your committee has discovered a very interesting fact about tourists: they LOVE number theory! A tourist who visits an attraction with label x will then visit another attraction with label y if $y > x$ and y is a multiple of x. Moreover, if the two attractions are not directly connected by a road the tourist will necessarily visit all of the attractions on the path connecting x and y, even if they aren't multiples of x. The number of attractions visited includes x and y themselves. Call this the *length* of a path.

Consider this city map:



Kattis Problem



Here are all the paths that tourists might take, with the lengths for each:
$1 \to 2 = 4$, $1 \to 3 = 3$, $1 \to 4 = 2$, $1 \to 5 = 2$, $1 \to 6 = 3$, $1 \to 7 = 4$,
$1 \to 8 = 3$, $1 \to 9 = 3$, $1 \to 10 = 2$, $2 \to 4 = 5$, $2 \to 6 = 6$, $2 \to 8 = 2$,
$2 \to 10 = 3$, $3 \to 6 = 3$, $3 \to 9 = 3$, $4 \to 8 = 4$, $5 \to 10 = 3$

To take advantage of this phenomenon of tourist behavior, the committee would like to determine the number of attractions on paths from an attraction x to an attraction y such that $y > x$ and y is a multiple of x. You are to compute the **sum** of the lengths of all such paths. For the example above, this is: $4+3+2+2+3+4+3+3+2+5+6+2+3+3+3+4+3 = 55$.

Kattis Problem

Input

Each input will consist of a single test case. Note that your program may be run multiple times on different inputs. The first line of input will consist of an integer n $(2 \leq n \leq 200\,000)$ indicating the number of attractions. Each of the following $n-1$ lines will consist of a pair of space-separated integers i and j ($1 \leq i < j \leq n$), denoting that attraction i and attraction j are directly connected by a road. It is guaranteed that the set of attractions is connected.

Output

Output a single integer, which is the sum of the lengths of all paths between two attractions x and y such that $y > x$ and y is a multiple of x.

Solution

- We start by doing LHD on the tree. Since the tree is not rooted, we can pick an arbitrary root.
- We then iterate over each x, y pair that we were asked to consider, taking the sum of the distances between each pair of nodes.

Solution

```java
class Node {

  // List of children
  List<Node> adj = new ArrayList<Node>();

  // These are set internally
  Node parent;
  int size, depth;
  Node head;

  // Do decomposition
  static void setup(Node root) { . . . }

  // Used to set depth and get total subtree size
  private int getSize(Node parent, int depth) { . . . }

  // Used to set up chains
  private void buildChains(Node head) { . . . }

  // Return the distance between two nodes in the tree
  static int dist(Node a, Node b) { . . . }

  // Find the lowest common ancestor of two nodes
  static Node lca(Node a, Node b) { . . . }

}
```

Solution

```java
// Do decomposition
static void setup(Node root) {
  root.getSize(null, 0);
  root.buildChains(null);
}

// Used to set depth and get total subtree size
private int getSize(Node parent, int depth) {

  this.parent = parent;
  this.depth = depth;

  size = 1;
  for (Node node : adj)
    if (node != parent)
      size += node.getSize(this, depth + 1);

  return size;

}
```

Solution

```java
// Used to set up chains
private void buildChains(Node head) {

  // Create new chain
  if (head == null) head = this;
  this.head = head;

  // Find heaviest child
  Node heavyChild = null;
  for (Node node : adj)
    if (node != parent)
      if (heavyChild == null || node.size > heavyChild.size)
        heavyChild = node;

  // Extend chain along heavy child, and start new chains for all other children
  for (Node node : adj)
    if (node != parent)
      node.buildChains(node == heavyChild ? head : null);

}
```

Solution

```java
// Used to set up chains
private void buildChains(Node head) {

  // Create new chain
  if (head == null) head = this;
  this.head = head;

  // Find heaviest child
  Node heavyChild = null;
  for (Node node : adj)
    if (node != parent)
      if (heavyChild == null || node.size > heavyChild.size)
        heavyChild = node;

  // Extend chain along heavy child, and start new chains for all other children
  for (Node node : adj)
    if (node != parent)
      node.buildChains(node == heavyChild ? head : null);

}
```

Solution

```java
// Find the lowest common ancestor of two nodes
static Node lca(Node a, Node b) {
  Node tmpA = a;
  while (true) {
    Node tmpB = b;
    while (tmpB != null) {
      if (tmpA.head == tmpB.head) return tmpA.depth < tmpB.depth ? tmpA : tmpB;
      if (tmpB.head == tmpB) tmpB = tmpB.parent;
      else tmpB = tmpB.head;
    }
    if (tmpA.head == tmpA) tmpA = tmpA.parent;
    else tmpA = tmpA.head;
  }
}

// Return the distance between two nodes in the tree
static int dist(Node a, Node b) {
  return a.depth + b.depth - 2 * lca(a, b).depth;
}
```

Solution

```java
// Read in the number of nodes
int n = Integer.valueOf(br.readLine());

// Setup
Node[] nodes = new Node[n];
for (int i = 0; i < n; i++)
  nodes[i] = new Node();

// Read in edges
for (int i = 1; i < n; i++) {
  String[] line = br.readLine().split(" ");
  int u = Integer.valueOf(line[0]) - 1;
  int v = Integer.valueOf(line[1]) - 1;
  nodes[u].adj.add(nodes[v]);
  nodes[v].adj.add(nodes[u]);
}
```

Solution

```
// Do Light Heavy Decomposition with an arbitrary root
Node.setup(nodes[0]);

// Sum paths
long sum = 0;
for (int i = 1; i <= n; i++)
  for (int j = i * 2; j <= n; j += i)
    sum += 1 + Node.dist(nodes[i - 1], nodes[j - 1]);

// Output answer
System.out.println(sum);
```

Solution

```
// Sum paths
long sum = 0;
for (int i = 1; i <= n; i++)
  for (int j = i * 2; j <= n; j += i)
    sum += 1 + Node.dist(nodes[i - 1], nodes[j - 1]);
```

$$\sum_{k=1}^{m} \frac{n}{k} = n\,H_m$$

$H_{200,000} = 12.783$

$200{,}000 * 12.783 = 2{,}556{,}600$