# Lowest Common Ancestor (LCA)

NOIP 2017 (Senior) Preparation Lecture 1

Daniel Yeh

# Definition

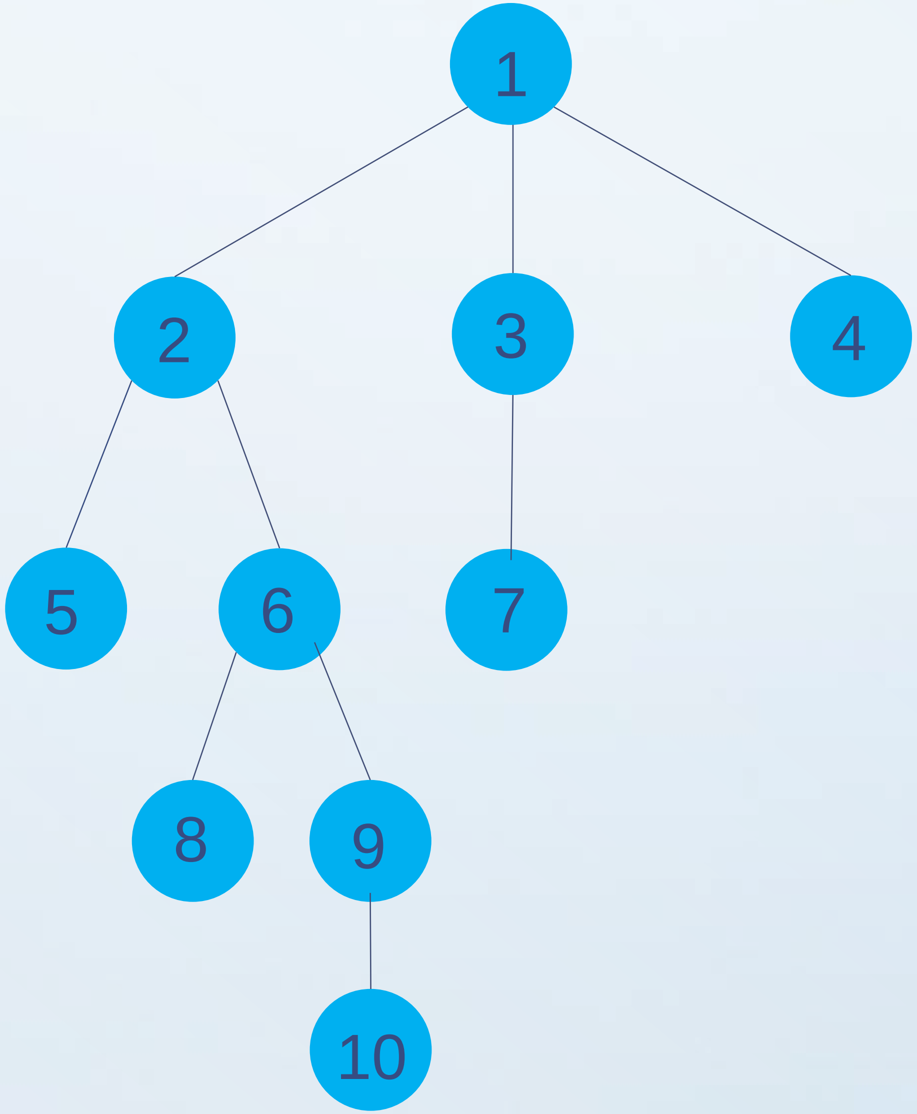
- Ancestor:

In a rooted tree, if u is an ancestor of v, u is on the unique path from node v to the root

- Lowest Common Ancestor (LCA):

The deepest node that is the ancestor of both u and v

# Examples




| Nodes | LCA |
|---|---|
| 2, 4 | 1 |
| 5, 6 | 2 |
| 6, 7 | 1 |
| 3, 7 | 3 |
| 8, 9 | 6 |
| 5, 9 | 2 |
| 2, 9 | 2 |
| 8, 10 | 6 |
| 9, 10 | 9 |

# The LCA Problem

- Given a rooted tree, preprocess it so that the retrieval of LCA of any two given nodes can be done in constant time
- In this lecture, we will discuss on various methods to solve the LCA problem.

# Naïve Solution

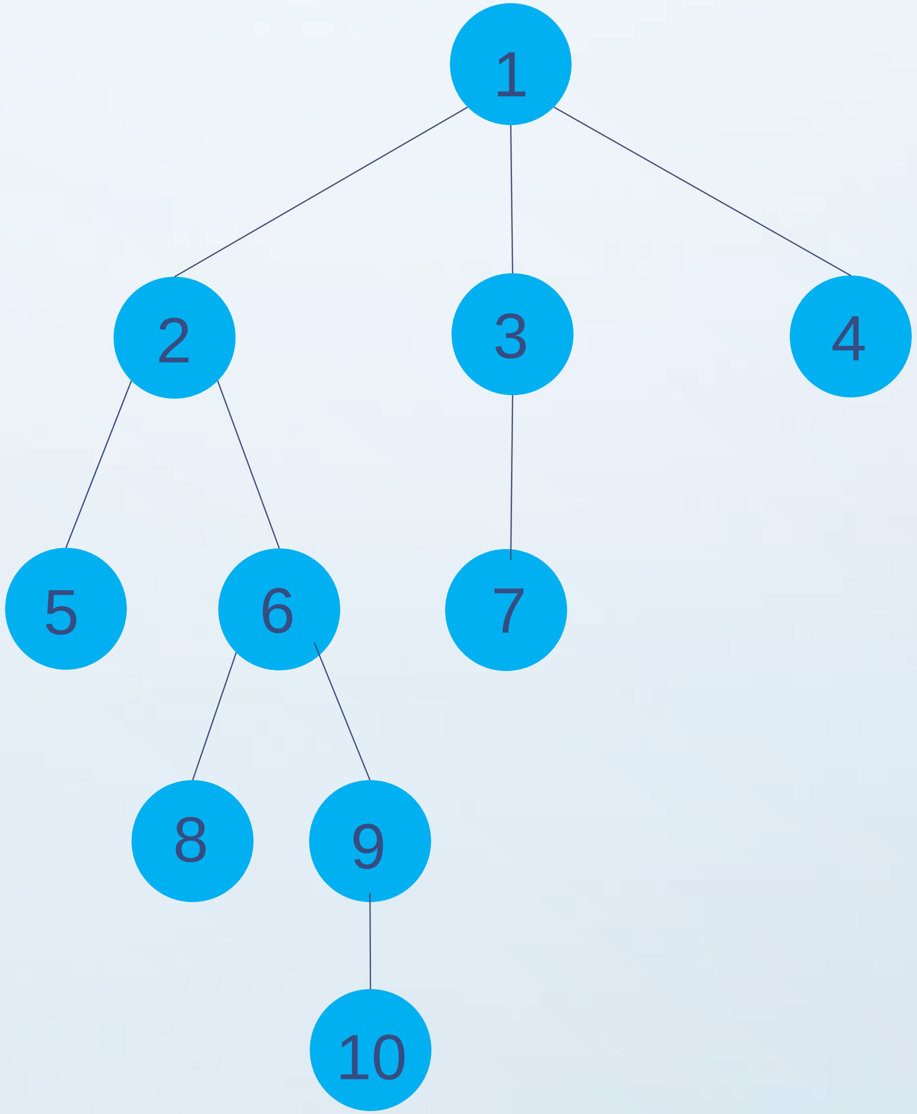
- Put N = Number of nodes
- par[u] = Direct parent of u
- For every pair of u and v:

    pathu = path from u to root

    pathv = path from v to root

- Paths can be found by using “par” array
- For every element of pathu in order, scan once for all elements in pathv for matches
- The first match is the LCA of 2 nodes

# Examples




- Put u = 5, v = 9:
- pathu = {5, 2, 1}
- pathv = {9, 6, 2, 1}
- After iterating each element in pathu, you may find that the first match is 2
- Thus, LCA of 5 and 9 = 2

# Complexity

- Preprocess: **O(N)**
- Per query: **O(N)**
- Time complexity: **<O(N), O(N)>**
- Memory complexity = **O(N)**

- If no. of queries = Q,  Overall time complexity = **O(NQ)**
- Can it be any faster?

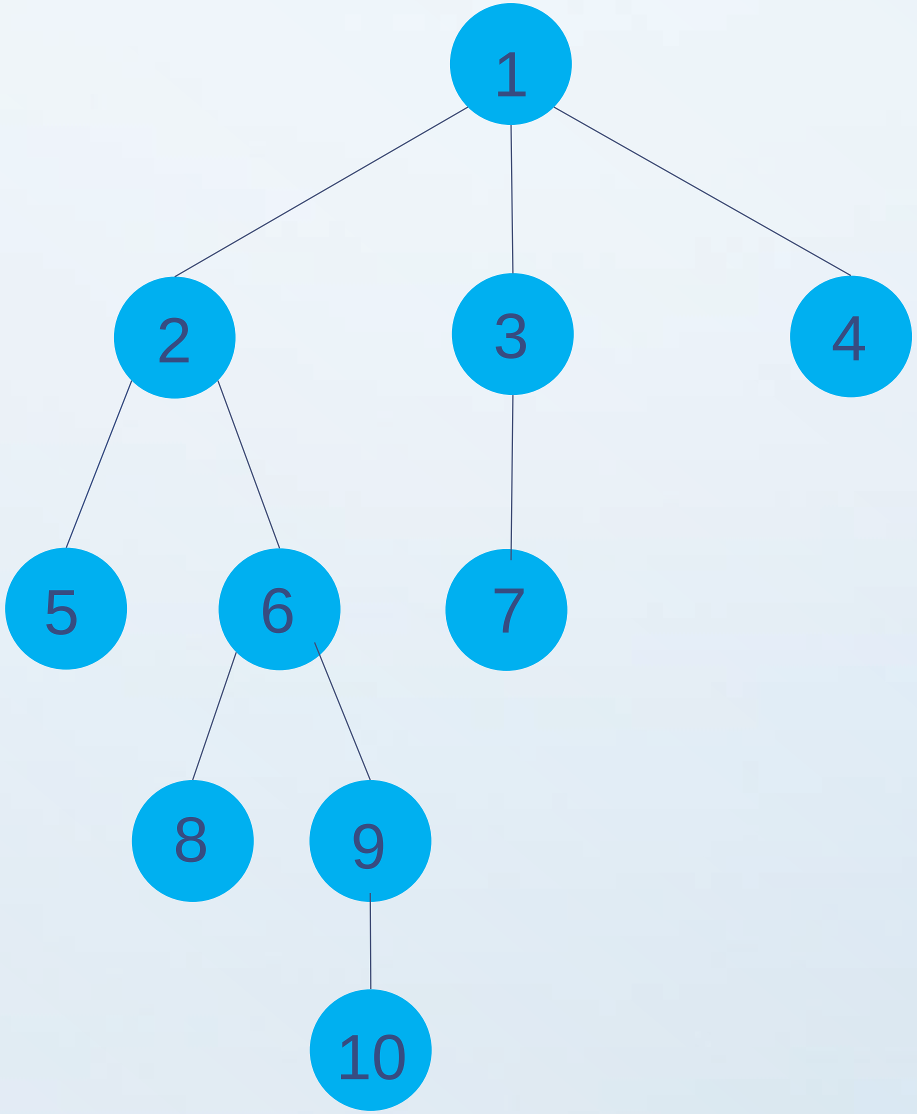
# Brief view on possible methods

- Better methods are available at **O(log N)** for every query
- It is worth noted that we primarily focus on **online** solutions
- Two major methods to achieve O(log N):
  - Sparse Table (Prerequisites: DFS)
  - Range Minimum Query (Prerequisites: DFS, **Segment Tree**)

# LCA by Sparse Table

- Sparse Table is an efficient data structure for fast range queries
- It can be reformed to be used in LCA problem.
- We use an array “st[n][log2(n)]”:
  - st[i][j] = the 2^j-th parent of i
- To find the m-th parent of node u, we use binary representation together with the sparse table

# Examples (eg1)




| ST | 0 | 1 | 2 |
|---|---|---|---|
| 1 | 1 | 1 | 1 |
| 2 | 1 | 1 | 1 |
| 3 | 1 | 1 | 1 |
| 4 | 1 | 1 | 1 |
| 5 | 2 | 1 | 1 |
| 6 | 2 | 1 | 1 |
| 7 | 3 | 1 | 1 |
| 8 | 6 | 2 | 1 |
| 9 | 6 | 2 | 1 |
| 10 | 9 | 6 | 1 |

# LCA by Sparse Table

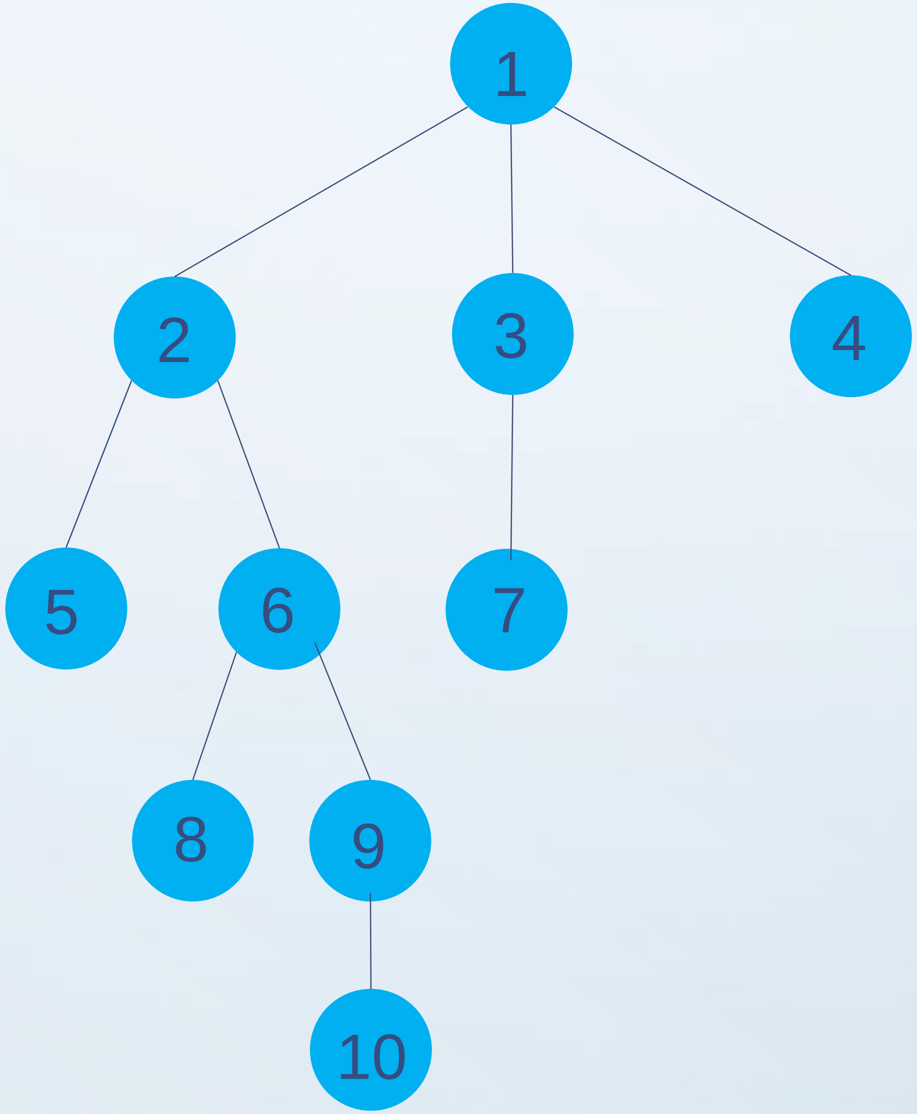
Steps of finding the m-th ancestor of u:

1. Change m into binary representation
2. For every bit (i) of m that is equal to 1, we carry out the following operation: **u = st[u][i]**
3. In the end, u will store the value of the m-th ancestor of the initial u

# LCA by Sparse Table

- Rationale:
- For every 1-bit you carry out the operation, what is implied?
- It can be observed that if you carry out u = st[u][i], it can be interpreted as walking 2^i steps towards the root
- As every integer can be represented in binary notation, by taking correct values of i, the m-th ancestor can be obtained

# Implementation Examples



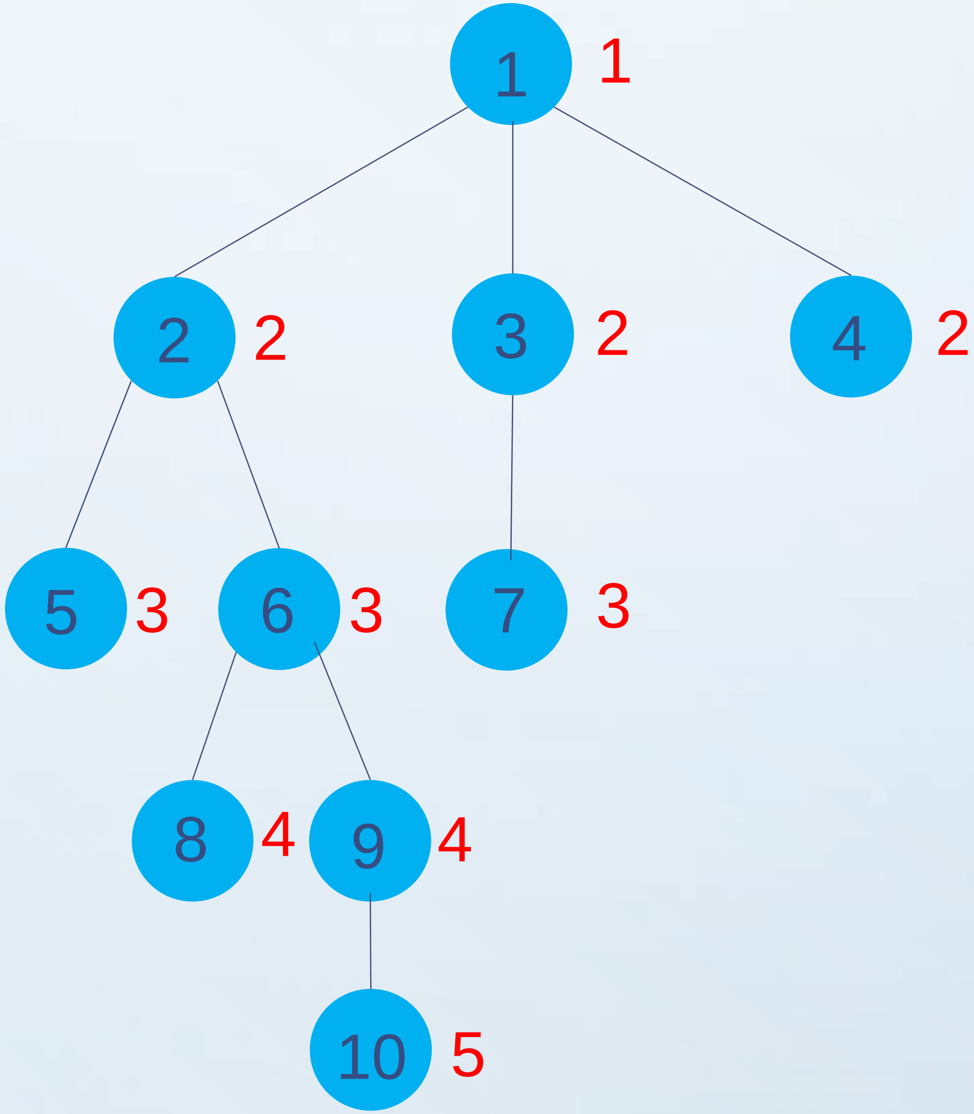

- Q: Find 3-th parent of 10:
- 3 (decimal) = 11 (binary)
- Put u = 10
- 0th-bit = 1:
- u = st[u][0] = st[10][0] = 9
- 1th-bit = 1:
- u = st[u][1] = st[9][1] = 2
- **Third parent of 10 = 2**

(ST)

# LCA by Sparse Table

- We can now find the m-th ancestor of any node in a rooted tree in O(log N)
- How to extend the algorithm for finding the lowest common ancestor of two nodes?
- First, for every node, we assign the depth of each node by running a DFS once.

# Examples




- The root has depth = 1
- All nodes in a rooted tree (except the root) has a depth = depth of parent + 1
- Depths of nodes are marked in **red** in the tree on the left

# LCA by Sparse Table

Given a pair of nodes u and v, we follow the below steps to find their LCA

- Compare the depths between u and v

  If dep[u] < dep[v]: swap(u, v)

- Put d = dep[u] – dep[v]
- Evaluate u1 = d-th ancestor of u
- What’s the aim of this step?
- It makes dep[u1] = dep[v]

# LCA by Sparse Table

- After making dep[u1] = dep[v]:
- First, we check if u1 = v holds

  If u1 = v, then u1 or v is the LCA
- Otherwise, we do sth. similar to binary search using the constructed sparse table

# LCA by Sparse Table

- For every value of i from log2(N) to 0, we check if st[u1][i] = st[v][i]
- If the expression holds, then we continue to loop and check the next i
- Otherwise, we evaluate 2 expressions:

  u1 = st[u1][i];   v = st[v][i];

- Why?

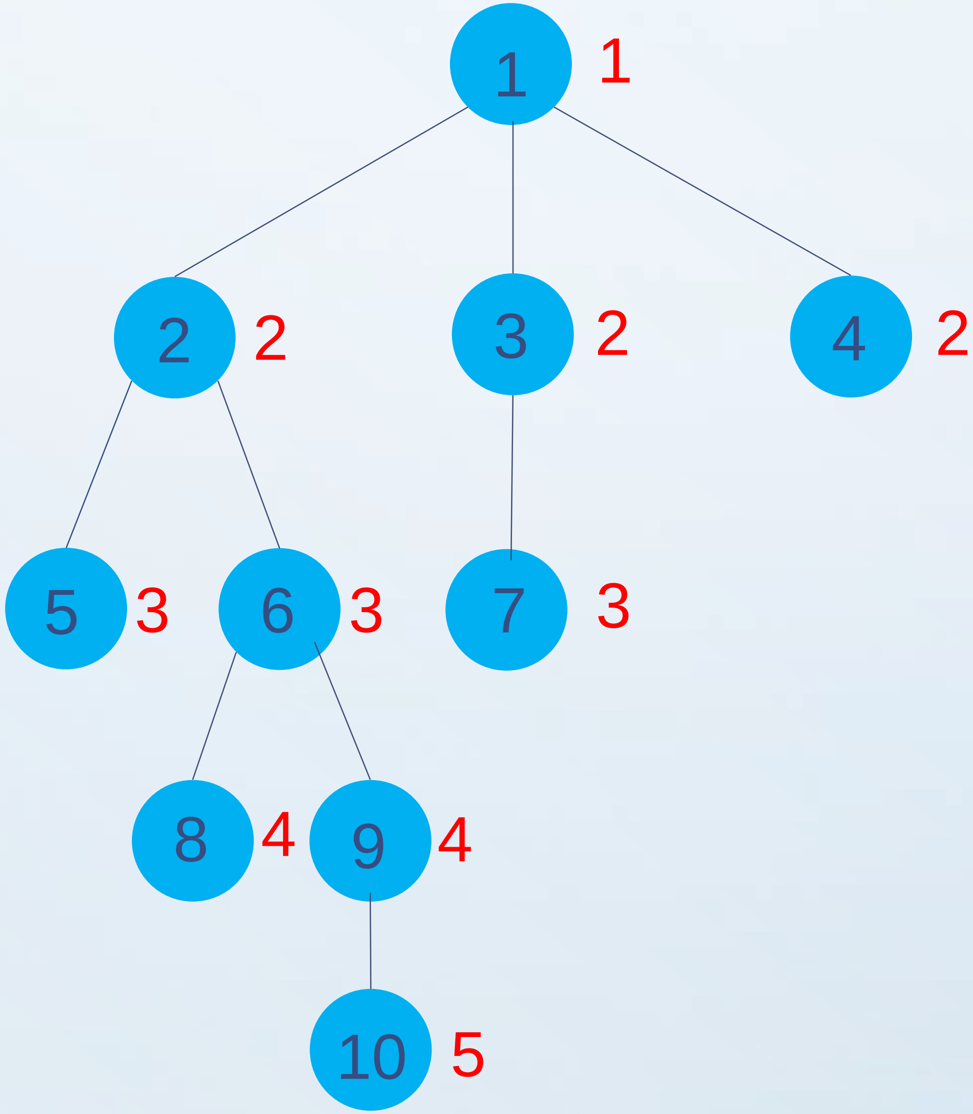
# LCA by Sparse Table

- If st[u1][i] = st[v][i], it implies that the node is a common ancestor of u1 and v, but it does not guarantee that the node is the **lowest** common ancestor
- Otherwise, st[u1][i] != st[v][i], which implies that if both u1 and v goes up 2^i steps, it has not passed through any ancestors of both u1 and v yet
- After carrying out the loop, LCA = st[u1][0] or st[v][0]
- (As after carrying out the lifting with sparse table, both u1 and v are still different, and only 1 step away from their LCA)

# LCA by Sparse Table

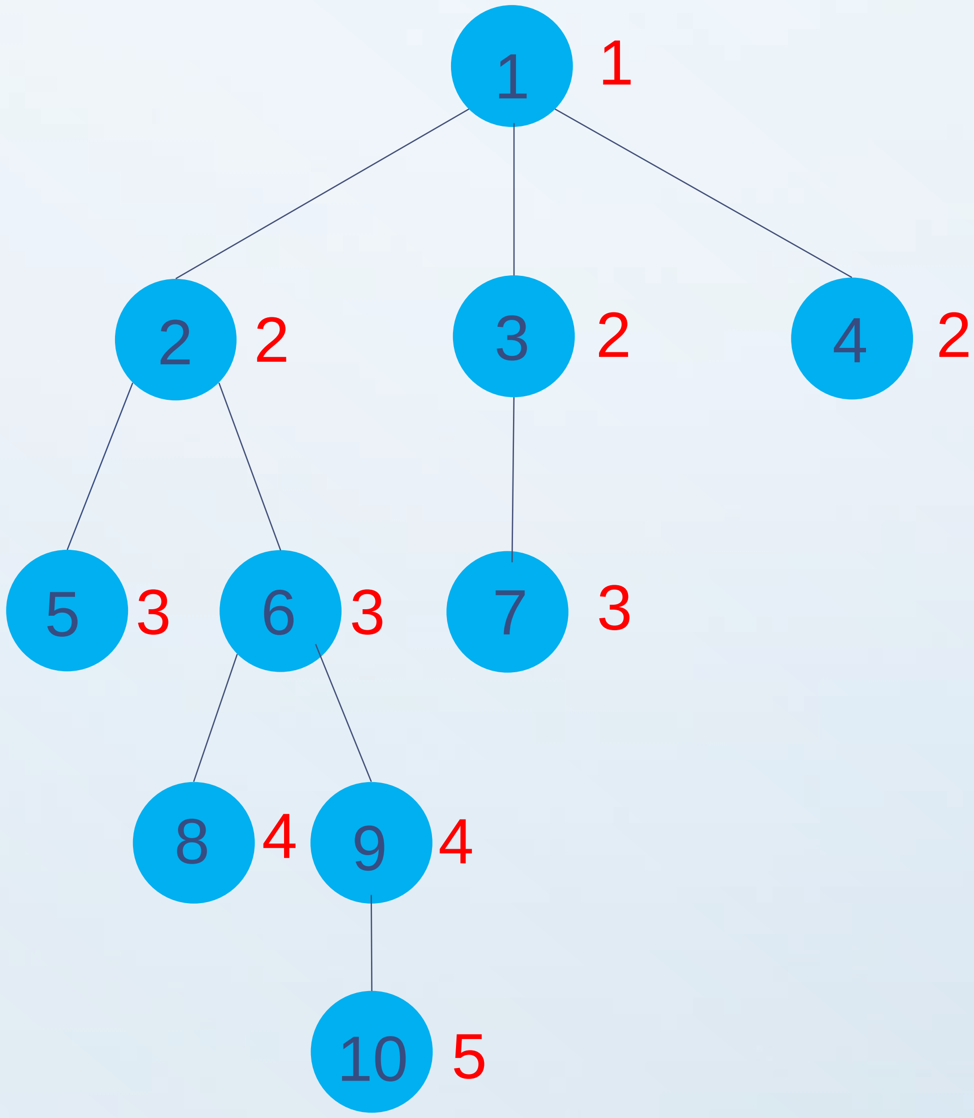
- Hard to understand?
- Let’s view an example!

# LCA by Sparse Table



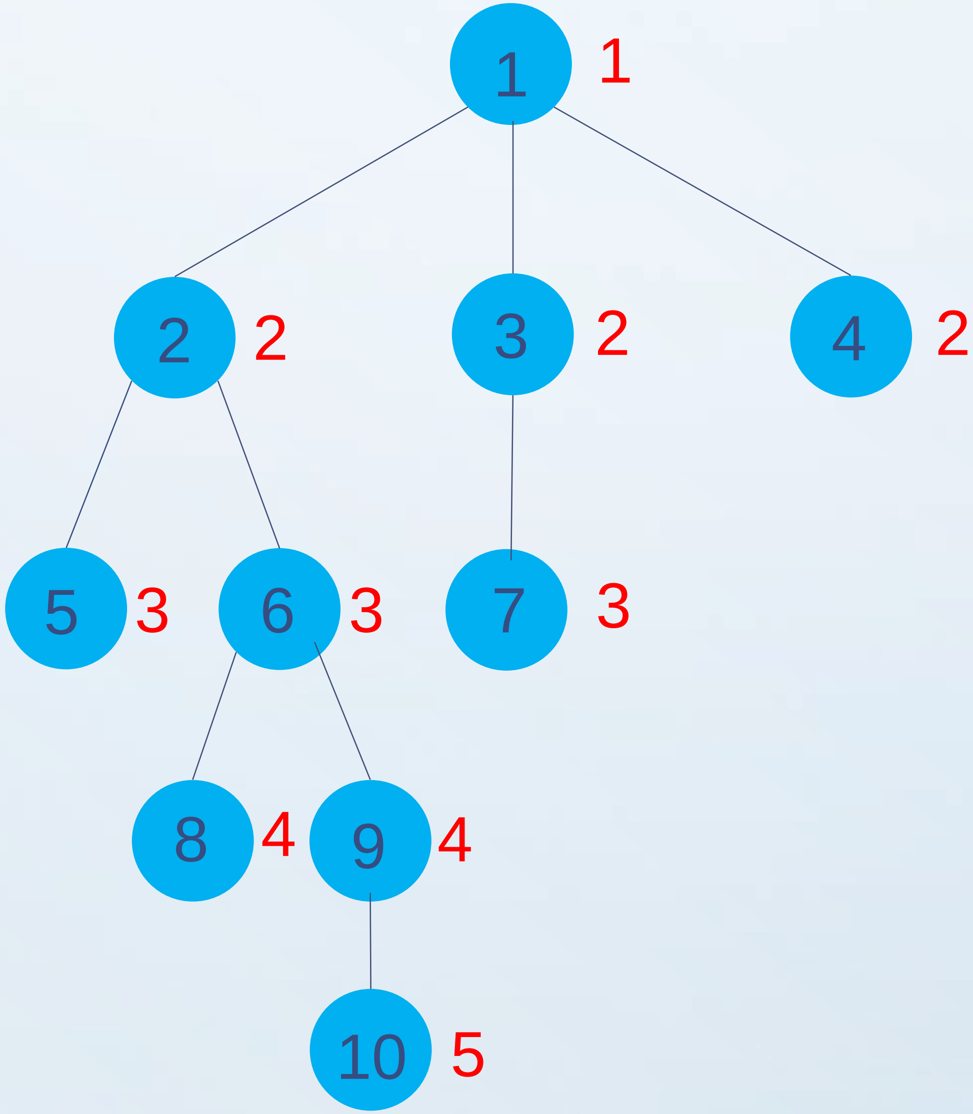

- Let’s find the LCA of node 5 and 10
- Put u = 5, v = 10
- As dep[u] < dep[v]: swap(u, v)
- Thus, u = 10, v = 5
- Evaluate d = dep[u] – dep[v] = 2
- 2(decimal) = 10(binary)

# LCA by Sparse Table



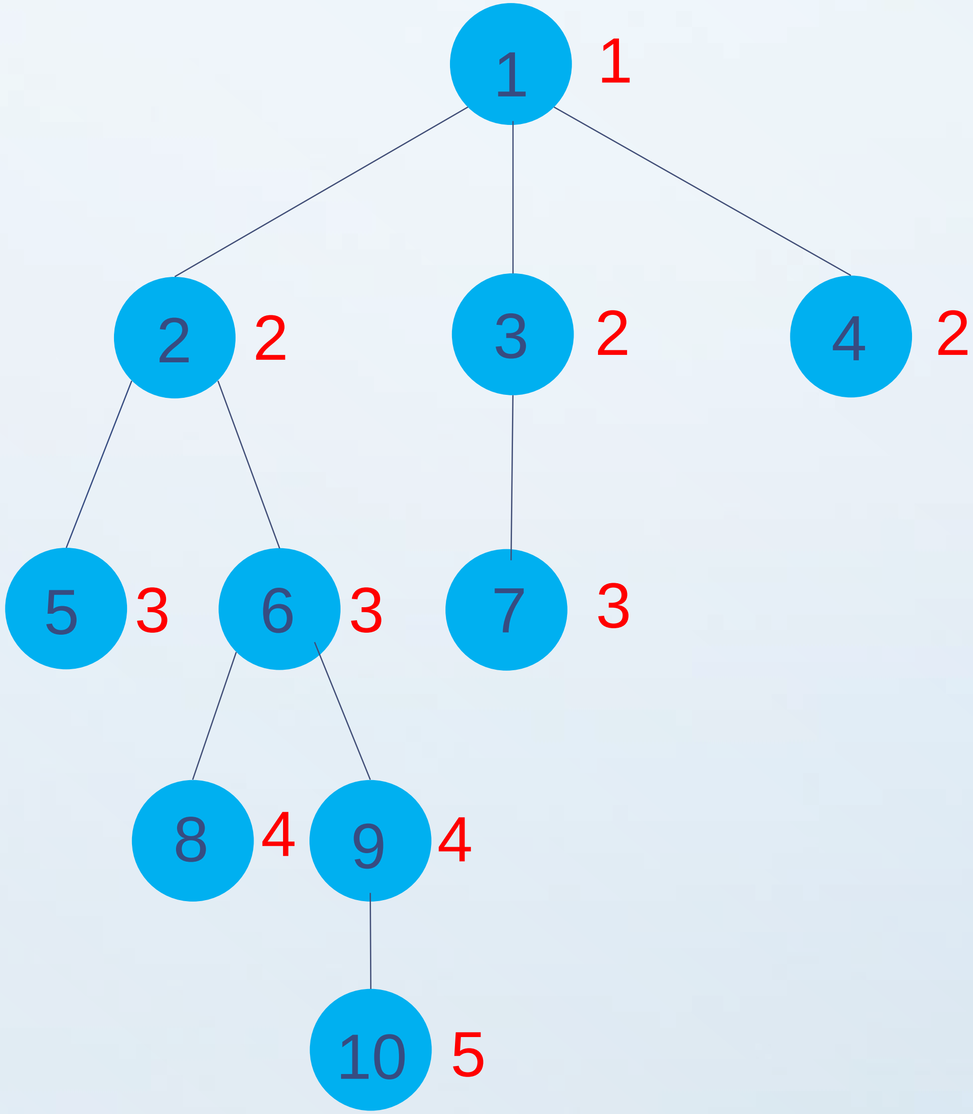

- As the 1-th bit = 1:
- u1 = st[u][1] = 6
- Now u1 = 6, v = 5
- We loop i from log2(N) = 2 to 0:
- When i = 2:
  - st[u1][i] = st[v][i] = 1: Consider next case
- When i = 1:
  - st[u1][i] = st[v][i] = 1: Consider next case

# LCA by Sparse Table



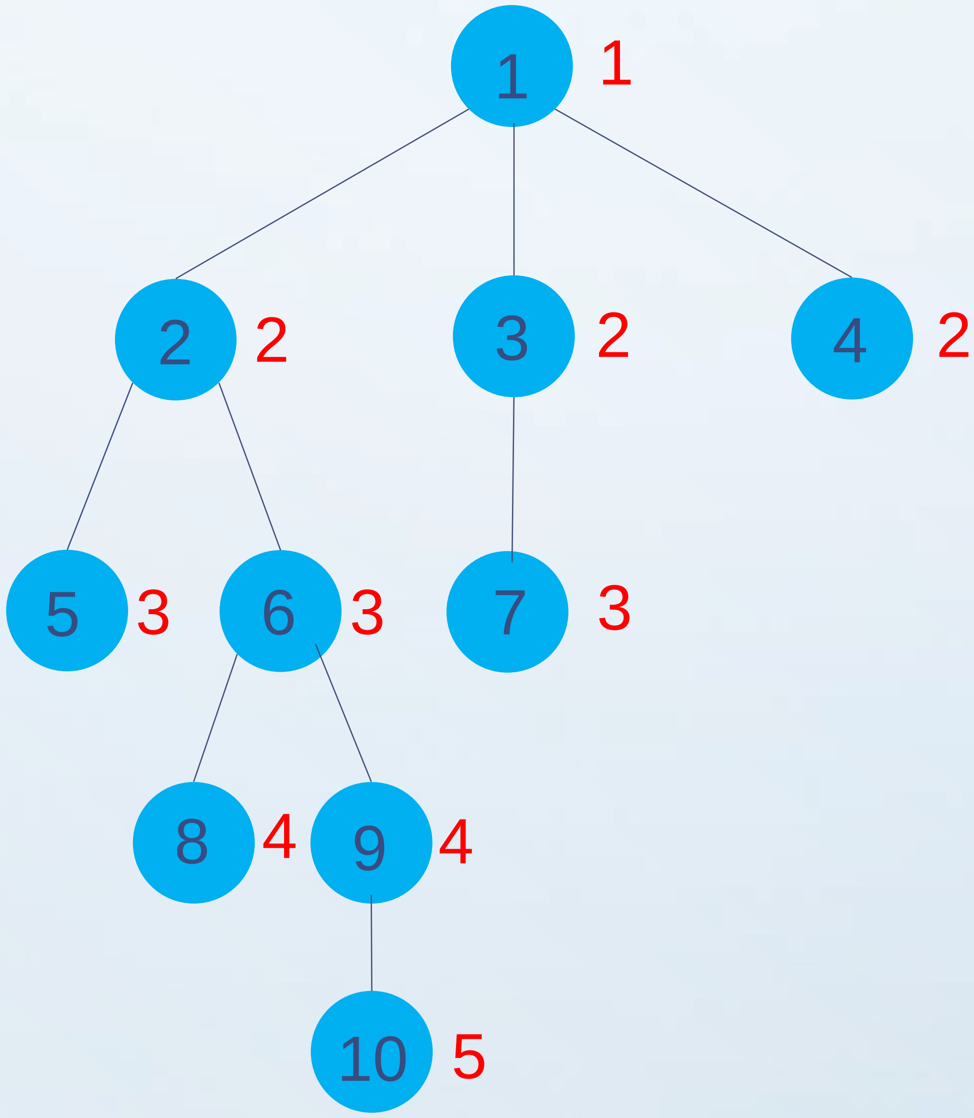

- When i = 0:
  - st[u1][i] = st[v][i] = 2: Consider next case
- LCA of 5 and 10 = st[u1][0] = **2**
- Can you feel the beauty of Sparse Table?

# LCA by Sparse Table



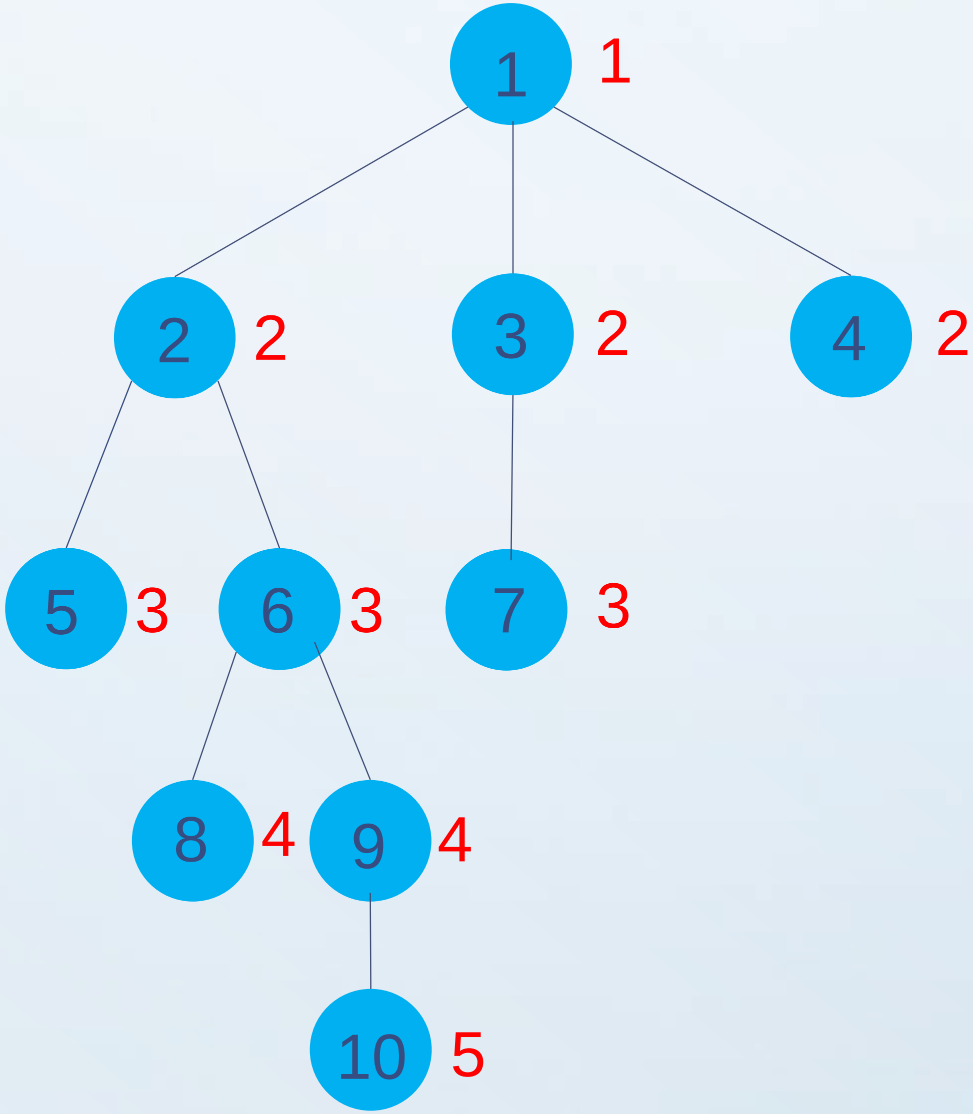

- One more example?
- Find the LCA of 7 and 8:
- Put u = 7, v = 8
- As dep[u] < dep[v]: swap(u, v)
- u = 8, v = 7
- d = dep[u] – dep[v] = 1
- 1(decimal) = 1(binary)
- u1 = st[u][0] = 6

# LCA by Sparse Table




- u1 = 6, v = 7
- Iterate i from log2(N) = 2 to 0:
- When i = 2:
  - st[u1][i] = st[v][i] = 1: continue to next case
- When i = 1:
  - st[u1][i] = st[v][i] = 1: continue to next case

# LCA by Sparse Table




- When i = 0:
  - st[u1][i] != st[v][i]:
  - u1 = st[u1][0] = 2
  - v = st[v][0] = 3
- LCA of 7 and 8 = st[u1][0] = **1**
- Understand now?
- Let’s take a look at the implementation part!

# Construct the Sparse Table

- Through DFS once, we can assign **dep[u]** and **st[u][0]** at the same time for every node in the rooted tree (remember <u>dep[root] = 1, st[root][0] = root</u>)

```
/* dep[root] = 1
   st[root][0] = 1*/
void dfs (int u, int p) {
  for (auto v : edges[u]) {
    if (v == p) continue;
    dep[v] = dep[u] + 1;
    st[v][0] = u;
    dfs(v, u);
  }
  return;
}
```

# Construct the Sparse Table

- Constructing the sparse table after DFS is easier than you think
- Only a five-liner code is needed
- It bases on this formula: st[u][i + 1] = st[st[u][i]][i]
- Why?
- Going upwards 2^(j)

steps = Go upwards

2^(j – 1) steps, then go

2^(j – 1) steps again

```
void cST (int n) {
  for (int j = 1; j <= log2(n); j++) {
    for (int i = 1; i <= n; i++) {
      st[i][j] = st[st[i][j - 1]][j - 1];
    }
  }
  return;
}
```

# Finding the LCA

- The implementation is identical to the description
- Note that d >> i means removing the i bits on the right of the binary notation of d

  e.g. (1010 >> 1) = 101

```
int lca (int n, int u, int v) {
  if (dep[u] < dep[v]) swap(u, v);
  int d = dep[u] - dep[v];
  for (int i = log2(n); i >= 0; i--) {
    if ((d >> i) & 1) u = st[u][i];
  }
  if (u == v) return u;
  for (int i = log2(n); i >= 0; i--) {
    if (st[u][i] != st[v][i]) {
      u = st[u][i]; v = st[v][i];
    }
  }
  return st[u][0];
}
```

# Complexity

- What is the complexity?
- Preprocess: **O(N log N)**
- Per query: **O(log N)**
- Time complexity: **<O(N log N), O(log N)>**
- Memory complexity: **O(N log N)**
- Overall time complexity: O(N log N + Q log N) = **O((N+Q) log N)**

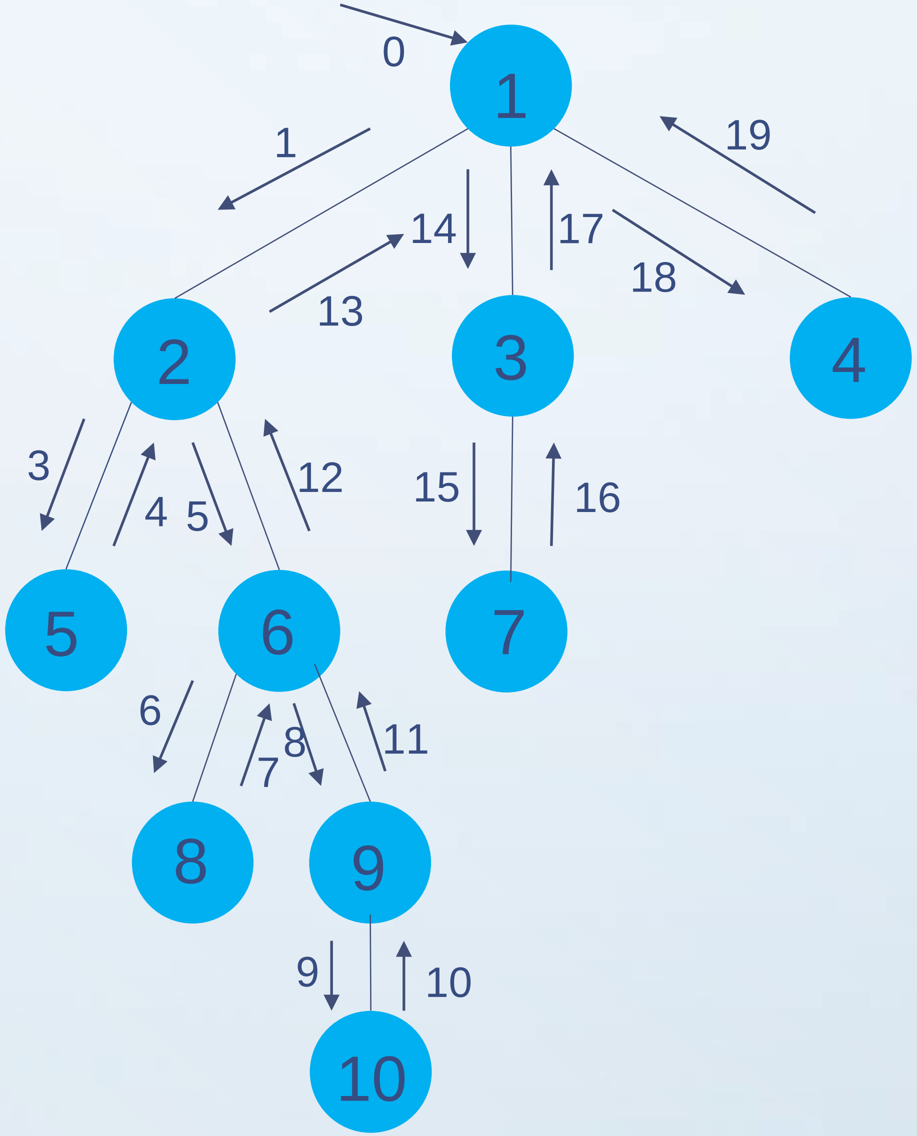
# Are there any other methods?

- Yes, absolutely!
- The next one to be introduced is by Range Minimum Query (RMQ)
- The time and memory complexities would have slight differences
- Special prerequisites: **Segment Tree** and Point Update

# LCA by RMQ

- LCA by RMQ uses a segment tree to consider depth of nodes to get the answer.
- To make use of RMQ to find the LCA, we must understand what the **Euler tour** for a tree is
- The Euler tour is the way of representing trees. It is similar to a DFS transversal of a tree, but it leaves records every time when a node is visited.
- The next slide will show a more intuitive example.

# Euler Tour examples




- The DFS transversal is shown by arrows, where numbers on arrows show the time when the edge is visited
- The Euler tour is as follows:

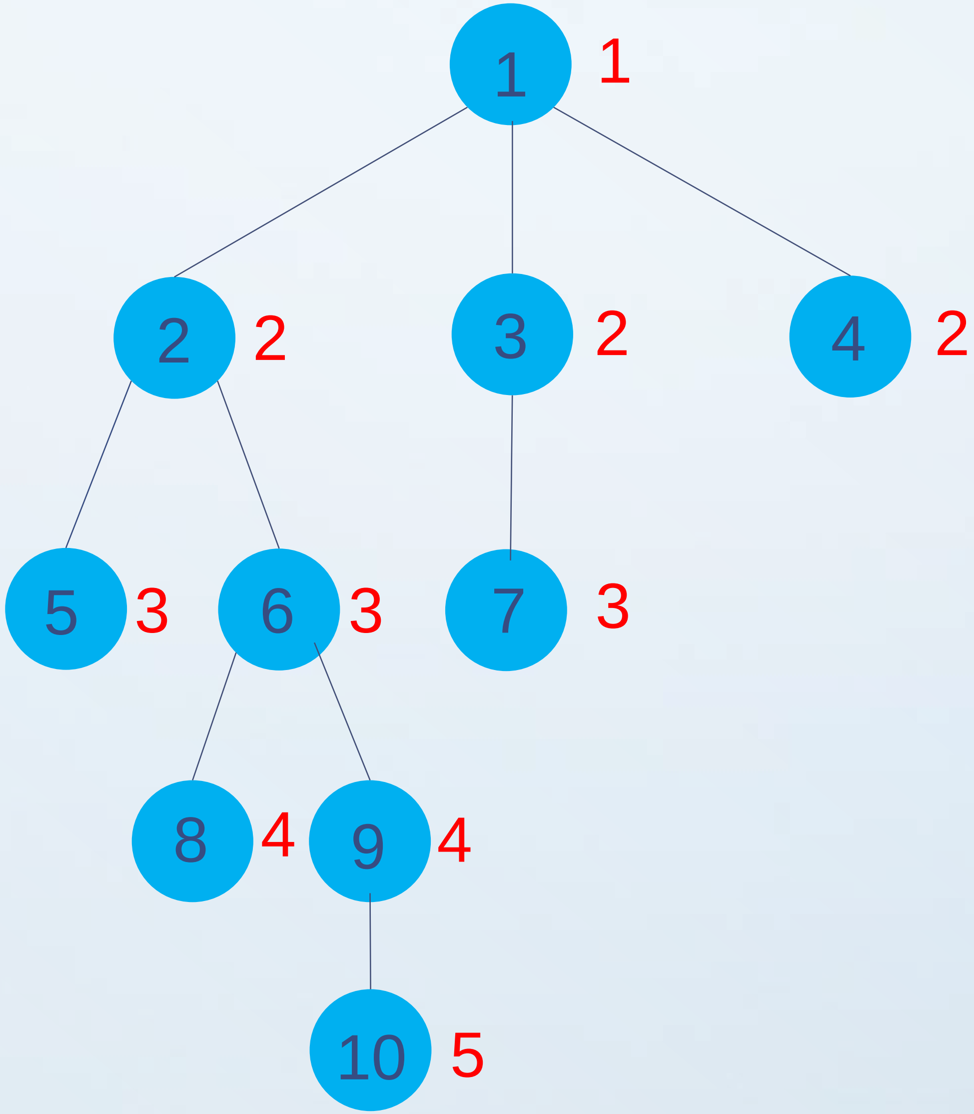
[1, 2, 5, 2, 6,   8, 6, 9, 10, 9,

6, 2, 1, 3, 7,   3, 1, 4, 1]

# LCA by RMQ

- Why we need the Euler tour?
- Observation: The LCA of two nodes, u and v, is the **shallowest node** (the node with smallest depth) on the path between the visits from u to v (or from v to u) during a DFS.
- Remarks:
  - Size of Euler tour = 2n – 1
  - To assure the concreteness when evaluating the LCA, we shall only consider the first occurrences for every node
  - LCA = shallowest node on the path from u to v

# LCA by RMQ examples




- LCA(5, 10) = 2

[1, 2, 5, **2**, 6, 8, 6, 9, 10, 9,

6, 2, 1, 3, 7, 3, 1, 4, 1]

- LCA(9, 7) = 1

[1, 2, 5, 2, 6, 8, 6, 9, 10, 9,

6, 2, **1**, 3, 7, 3, 1, 4, 1]

# LCA by RMQ

- The idea is simple enough, right?
- Now let’s take a look at the implementation part
- It requires a efficiently-implemented and simple-practised segment tree to ensure the runtime.
- The length of code is quite similar to that of Sparse Table.

# Construct Euler's tour

- We first DFS the tree once to assign depths and build the Euler's tour
- Please remember to put u again in the Euler's tour when you leave u

```
int it = 262143;
/* dep[root] = 1 */
void dfs (int u, int p) {
  for (auto v : edges[u]) {
    if (v == p) continue;
    dep[v] = dep[u] + 1;
    eul[++it] = u;
    dfs(v, u);
  }
  eul[++it] = u;
  return;
}
```

# Construct Segment Tree

- Next, we build the segment tree for Range Minimum Query as usual.
- Note that we consider the depth of nodes instead of the node ids!

```
void buildSegTree () {
  for (int i = 262143; i >= 1; i--) {
    eul[i] = (dep[eul[i * 2]] < dep[eul[i * 2 + 1]]) ? eul[i * 2] : eul[i * 2 + 1];
  }
  return;
}
```

# Record First Occurrence

- Recall that during queries, we only consider the first occurrence of every node in the RMQ
- occ[u] stores the index of first occurrence of u in the eul array

```
void buildOcc () {
  SET(occ, -1);
  for (int i = 262144; i <= it; i++) {
    if (occ[eul[i]] == -1) {
      occ[eul[i]] = i - 262143;
    }
  }
  return;
}
```

# Segment Tree Query

- Now we have come to the query part

```
int gl, gr;
int query (int l, int r, int i) {
  if (r < gl || l > gr) return 0;
  if (gl <= l && r <= gr) return eul[i];
  int m = (l + r) / 2;
  int n1 = query(l, m, i * 2), n2 = query(m + 1, r, i * 2 + 1);
  return (dep[n1] < dep[n2]) ? n1 : n2;
}
```

# Complexity

- Preprocess: **O(N)**
- Query: **O(log N)**
- Time complexity: **<O(N), O(log N)>**
- Memory complexity: **O(N)** (actually 2N)
- Overall time complexity: **O(N + Q log N)**

# Comparison between the 2 methods

| | Sparse Table | RMQ |
|---|---|---|
| Preprocess | O(N log N) | O(N) |
| Query | O(log N) | O(log N) |
| Memory complexity | O(N log N) | O(N) |
| Overall time complexity | O((N + Q) log N) | O(N + Q log N) |
| Code length | 31 lines | 34 lines |

- It is rather obvious that RMQ has a slightly better performance than Sparse Table
- For better performance, we prefer using RMQ instead of Sparse Table

# Comparison between the 2 methods

| | Sparse Table | RMQ |
|---|---|---|
| Advantages | More intuitive (easier to understand) | Better performance in both runtime and memory |
| | Easier to be extended to other types of queries | Implementation is more standardized (as segment tree is used as the major tool) |
| Disadvantages | Sometimes, with strict runtime and memory constraints, Sparse Table may not pass the tests | Harder to code for those who are not familiar with the use of segment tree |

# Using the LCA to find Distances

- In often times, we are required to find the distance from u to v in a rooted weighted tree
- It is highly related to the LCA problem
- Why?
- The problem is based on an important observation:

**“To travel from u to v in minimum distance, we must pass through their LCA”**

- So the following formula can be derived:

dist(u, v) = dist(u, lca(u, v)) + dist(lca(u, v), v)

# Using the LCA to find Distances

- So how are we going to use the formula?
- We implements an idea similar to partial sum:
- We open one more array “pdist” to store the distance from the root to the node for every node, which can be easily done by DFS once
- Then distance from ancestor (u) to descendants (v) (or vice versa) = **pdist[v] – pdist[u]**

# Using the LCA to find Distances

- So we can derive the following extension algorithm:
- dist(u, v) = dist(u, lca(u, v)) + dist(lca(u, v), v)

  = pdist[u] – pdist[lca(u, v)] + pdist[v] – pdist[lca(u, v)]

  = pdist[u] + pdist[v] – pdist[lca(u, v)] * 2
- Once obtain the LCA, **O(1)** to find the distance
- Clearer idea now?

# Getting tired?

- Methods up till now are good enough for most competitive programming contests
- However, at times when your program runtime exceed a little by the limit, it can be annoying
- The slides after this are optional. Interested readers can continue

# Can the above ways be optimised?

- Yes
- Runtime can be further optimised a lot
- There’s a considerably easy optimisation, which requires a combination of Sparse Table and RMQ (without using Segment Tree)

# RMQ by Sparse Table

- Recall the idea of RMQ:

  LCA of u and v = the node in Euler's tour with the smallest depth between first occurrence of u and first occurrence of v

- Instead of Segment Tree, we may use a faster method
- Let's preview the time complexity first:

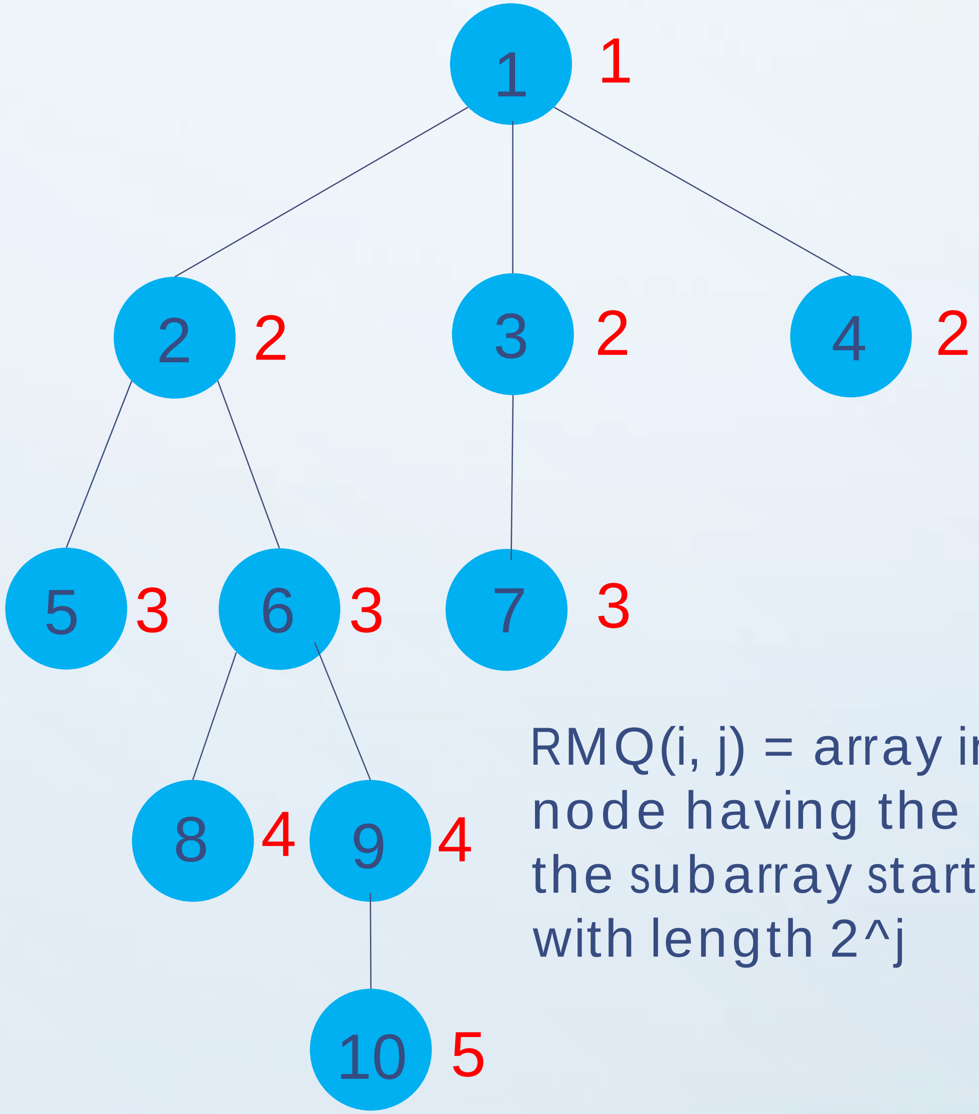
  **<O(N log N), O(1)> / O(N log N + Q)**

- Attractive?

# RMQ by Sparse Table

- In a Sparse Table, we preprocess sub-arrays with length 2^j
- RMQ(i, j) = array index of the node having the min depth in the subarray starting from i, with length 2^j
- Confusing?
- Let’s see an example!

# RMQ by Sparse Table




- First 9 elements in Euler's tour:

| A[1] | A[2] | A[3] | A[4] | A[5] | A[6] | A[7] | A[8] | A[9] |
|---|---|---|---|---|---|---|---|---|
| **1** | **2** | **5** | **2** | **6** | **8** | **6** | **9** | **10** |
| 1 | 2 | 3 | 2 | 3 | 4 | 3 | 4 | 5 |

RMQ(3, 0) = 3 (A[3] = 5, dep[5] = 3)

RMQ(3, 1) = 4 (A[4] = 2, dep[2] = 2)

RMQ(3, 2) = 4 (A[4] = 2, dep[2] = 2)

RMQ(i, j) = array index of the node having the min depth in the subarray starting from i, with length 2^j

# RMQ by Sparse Table

- Intuition:
- Precompute a Sparse Table, storing M(i, j) for every i from 1 to N and every j from 1 to floor(log N)
- Set st[i][j] = RMQ(i, j) = array index of the node having the min depth in the subarray starting from i, with length 2^j
- It makes use of a DP idea:

If dep[ eul[ st[i][j] ] ] < dep[ eul[ st[i + 2^j][j] ] ]:

st[i][j + 1] = st[i][j]

Else: st[i][j + 1] = st[i + 2^j][j]

- Table size (Precompute time complexity) : **N log N**

# RMQ by Sparse Table

| A[1] | A[2] | A[3] | A[4] | A[5] | A[6] | A[7] | A[8] | A[9] |
|---|---|---|---|---|---|---|---|---|
| **1** | **2** | **5** | **2** | **6** | **8** | **6** | **9** | **10** |
| 1 | 2 | 3 | 2 | 3 | 4 | 3 | 4 | 5 |

st[4][1] = 4    st[6][1] = 7

st[4][2] = 4

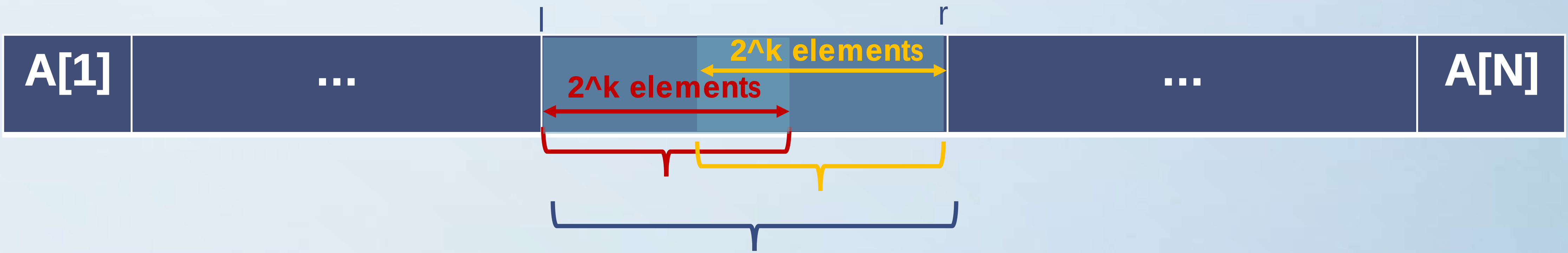
# RMQ by Sparse Table

- Now we have come to the most amazing part
- An **O(1) query** trick will be introduced
- How are we going to find the array index storing the node with smallest depth from l to r?
- We select two blocks entirely covering from l to r (may overlap each other)

# RMQ by Sparse Table

- Let k = floor( log(r – l) )
- Meaning: 2^k is the largest block that entirely fits into the range l to r
- So RMQ(l, r) = min(RMQ(l, k), RMQ(r – 2^k + 1, k))

  = min(st[l][k], st[r – 2^k + 1][k])
- By taking only two values, we can get the result!



# Complexity

- Preprocess: **O(N log N)**
- Query: **O(1)**
- Time Complexity: **<O(N log N), 1>**
- Memory Complexity: **O(N log N)**
- Overall Time Complexity: **O(N log N + Q)**

# Are there still any better methods to find the LCA?

- Sad but true
- The best solution up till now has overall time complexity of **O(N)**, or **<O(N), O(1)>**
- The knowledge is rarely required in OI competitions
- This algorithm is left for interested readers with ability to dig in

# Practise Problems

Easy:

- LSCCT L12D4
- SPOJ LCA
- SPOJ QTREE2
- SPOJ DISQUERY

Medium:

- LSCCT NP159
- SPOJ Lowest Common Ancestor

Hard:

- CF 832D
- LSCCT NP165