

The background is a light cream color with various decorative elements. In the top left, there is a blue line-art floral motif. In the top right, there is a large blue abstract shape with a yellow circle and several orange dots. In the bottom left, there is an orange abstract shape with a yellow circle and several orange dots. In the bottom right, there is a pink line-art floral motif. Two dashed lines, one in the top center and one on the left side, curve across the background.

# DISJOINT SET UNION

Presented by Group 57

The background is a light cream color with various decorative elements. In the top left, there is a blue line-art floral motif. In the top right, there is a large blue abstract shape with a yellow circle and orange dots. In the bottom left, there is an orange abstract shape with a yellow circle and orange dots. In the bottom right, there is a pink line-art floral motif. Two dashed lines, one blue and one pink, curve across the top and bottom of the slide. The title 'DISJOINT SET UNION' is written in large, bold, maroon letters in the center. Below it, 'Presented by Group 57' is written in a smaller, dark maroon font.

# DISJOINT SET UNION

Presented by Group 57

# OUR TEAM



Kiet Phan  
-23APCS2-

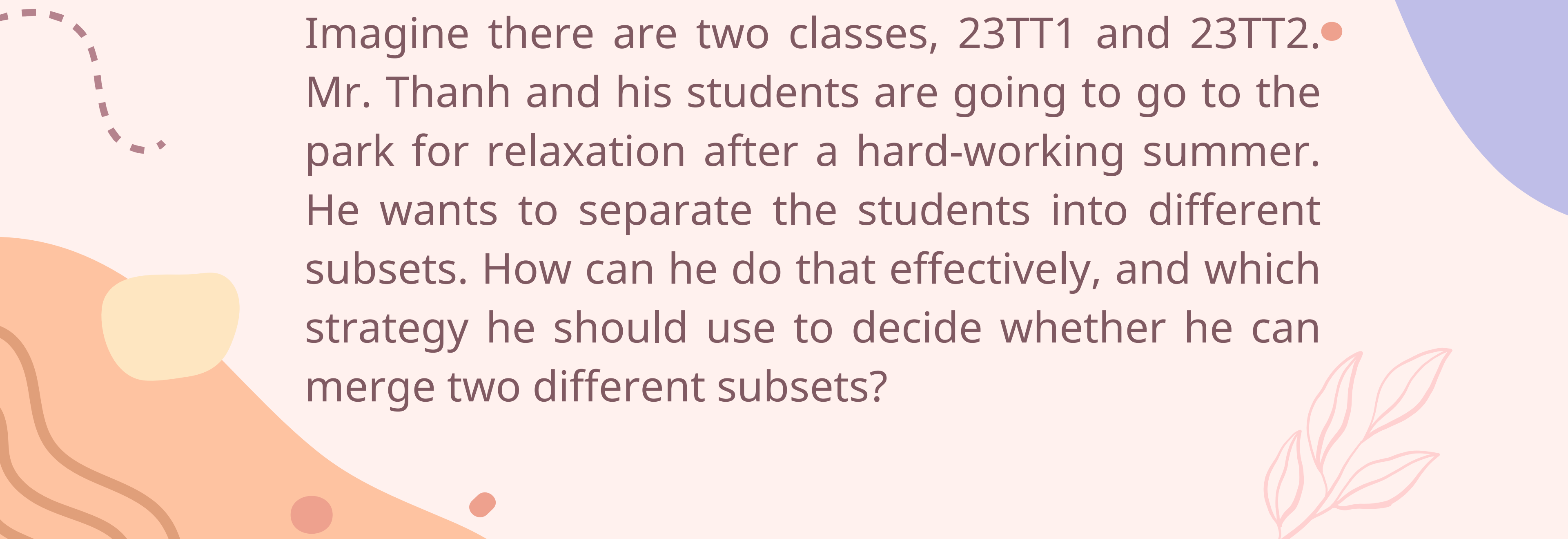


Vu Hoang  
-23APCS2-



# ABSTRACTION

(CRITICAL THINKING)



Imagine there are two classes, 23TT1 and 23TT2. Mr. Thanh and his students are going to go to the park for relaxation after a hard-working summer. He wants to separate the students into different subsets. How can he do that effectively, and which strategy he should use to decide whether he can merge two different subsets?

# 1) INTRODUCTION

In computer science, a disjoint-set data structure, also known as a union-find or merge-find set, efficiently manages a partition of a set into disjoint subsets. It supports operations for adding new sets, merging existing sets, and finding a representative member to determine if any two elements belong to the same set.





# 1.1) HISTORY

1. Disjoint-set forests were first described by Bernard A. Galler and Michael J. Fischer in 1964.
2. In 1975, Robert Tarjan first proved the  $O(m(a(n)))$  (inverse Ackermann function) upper bound on the algorithm's time complexity.
3. In 1989, Fredman and Saks showed that any disjoint-set data structure must access an optimal number of words of logarithmic bits per operation.
4. In 2007, Sylvain Conchon and Jean-Christophe Filliâtre developed a semi-persistent disjoint-set forest and formalized its correctness using the function Coq.

# 1.2) MAIN OPERATIONS

**1. Adding new sets**

**2. Merging sets (combining two sets)**

**3. Finding a representative member of a set**

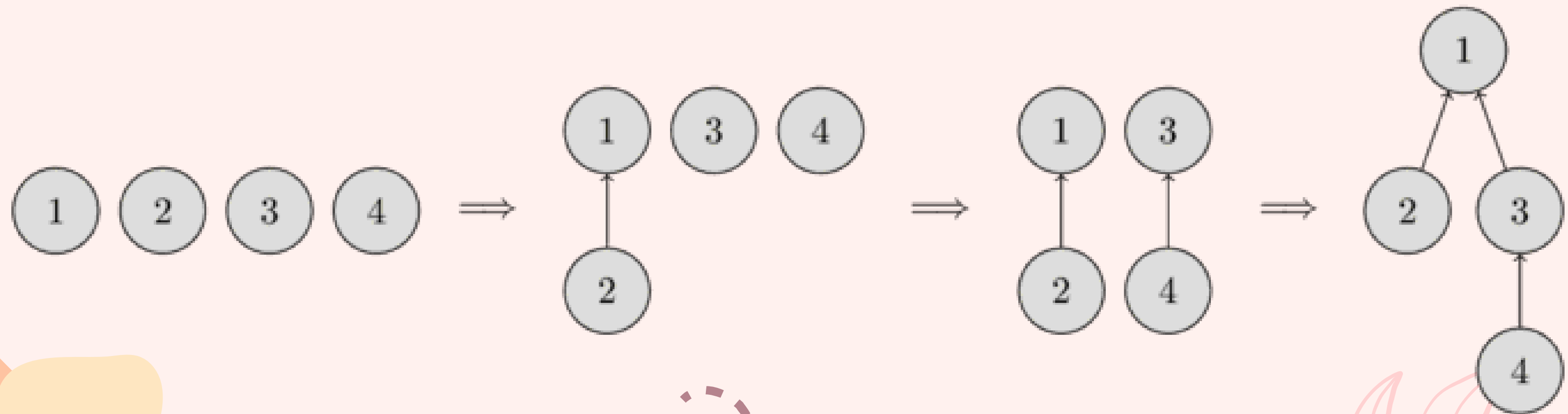
# 2) MAIN ALGORITHMS

- `make_set(v)`
- `union_sets(u, v)`
- `find_set(v)`
- path compression
- union by size / rank
- linking by index / coin - flip linking



# 2.1) INITIALIZATION

- Store the sets in form of tree
- Maintain parent that stores a reference to its immediate ancestor





## 2.2) NAIVE APPROACH

```
16  int parent[N];
17
18  void make_set(int v) {
19      parent[v] = v;
20  }
21
22  int find_set(int v) {
23      if (v == parent[v]) return v;
24      return find_set(parent[v]);
25  }
26
27  void Union(int a, int b) {
28      a = find_set(a);
29      b = find_set(b);
30      if (a != b) {
31          parent[b] = a; // or parent[a] = b;
32      }
33  }
34
```



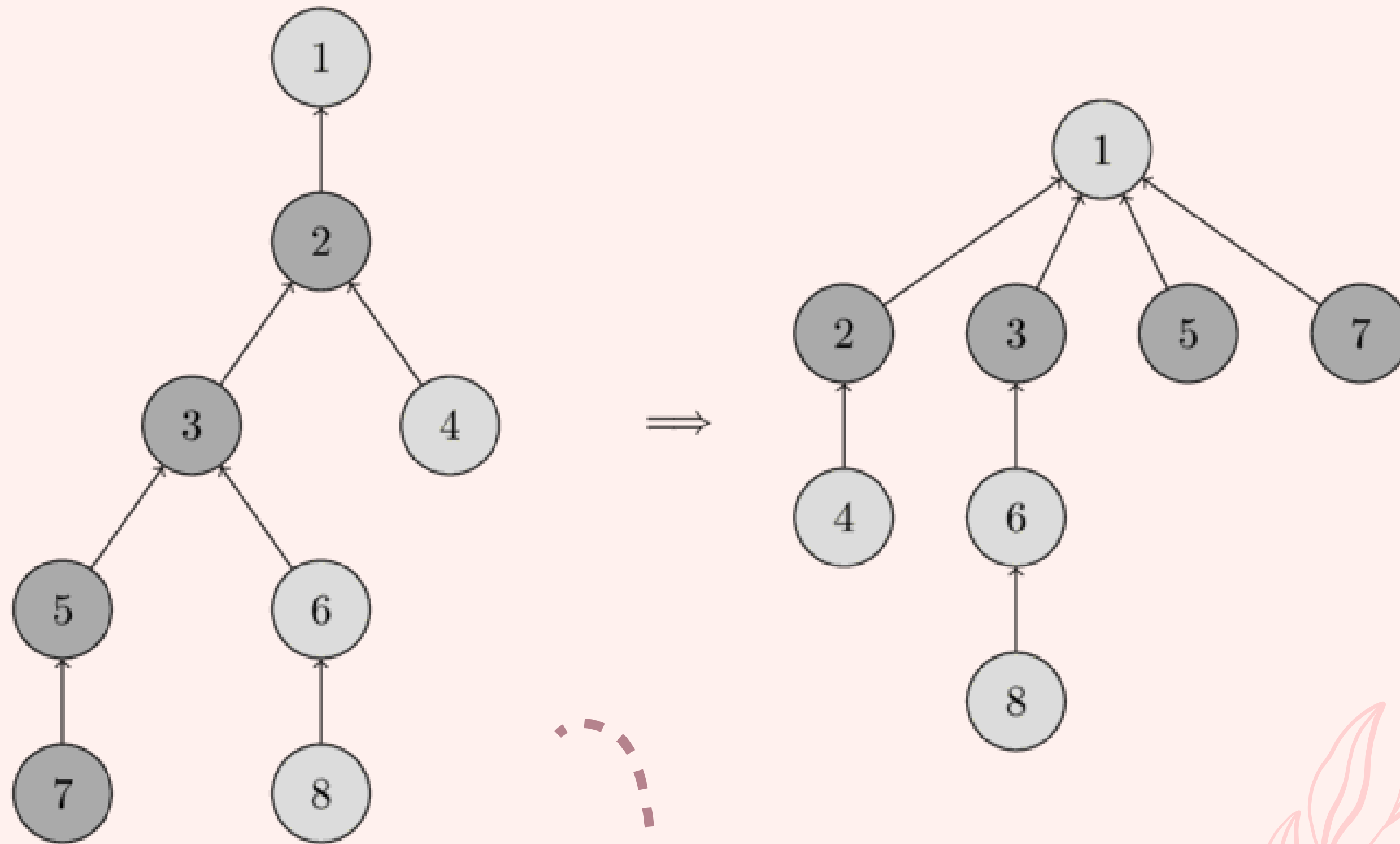
# COMMENTS

- However this implementation is inefficient. It is easy to construct an example, so that the trees degenerate into long chains. In that case each call `find_set(v)` can take  $O(n)$  time complexity.
- This is not our expectation! We want linear constant time! Therefore we will consider two optimizations that will allow to significantly accelerate the work.

## 2.3) PATH COMPRESSION

- This is the optimization for speeding up the function `find_set(v)`
- When we call `find_set(v)` for a vertex  $v$ , we are essentially finding the representative  $p$  of the set containing  $v$ !
- The trick is to make the paths for all those nodes shorter, by setting the parent of each visited vertex directly to  $p$ .
- This simple modification of the operation already achieves the time complexity  $O(\log(n))$  on average!

# VISUALIZATION



# CODE

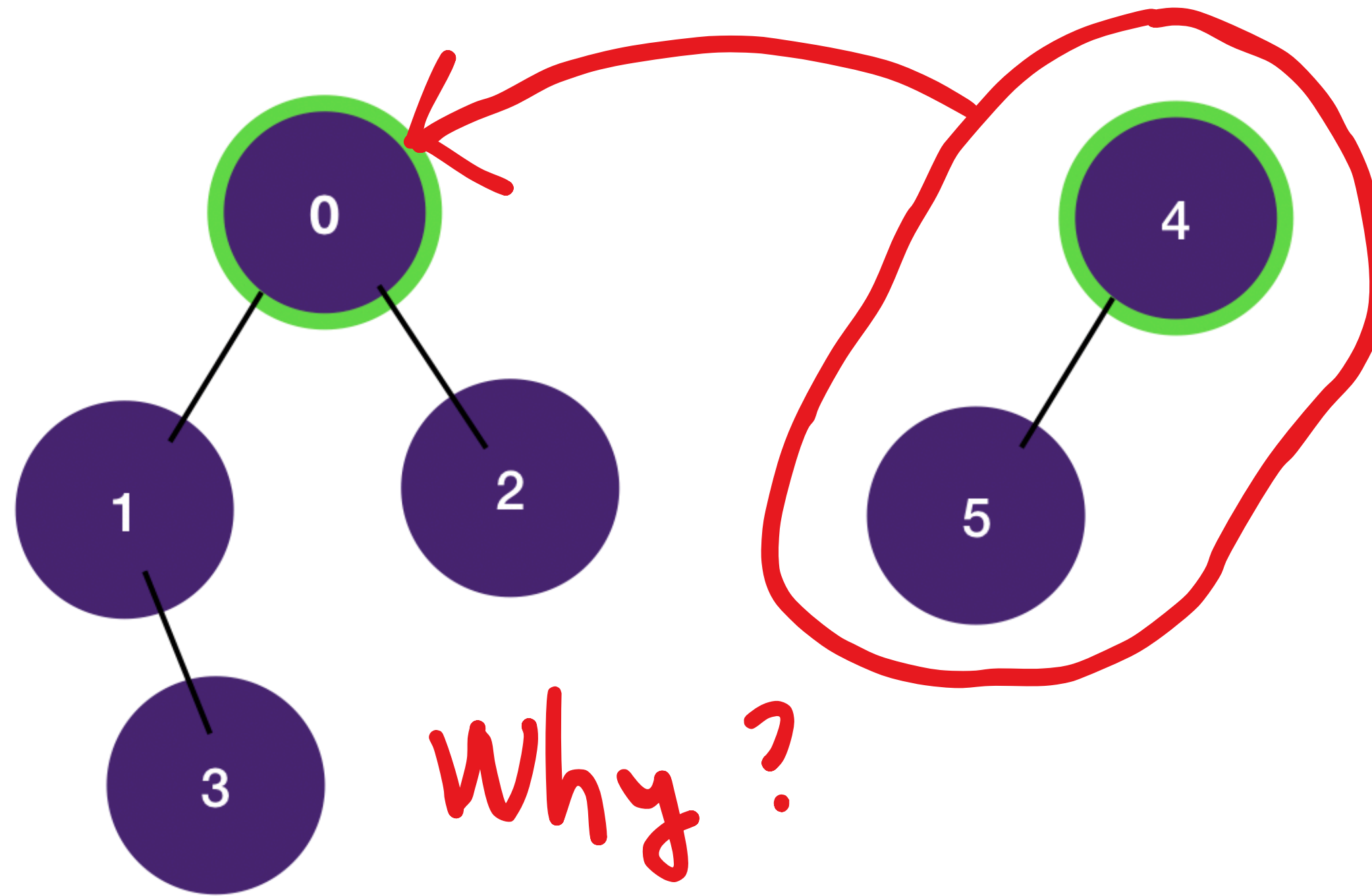
```
27
28 int find_set(int v) {
29     if (v == parent[v]) return v;
30     return parent[v] = find_set(parent[v]);
31 }
32
33
34 // or shorter
35
36
37 int find_set(int v) {
38     return v == parent[v] ? v : parent[v] = find_set(parent[v]);
39 }
40
```

## 2.4) UNION BY SIZE/RANK •

- This is the optimization for speeding up the function  $\text{Union}(a,b)$ . To be precise, we will change which tree gets attached to the other one.
- In the naive implementation, the second tree always attaches to the first, which can create long chains of length  $O(n)$ . This optimization prevents that by carefully choosing which tree to attach.
- There are two heuristic approaches for this: union by size and union by rank. However, in both methods, we will attach the lower one to the bigger one! Think why?



# VISUALIZATION



# UNION BY SIZE

```
41 void make_set(int v) {  
42     parent[v] = v;  
43     size[v] = 1;  
44 }  
45  
46 void union_sets(int a, int b) {  
47     a = find_set(a);  
48     b = find_set(b);  
49     if (a != b) {  
50         if (size[a] < size[b])  
51             swap(a, b);  
52         parent[b] = a;  
53         size[a] += size[b];  
54     }  
55 }
```

# UNION BY RANK

```
41 void make_set(int v) {
42     parent[v] = v;
43     rank[v] = 0;
44 }
45
46 void union_sets(int a, int b) {
47     a = find_set(a);
48     b = find_set(b);
49     if (a != b) {
50         if (rank[a] < rank[b])
51             swap(a, b);
52         parent[b] = a;
53         if (rank[a] == rank[b])
54             rank[a]++;
55     }
56 }
57
```

## 2.5) COIN-FLIP LINKING

- Both union by rank and union by size need extra data for each set and must update these values with each union. An alternative is a simpler randomized algorithm called linking by index.
- We assign a random index to each set and attach the set with the smaller index to the one with the larger index. This approach, similar to union by size, has the same time complexity but is slightly slower in practice.

# CODE

```
41 void make_set(int v) {  
42     parent[v] = v;  
43     index[v] = rand();  
44 }  
45  
46 void union_sets(int a, int b) {  
47     a = find_set(a);  
48     b = find_set(b);  
49     if (a != b) {  
50         if (index[a] < index[b])  
51             swap(a, b);  
52         parent[b] = a;  
53     }  
54 }  
55
```

# MISCONCEPTION

- It's a common misconception that just flipping a coin, to decide which set we attach to the other, has the same complexity. However that's not true.
- People has proved that coin-flip linking combined with path compression has complexity  $\Omega\left(n \frac{\log n}{\log \log n}\right)$ .
- In benchmark, it performs a lot worse than union by size/rank or linking by index.



# 3) TIME COMPLEXITY

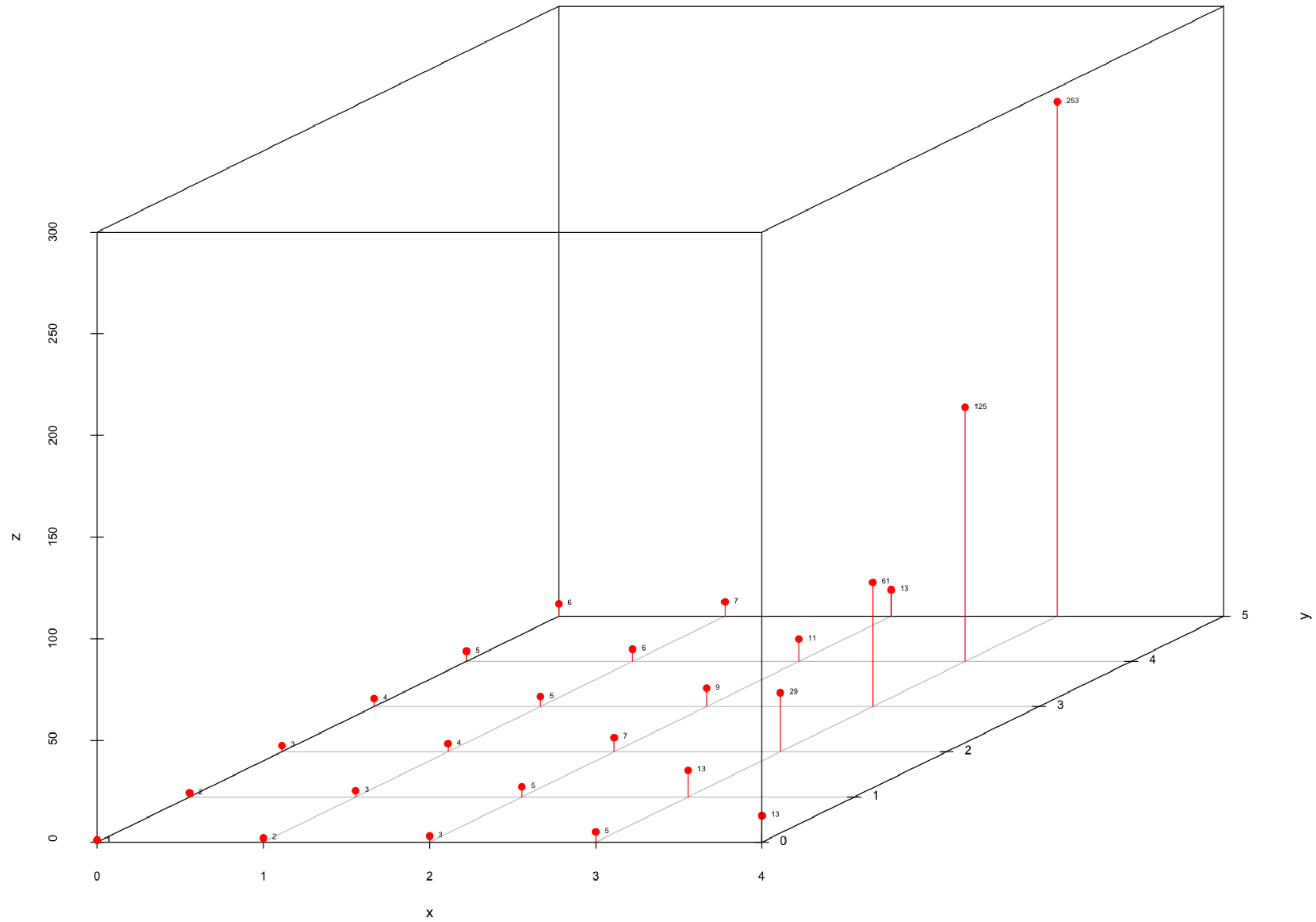
- Combining path compression with union by size or rank results in nearly constant time queries. The final amortized time complexity is  $O(\alpha(n))$ , where  $\alpha(n)$  is the inverse Ackermann function, which grows very slowly. For all practical purposes, it doesn't exceed 4 for any reasonable  $n$  (approximately  $n < 10^{600}$ ).
- Additionally, DSU with union by size or rank but without path compression operates in  $O(\log n)$  time per query.

# INVERSE ACKERMANN FUNCTION

- An algorithm with  $O(\alpha(n))$  time complexity, where  $\alpha(n)$  is the inverse Ackermann function, is an algorithm that has a time complexity that is bounded by the inverse Ackermann function. This means that the running time of the algorithm is always less than or equal to the value of the inverse Ackermann function for the input size  $n$ .

$$A(m, n) = \begin{cases} n + 1 & \text{if } m = 0, \\ A(m - 1, 1) & \text{if } m > 0 \text{ and } n = 0, \\ A(m - 1, A(m, n - 1)) & \text{if } m > 0 \text{ and } n > 0. \end{cases}$$

# Ackermann Function



# 4) APPLICATIONS

- Connected components in a graph - Kruskal's algorithm
- Search for connected components in an image
- Store additional information for each set
- Compress jumps along a segment
- Support distances up to representative
- Support the parity of the path length
- Offline RMQ (range minimum query)
- Offline LCA (lowest common ancestor in a tree)
- Storing the DSU explicitly in a set list
- Storing the DSU by maintaining a clear tree structure

# 4.1) CONNECTED COMPONENTS

- This is one of the obvious applications of DSU.
- The problem is defined as follows: Starting with an empty graph, we add vertices and undirected edges, then answer queries asking if two vertices are in the same connected component.
- We can use a DSU to handle vertex and edge additions and queries in nearly constant average time.
- This approach is crucial because a similar problem occurs in Kruskal's algorithm for finding a minimum spanning tree. By using DSU, we can improve the complexity from  $O(m \log n + n^2)$  to  $O(m \log n)$ .

## 4.2) SEARCH IMAGES

- There is an image of  $n \times m$  pixels. Originally all are white, but then a few black pixels are drawn. You want to determine the size of each white connected component in the final image.
- To solve this, we iterate over all white pixels in the image. For each cell, we check its four neighbors, and if a neighbor is white, we call union sets. This creates a DSU with nodes corresponding to image pixels, where the resulting trees are the desired connected components.



## 4.3) STORE INFORMATION

- DSU lets you store extra information in sets. For example, you can store the size of sets at the representative nodes, as described in the Union by size section. Similarly, you can store any other information about the sets at these nodes:
  - Sum of Elements
  - Minimum or Maximum Value
  - Rank
  - Metadata

## 4.4) COMPRESS JUMPS

- An example of this is the problem of Painting subarrays offline.
- We have a segment of length  $L$ , where each element initially has the color zero. For each query  $(l, r, c)$ , we need to repaint the subarray from index  $l$  to  $r$  with the color  $c$ . At the end, we want to find the final color of each cell. We assume that all queries are known in advance, meaning the task is offline.

## 4.4.1) PAINTING SUBARRAYS

- What are your ideas? Of course, we have to do the painting in reverse as follow the initial traversal would be inefficient.
- We first make the parent of a cell the next unpainted cell. While traversing in reverse order, we will find the left-most unpainted cell and color it, make the pointer of it to the right then move to its parent.
- We will use path compression in DSU, therefore, each query will take  $O(\log(n))$ .

## 4.4.1) PAINTING SUBARRAYS

```
for (int i = 0; i <= L; i++) {
    make_set(i);
}

for (int i = m-1; i >= 0; i--) {
    int l = query[i].l;
    int r = query[i].r;
    int c = query[i].c;
    for (int v = find_set(l); v <= r; v = find_set(v)) {
        answer[v] = c;
        parent[v] = v + 1;
    }
}
```

# 4.5) SUPPORT DISTANCES

- We can use DSU to count the total distances of a vertex to its representation (i.e. the path length in the tree from the current node to the root of the tree).
- If we don't use path compression, the distance is just the number of recursive calls. But this will be inefficient.
- However it is possible to do path compression, if we store the distance to the parent as additional information for each node.

```
16 void make_set(int v) {
17     parent[v] = make_pair(v, 0);
18     rank[v] = 0;
19 }
20
21 pair<int, int> find_set(int v) {
22     if (v != parent[v].first) {
23         int len = parent[v].second;
24         parent[v] = find_set(parent[v].first);
25         parent[v].second += len;
26     }
27     return parent[v];
28 }
29
30 void union_sets(int a, int b) {
31     a = find_set(a).first;
32     b = find_set(b).first;
33     if (a != b) {
34         if (rank[a] < rank[b])
35             swap(a, b);
36         parent[b] = make_pair(a, 1);
37         if (rank[a] == rank[b])
38             rank[a]++;
39     }
40 }
41
```



## 4.6) OFFLINE RMQ (RANGE MINIMUM QUERY)

- We are given an array  $a[]$  and we have to compute some minima in given segments of the array.
- To solve this with DSU, iterate over the array and for each  $i$ th element, answer all queries  $(L, R)$  where  $R$  equals  $i$ . Maintain a DSU for the first  $i$  elements where each element's parent is the next smaller element to its right. The answer to a query is then  $a[\text{find\_set}(L)]$ , the smallest number to the right of  $L$ .

```
16 struct Query {
17     int L, R, idx;
18 };
19
20 vector<int> answer;
21 vector<vector<Query>> container;
22
23 // Container[i] will contain all queries with R == i
24
25 int main() {
26
27     stack<int> s;
28     for (int i = 0; i < n; i++) {
29         while (!s.empty() && a[s.top()] > a[i]) {
30             parent[s.top()] = i;
31             s.pop();
32         }
33         s.push(i);
34         for (Query q : container[i]) {
35             answer[q.idx] = a[find_set(q.L)];
36         }
37     }
38     return 0;
39 }
40 }
```

## 4.7) STORING THE DSU IN A SET LIST

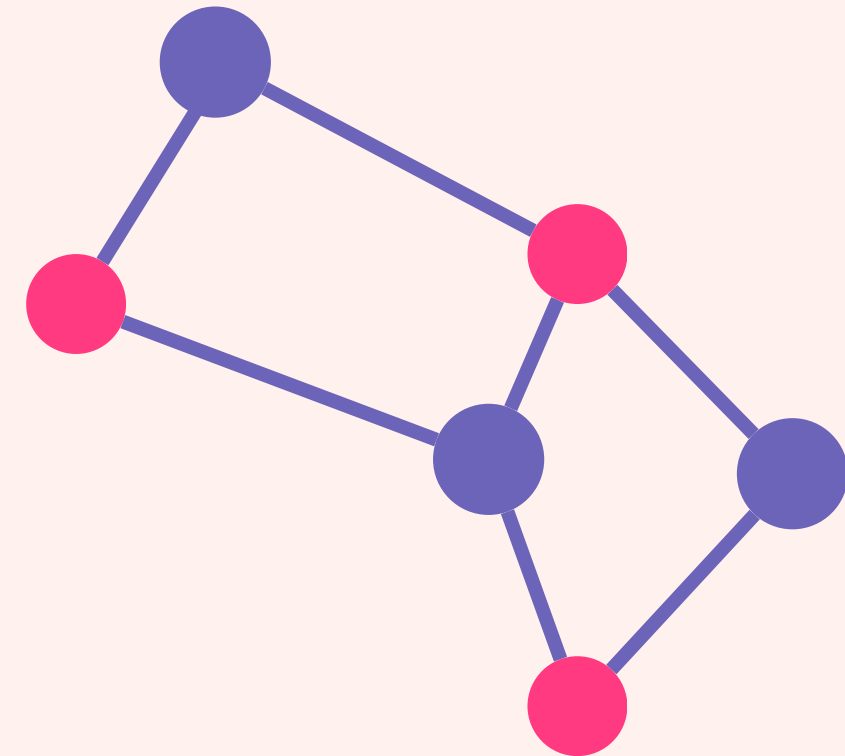
- One alternative way to store a DSU is to explicitly store each set as a list of its elements, with each element also referencing its set's representative.
- At first, this seems inefficient: merging two sets requires appending one list to another and updating the representative for each element in one list.
- However, using a weighting heuristic (similar to Union by size) can reduce the complexity to  $O(m + n \log n)$  for  $m$  queries on  $n$  elements.

## 4.8) STORING THE DSU IN A CLEAR TREE ●

- One powerful application of DSU (Disjoint Set Union) is its ability to store trees in both compressed and uncompressed forms. The compressed form is useful for merging trees and checking if two vertices are in the same tree. The uncompressed form is useful for tasks like finding paths between two vertices or other tree traversals.

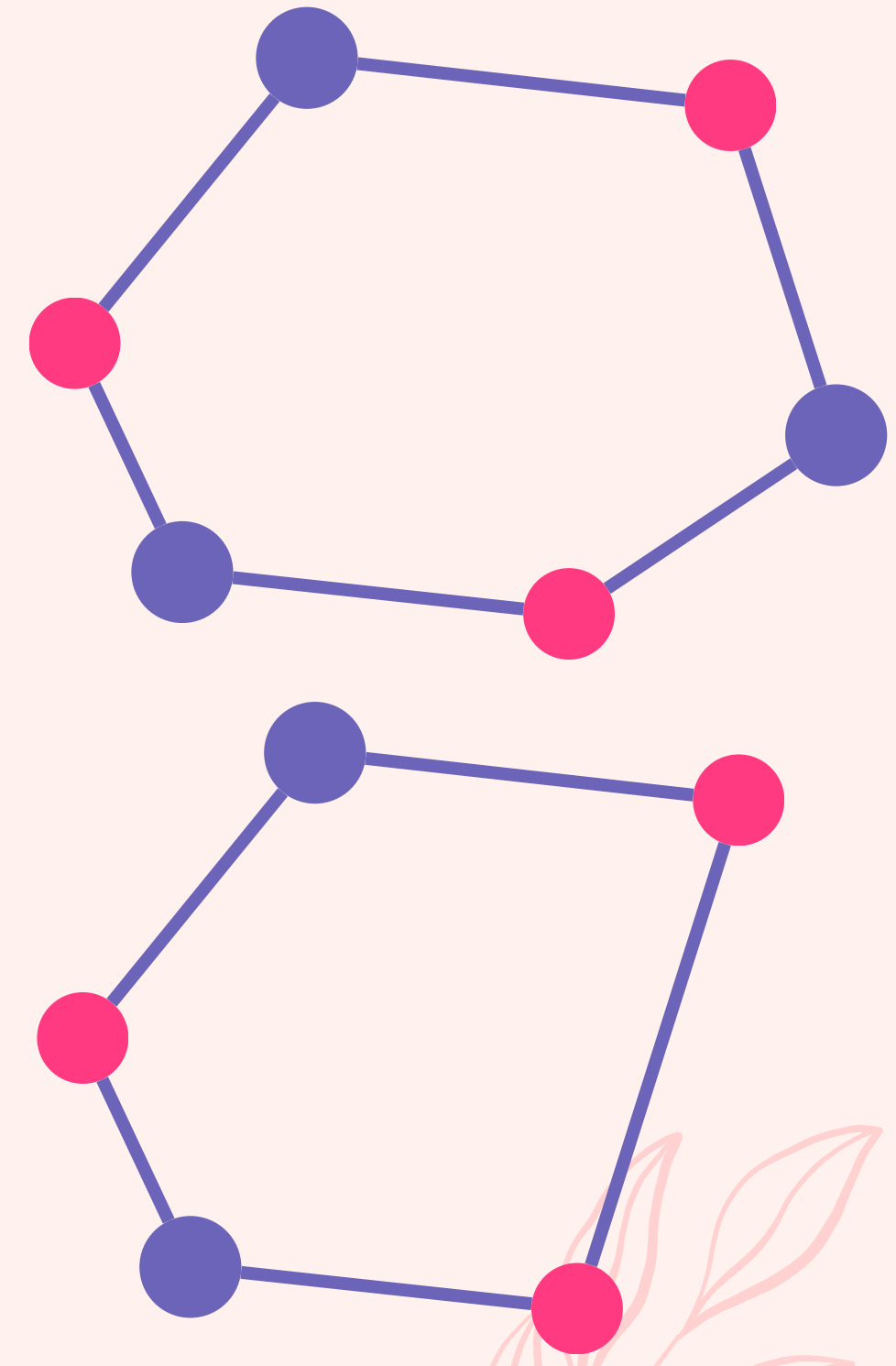
# 4.9.1) SUPPORT THE PARITY OF THE PATH LENGTH / CHECKING BIPARTITENESS ONLINE

- For example, you receive a problem. You are asked to connect two vertices  $u$  and  $v$  together, and then asked whether the connected component containing these two vertices can be colored with two different colors such that any two vertices connected by an edge always have different colors.



# 4.9.1) SUPPORT THE PARITY OF THE PATH LENGTH / CHECKING BIPARTITENESS ONLINE

- To solve this problem, I will manage the parity of the path lengths between any pair of vertices. This is because when the vertices form an odd cycle, it is always impossible to color them properly

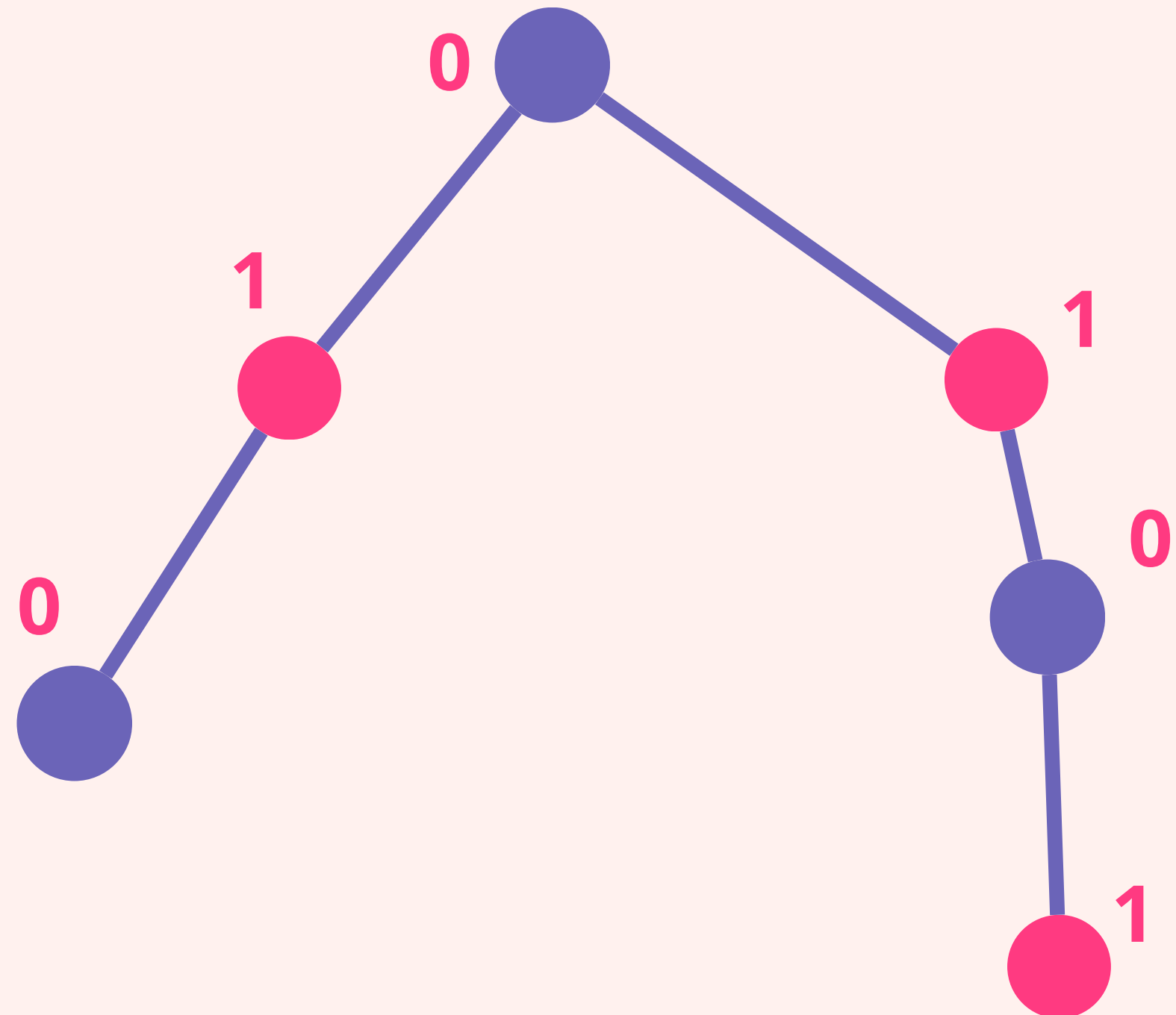


## 4.9.2) SUPPORT THE PARITY OF THE PATH LENGTH / CHECKING BIPARTITENESS ONLINE

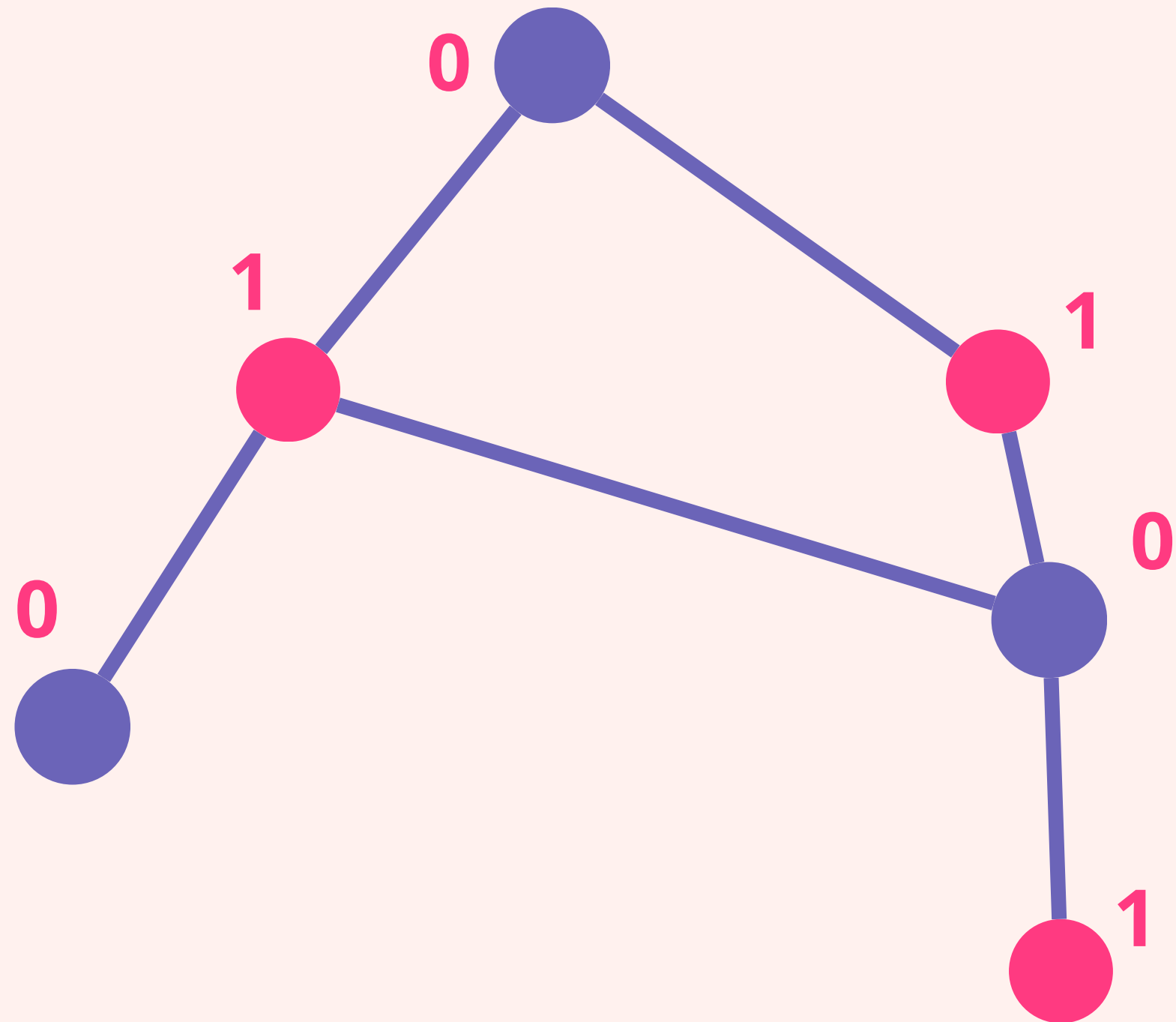
- I will apply DSU to this problem by maintaining an additional variable for each vertex to indicate whether the path from that vertex to the representative vertex is even or odd.
- Then, I can determine the parity of the path between any two vertices (imagine it like the paths from the two vertices going up to the LCA of the tree).
- Follow the code to understand better



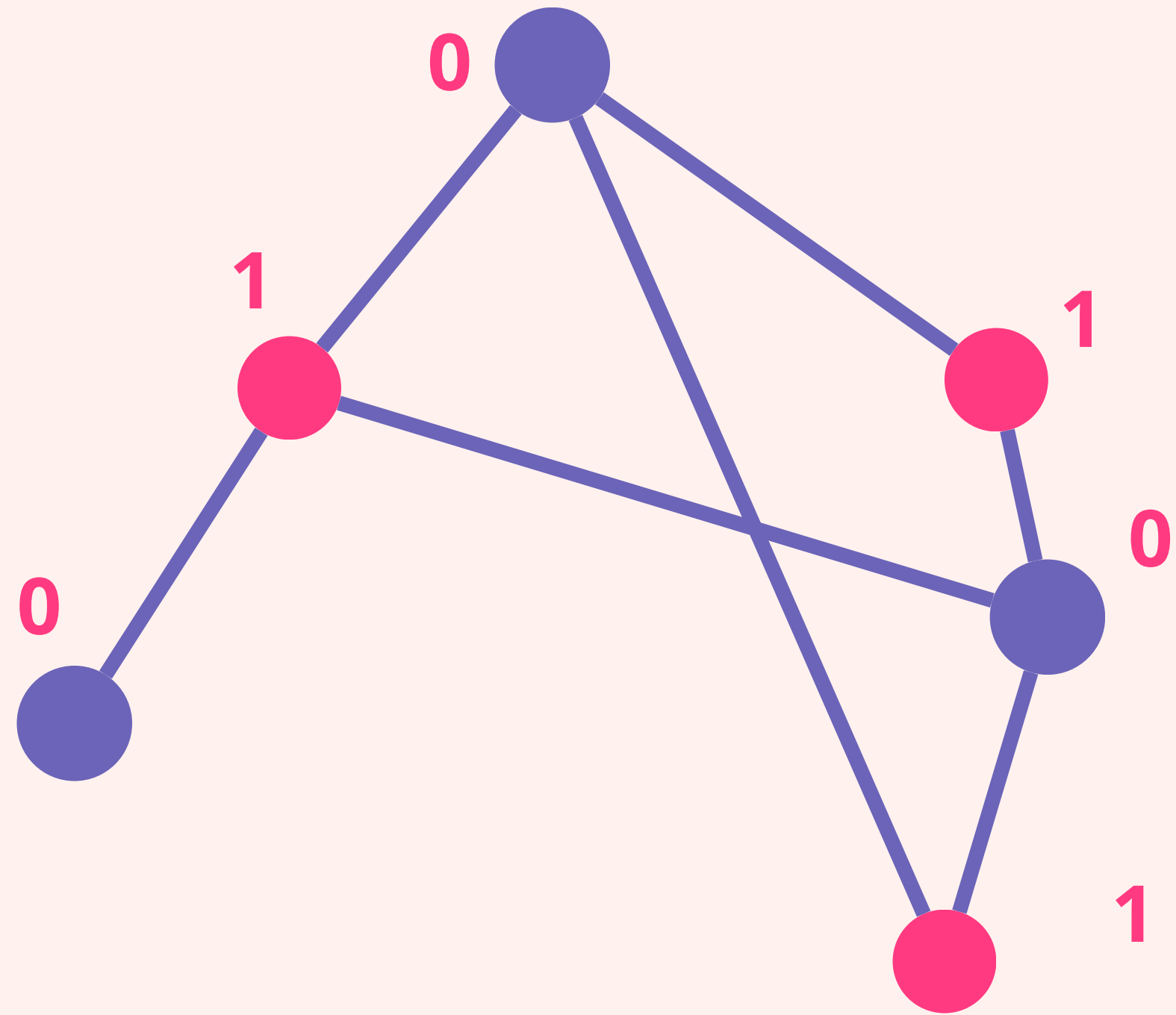
Root



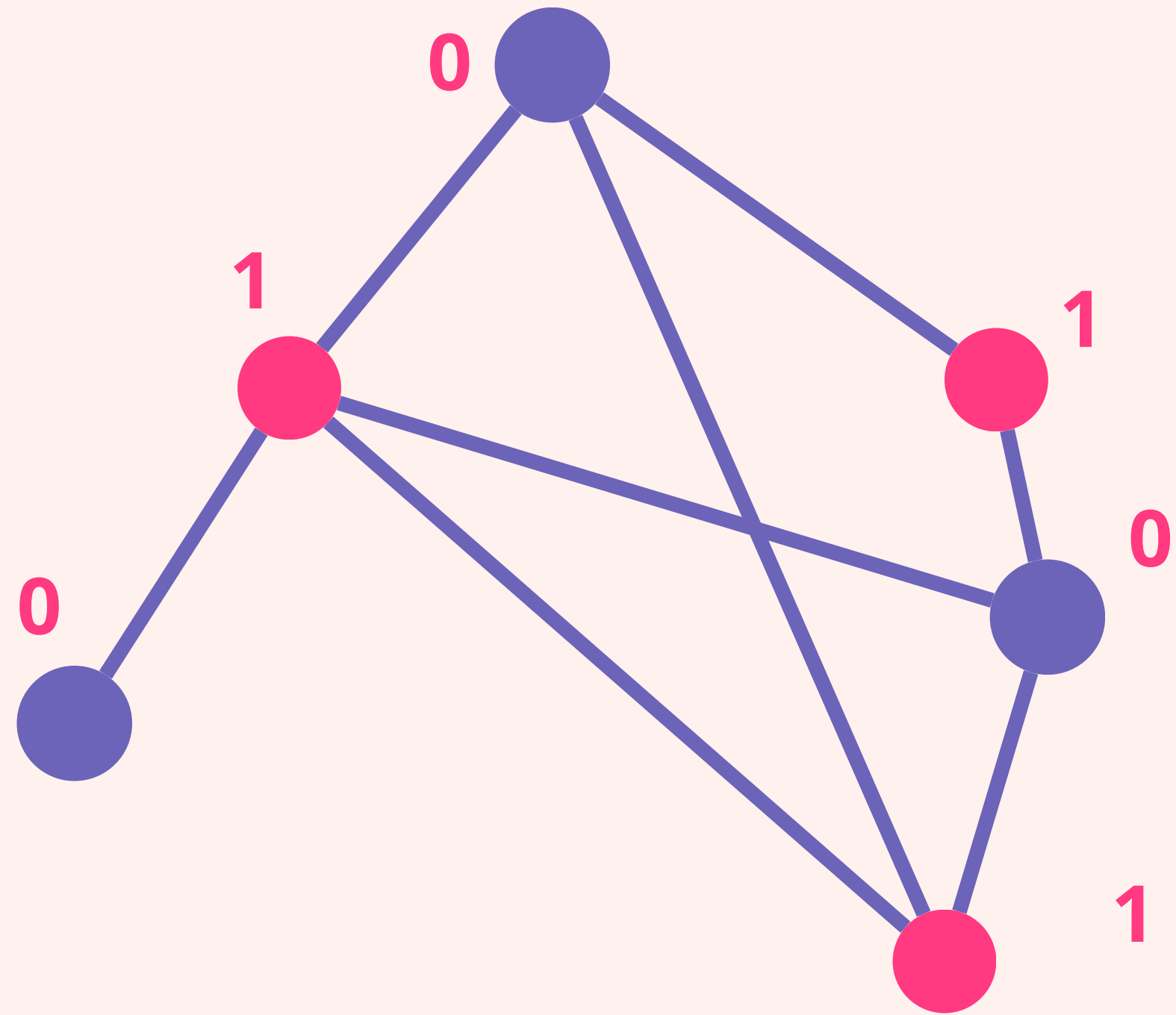
Root



Root



Root



## 4.9.3) CODE

```
void make_set(int v) {
    parent[v] = make_pair(v, 0);
    rank[v] = 0;
    bipartite[v] = true;
}

pair<int, int> find_set(int v) {
    if (v != parent[v].first) {
        int parity = parent[v].second;
        parent[v] = find_set(parent[v].first);
        parent[v].second ^= parity;
    }
    return parent[v];
}
```

```
void add_edge(int a, int b) {
    pair<int, int> pa = find_set(a);
    a = pa.first;
    int x = pa.second;

    pair<int, int> pb = find_set(b);
    b = pb.first;
    int y = pb.second;

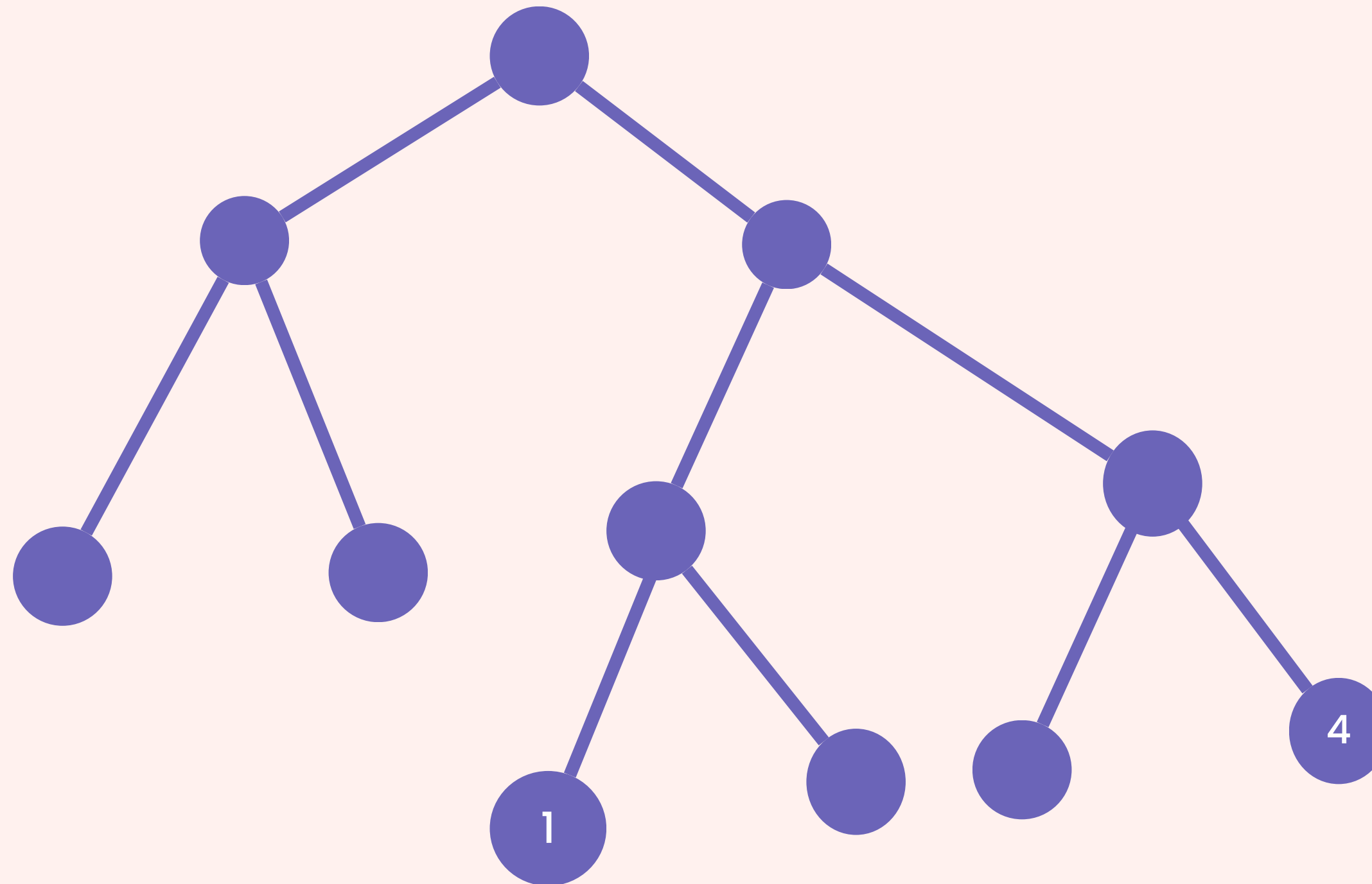
    if (a == b) {
        if (x == y)
            bipartite[a] = false;
    } else {
        if (rank[a] < rank[b])
            swap(a, b);
        parent[b] = make_pair(a, x^y^1);
        bipartite[a] ^= bipartite[b];
        if (rank[a] == rank[b])
            ++rank[a];
    }
}

bool is_bipartite(int v) {
    return bipartite[find_set(v).first];
}
```

## 4.10.1) OFFLINE LCA

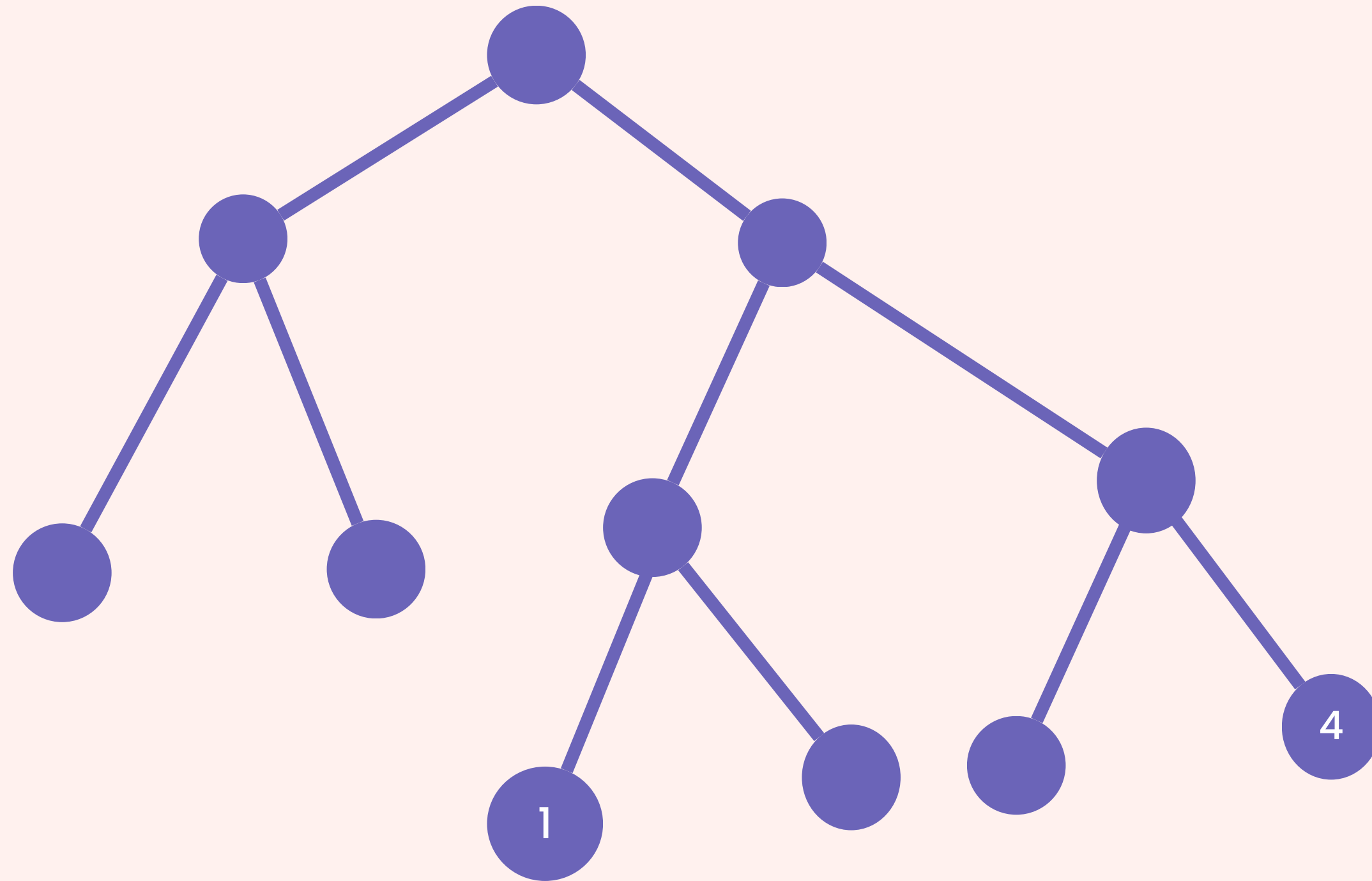
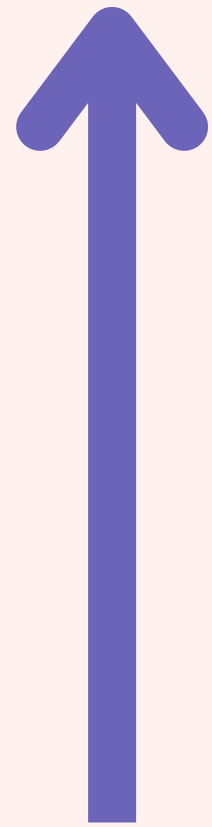
- You need to answer queries about finding the LCA of any two vertices  $u$  and  $v$  regardless of the order of the questions. This is the optimal solution for you as the complexity for each query approaches  $O(1)$ .
- To solve this problem, for the representative vertices, maintain an **ancestor** variable to know the vertex with the lowest rank that the same connected component with it when performing DSU.
- I traverse using DFS according to Tarjan's algorithm. When visiting a vertex, mark it as visited and **union\_sets** itself with its child vertices.
- Finally, check if the queries contain it. If the other vertex has been visited, then the LCA is **ancestor[find\_set[v]]**

Root

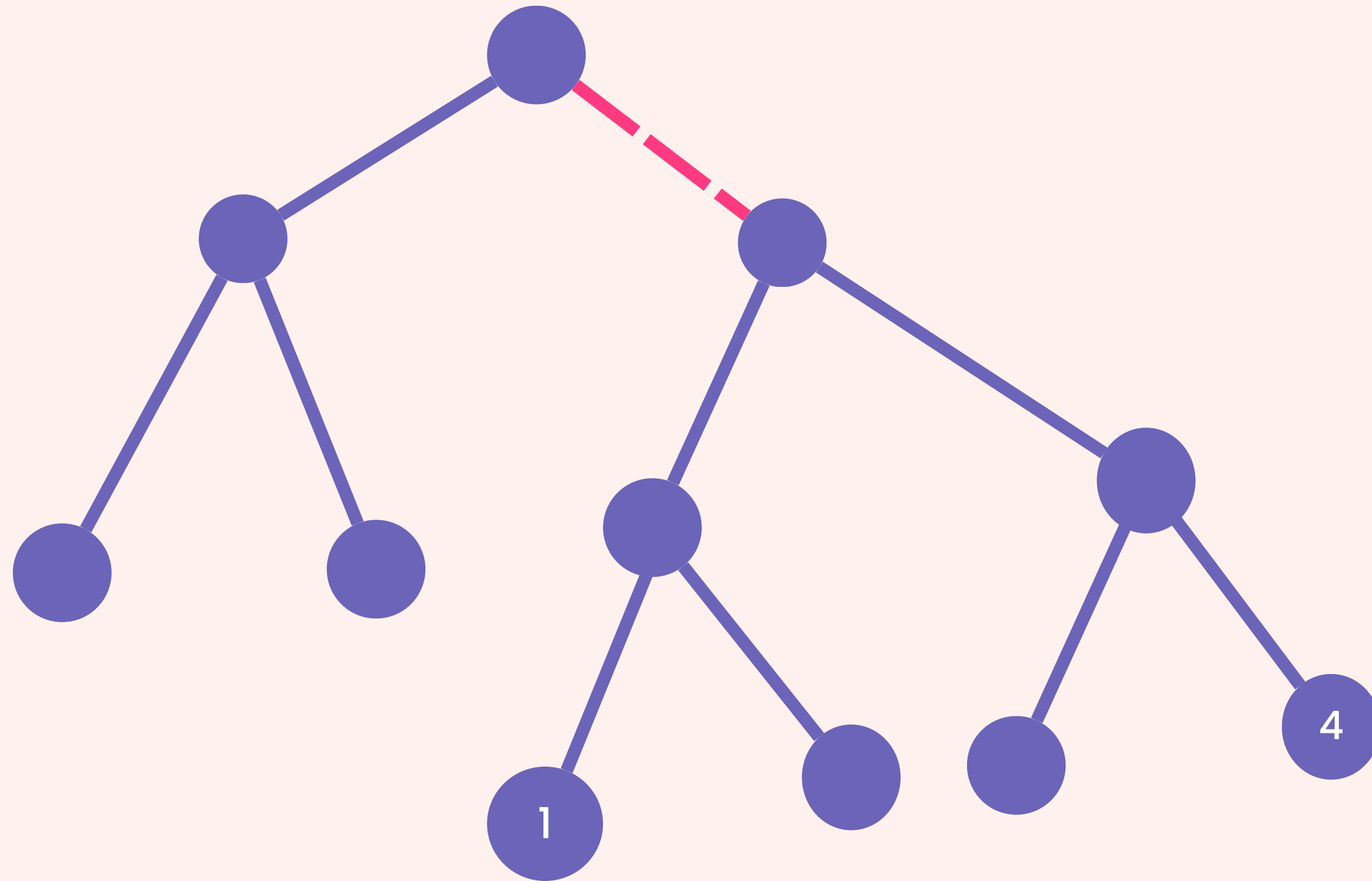
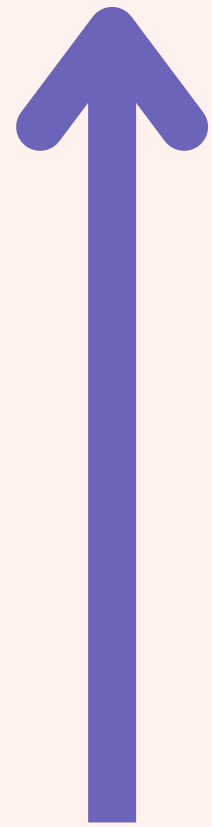




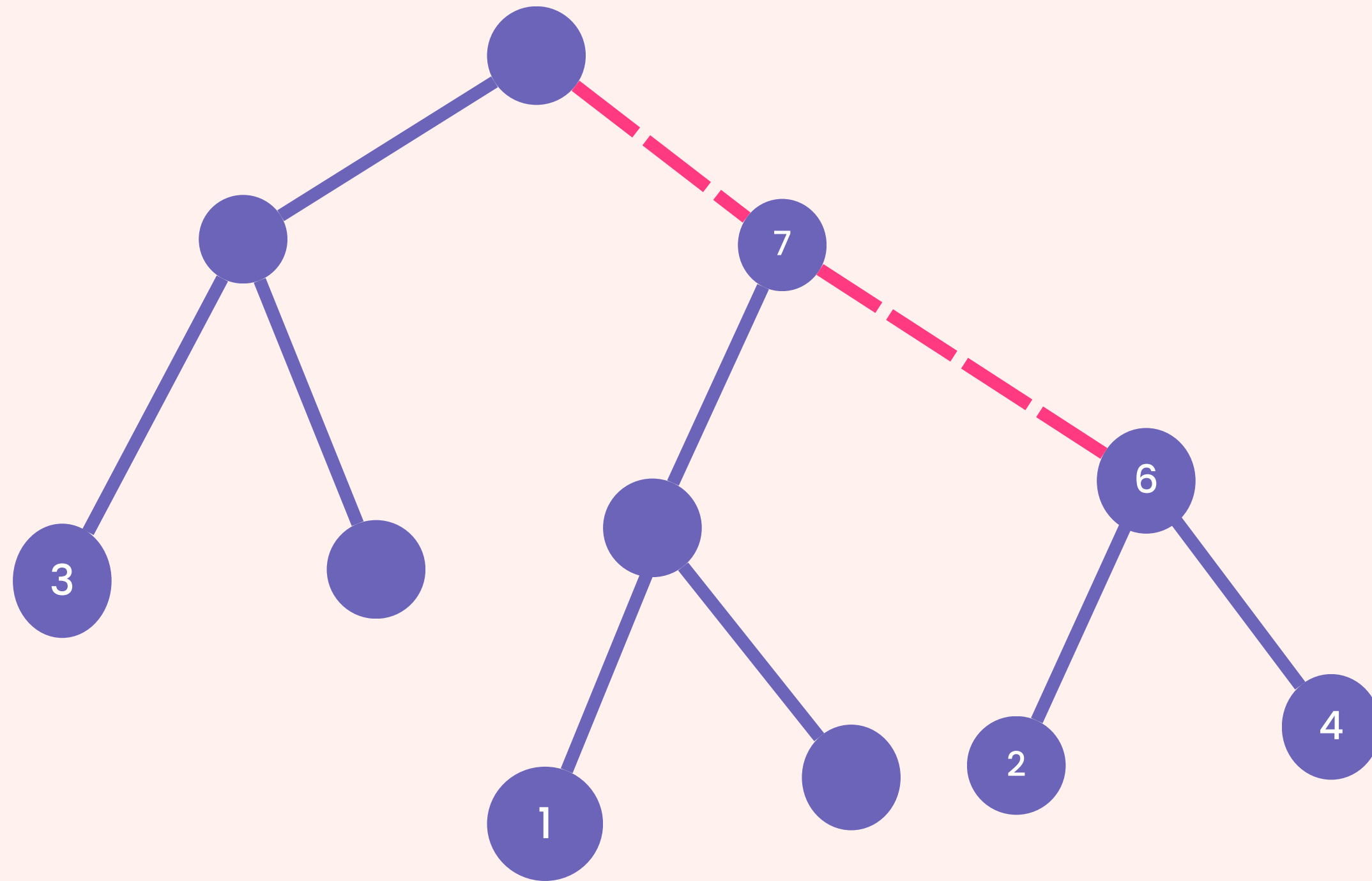
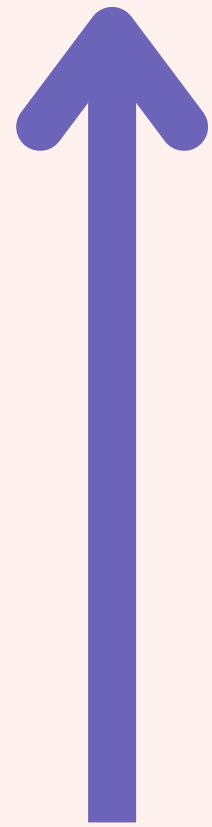
Root



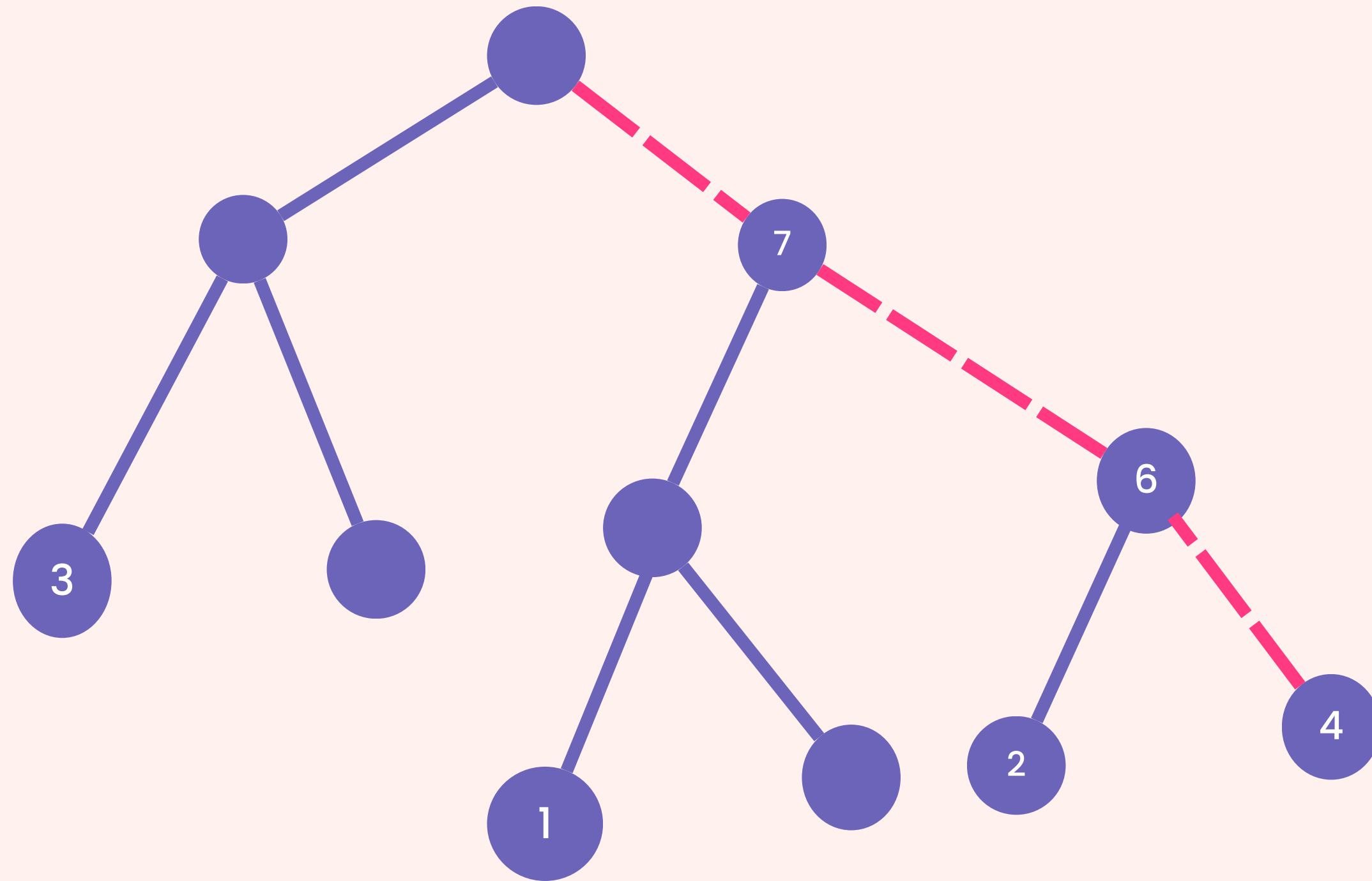
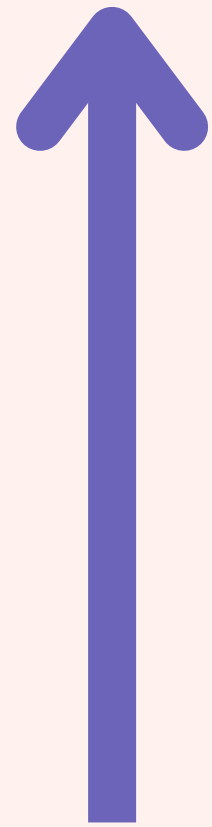
Root



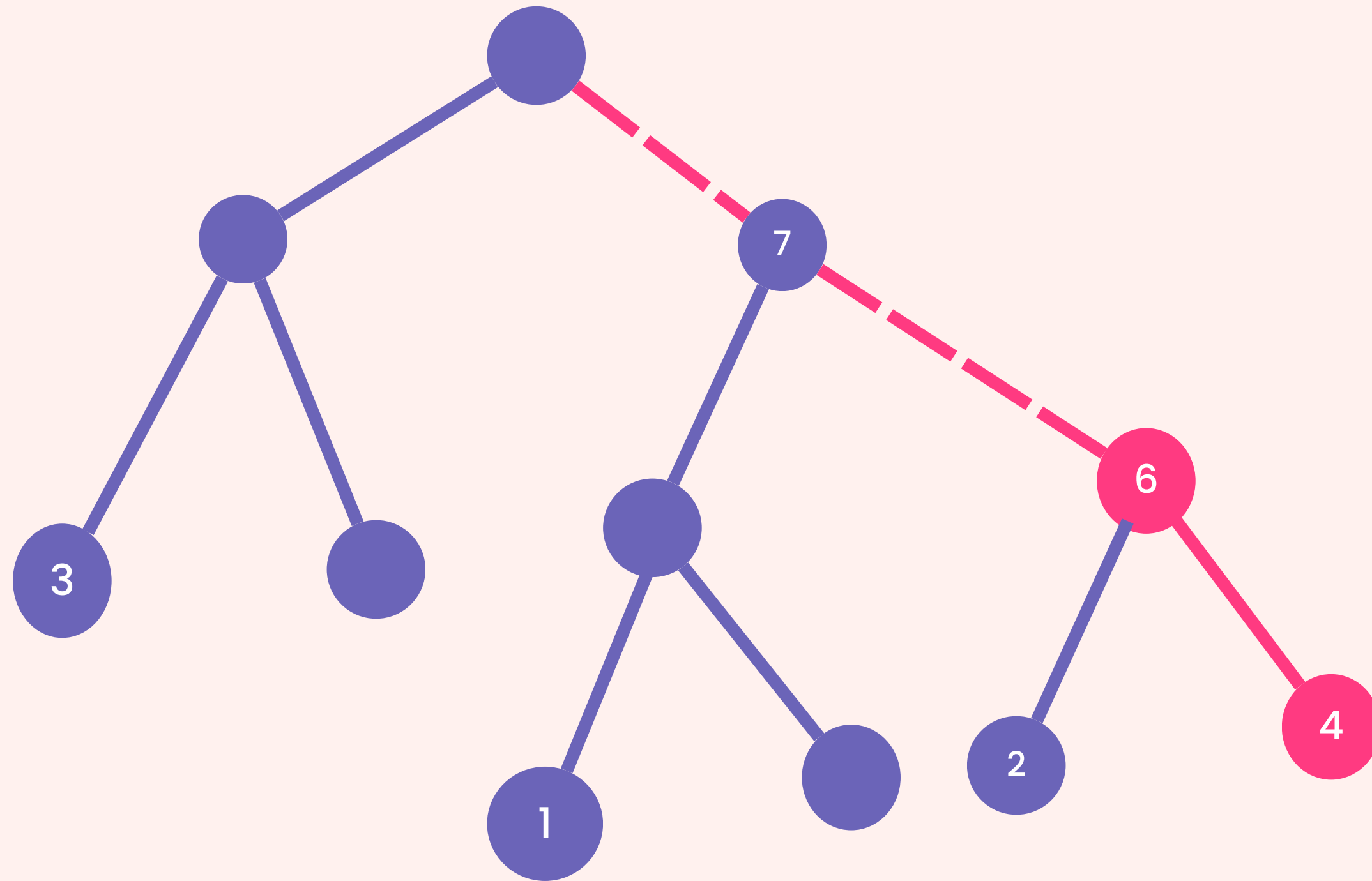
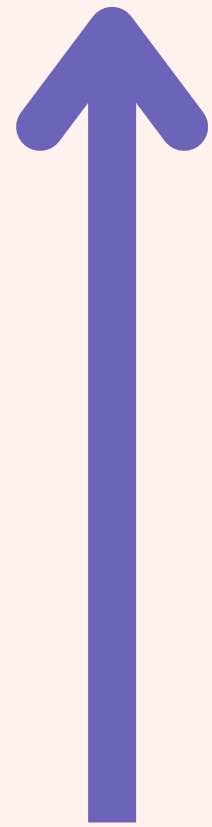
Root



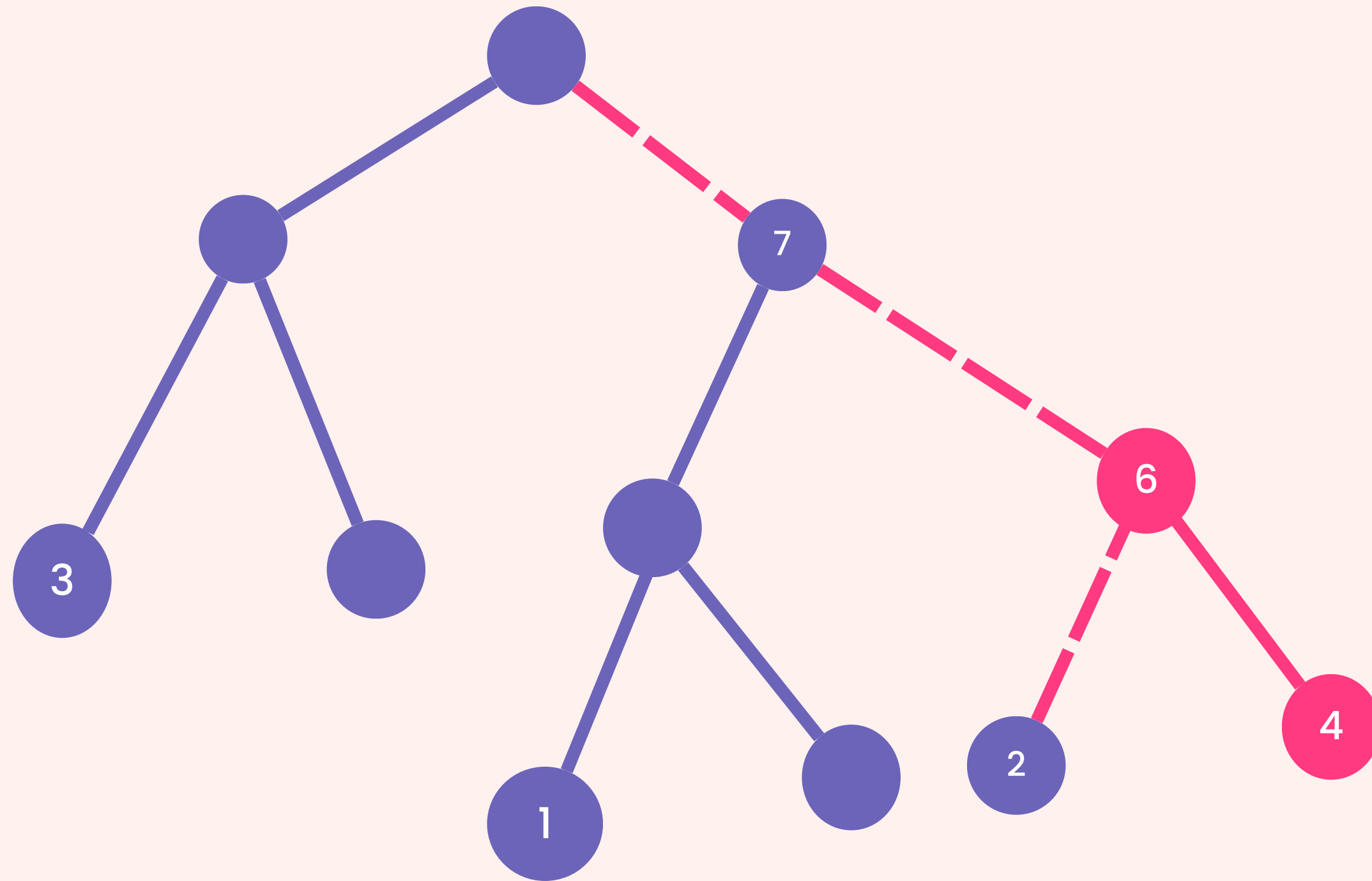
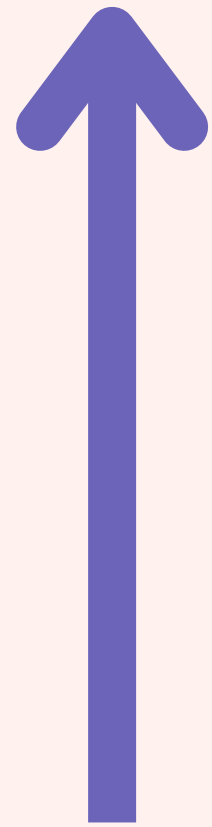
Root



Root

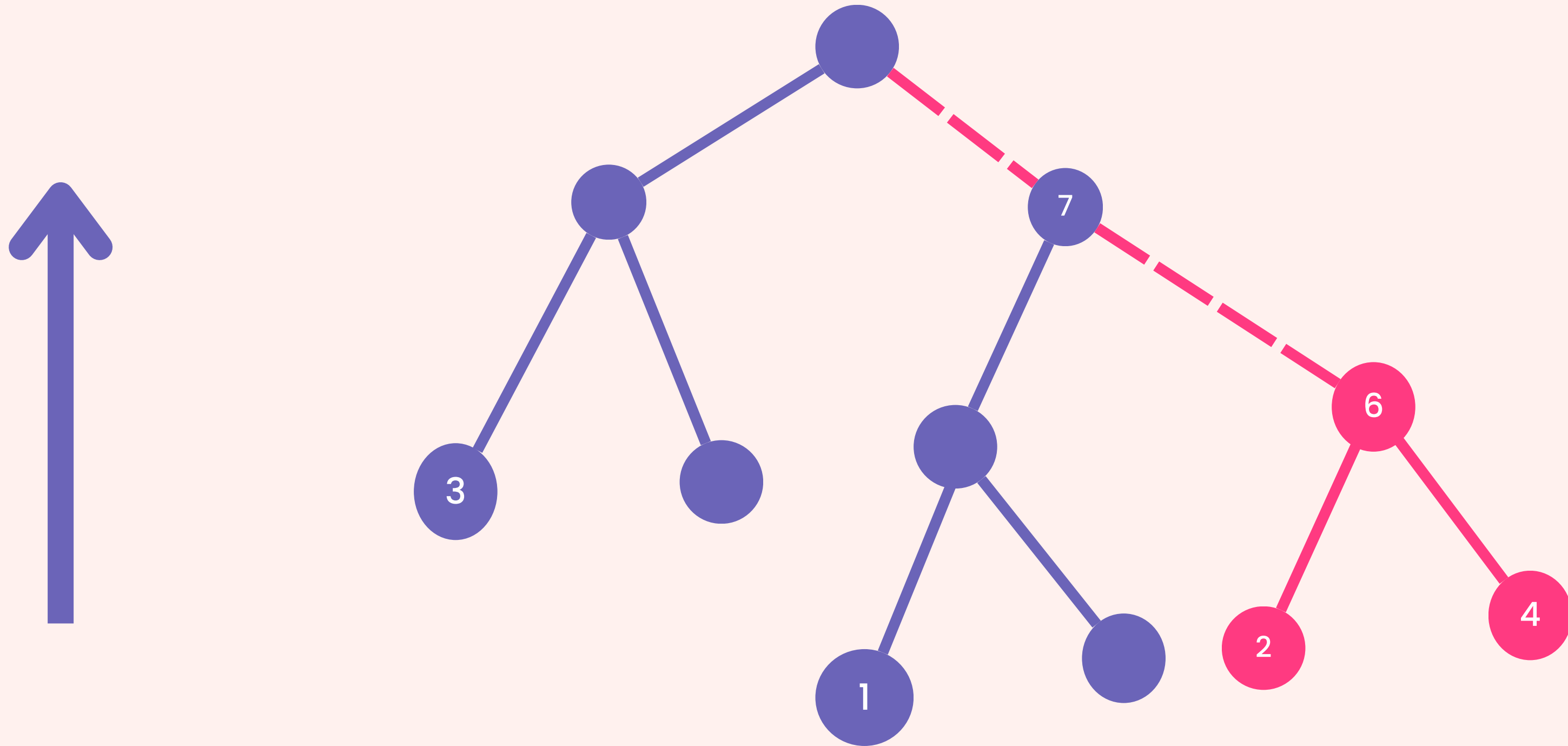


Root

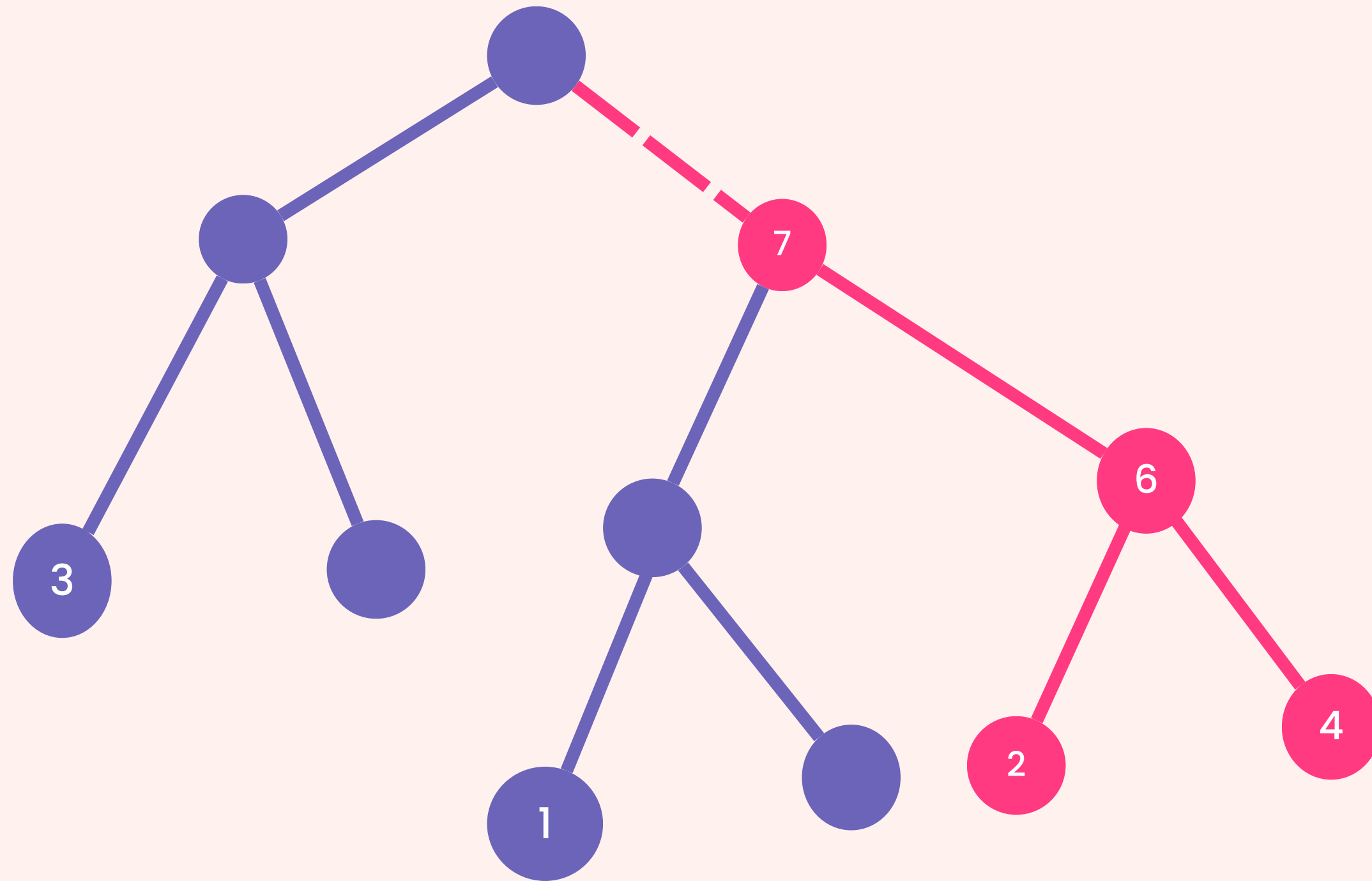
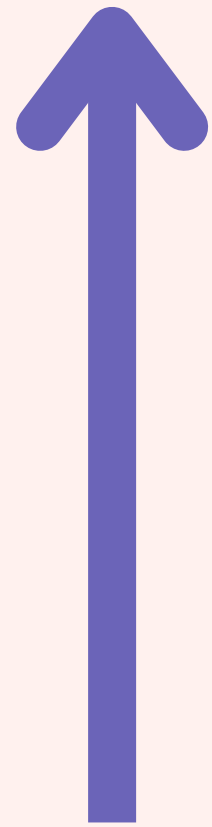


Query: 2 4, 2 6

Root

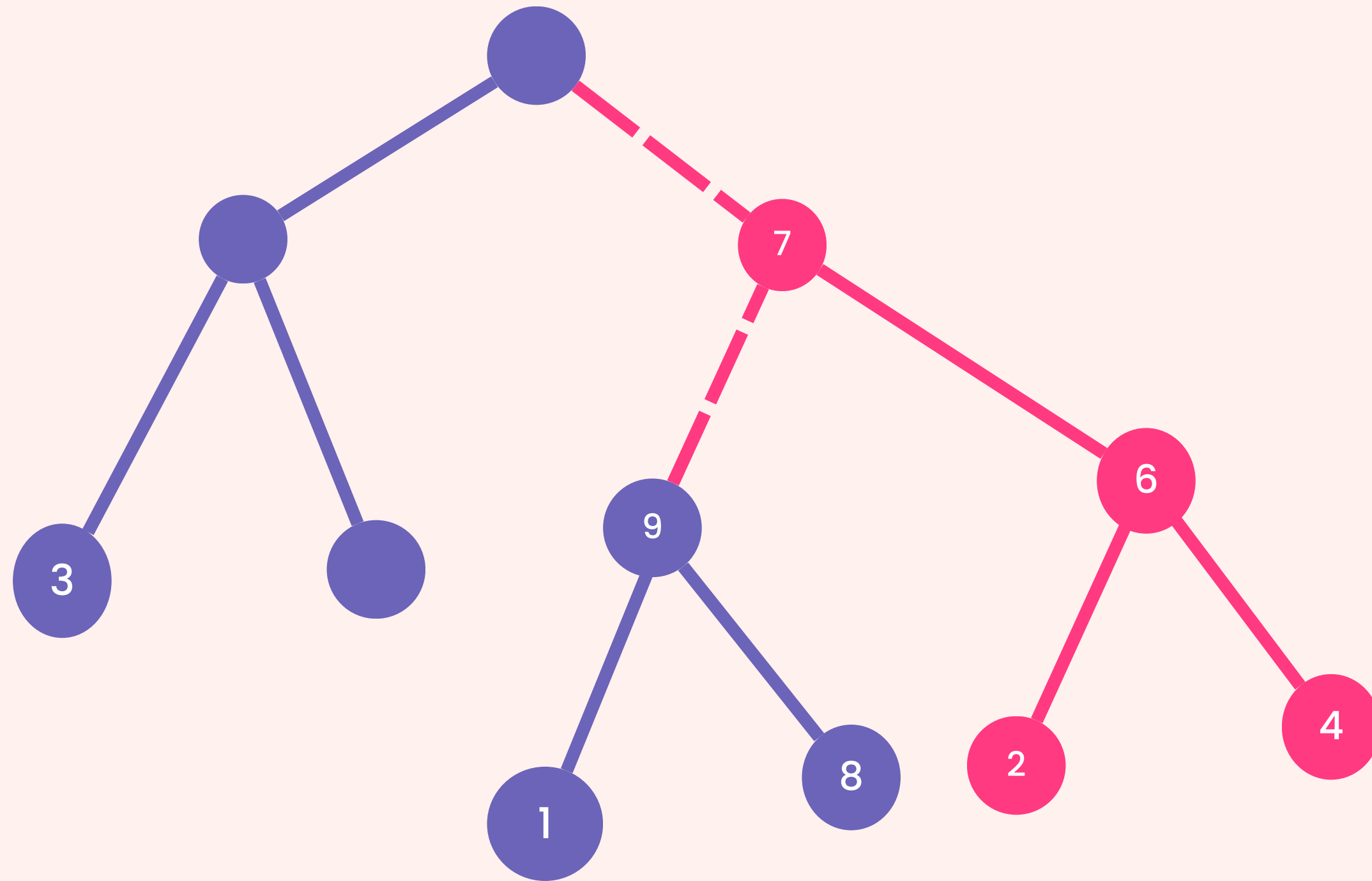
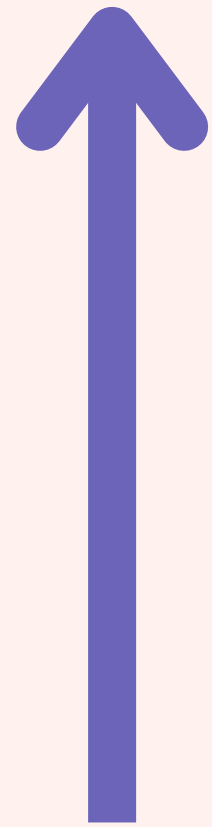


Root

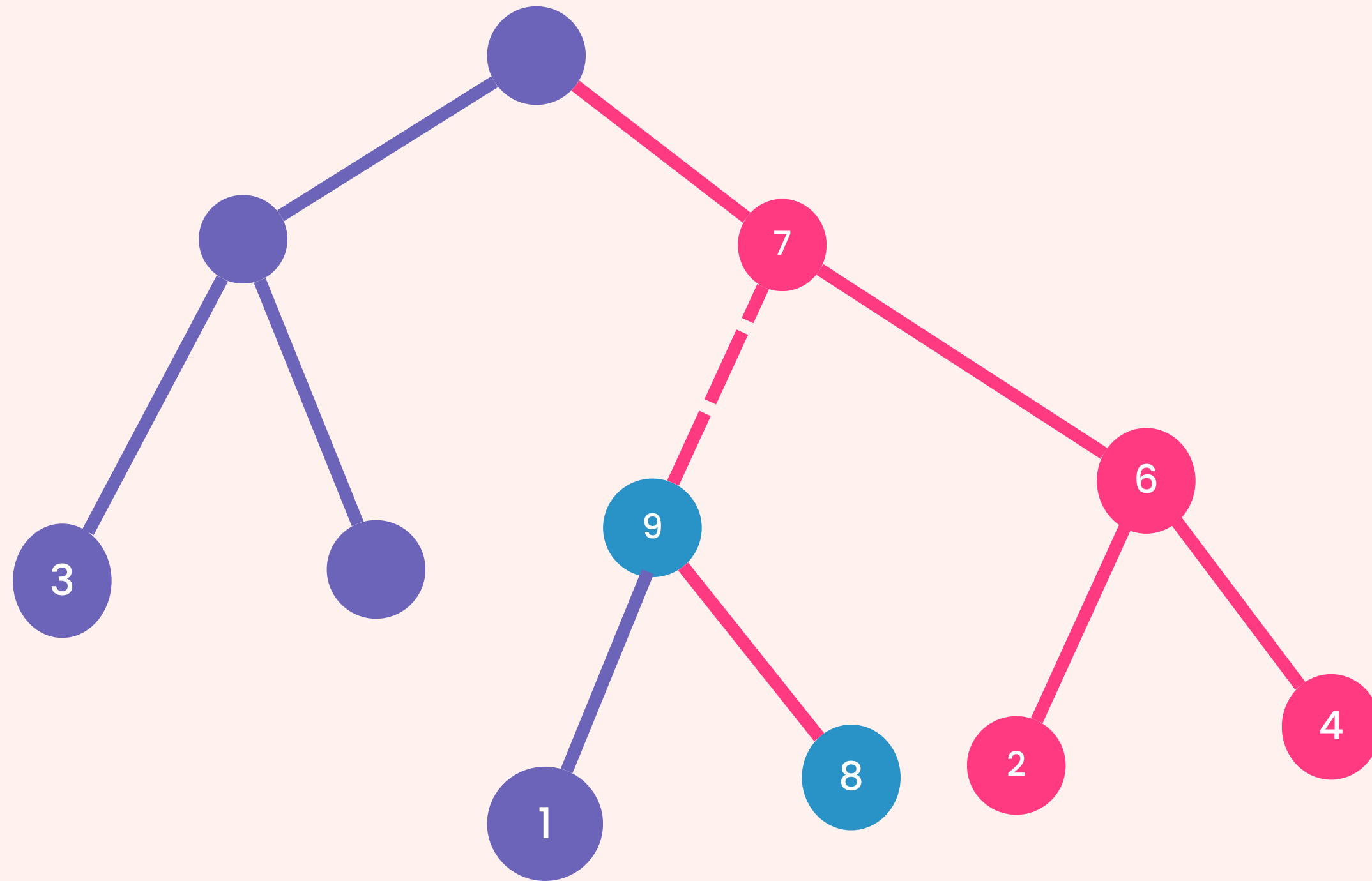
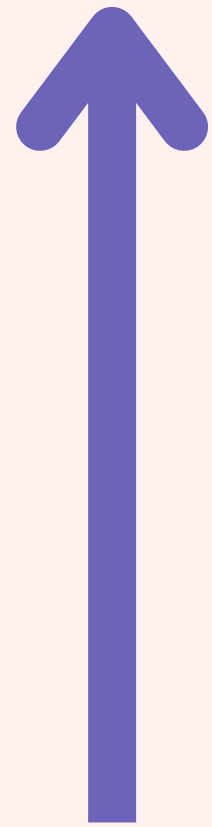




Root

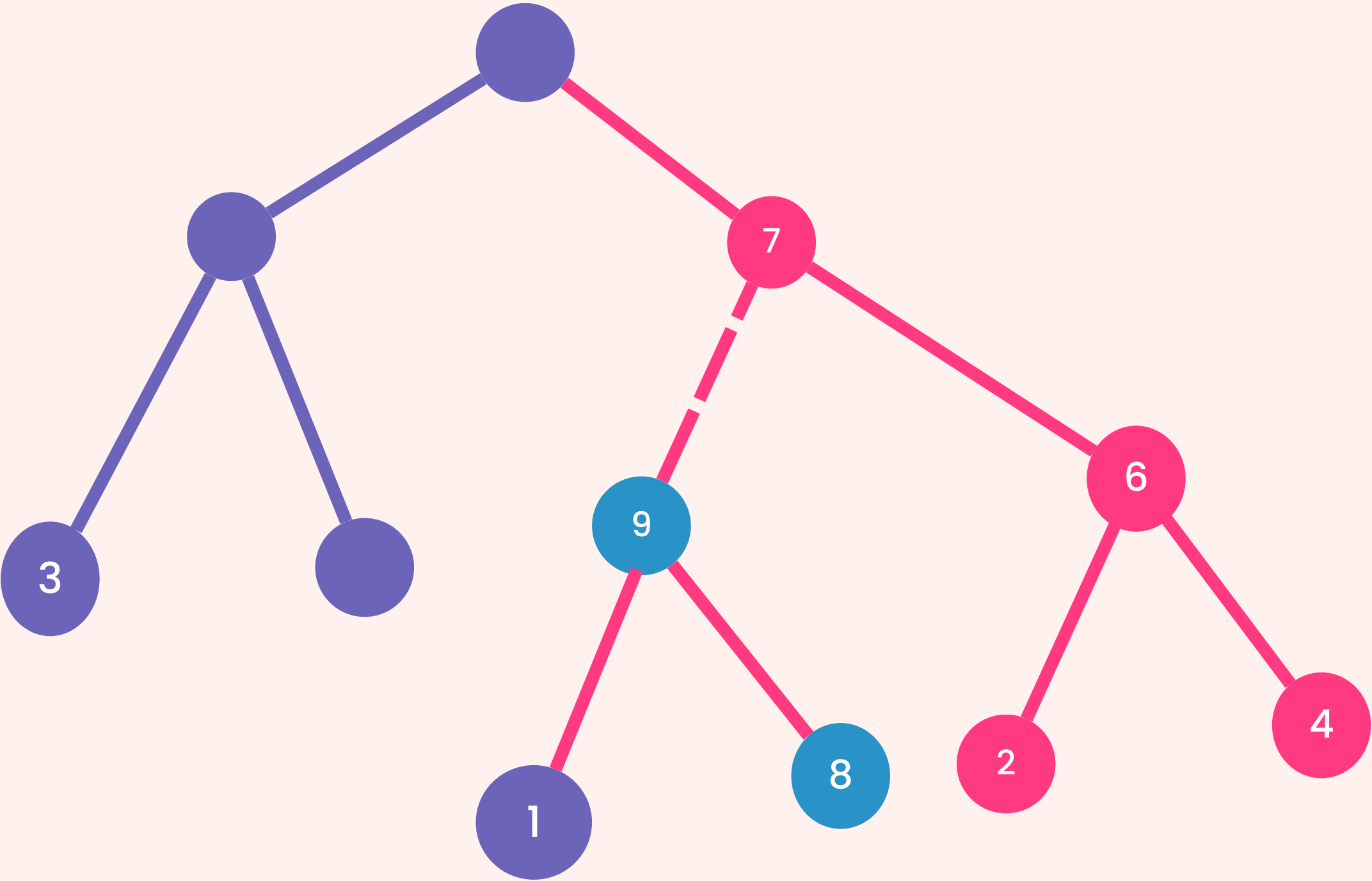
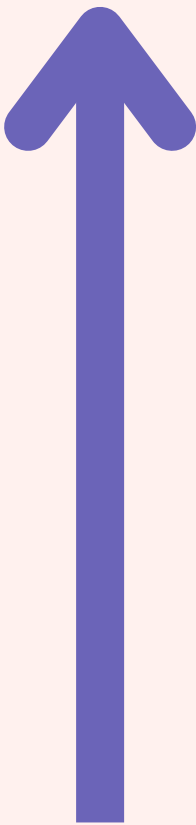


Root

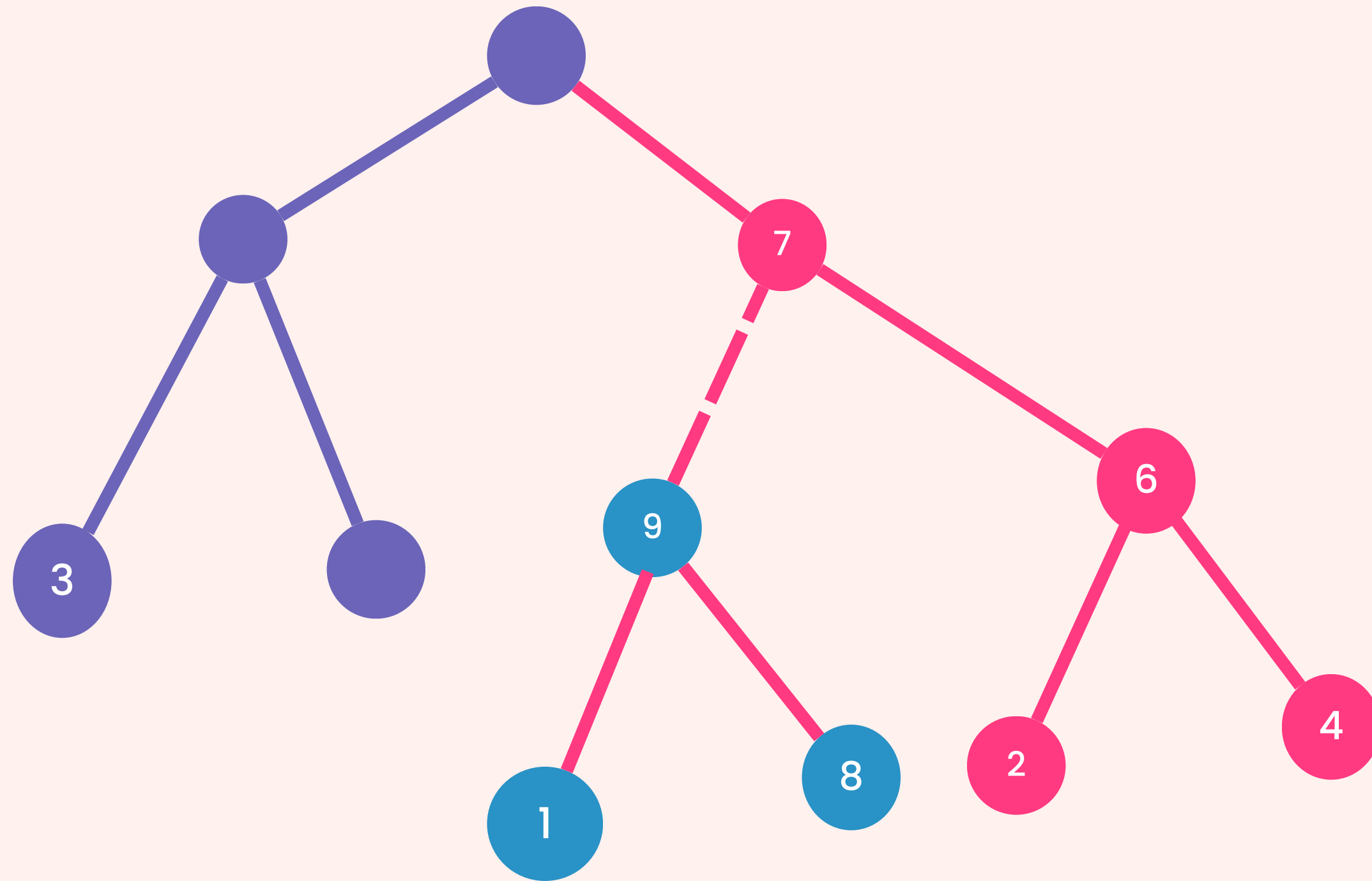
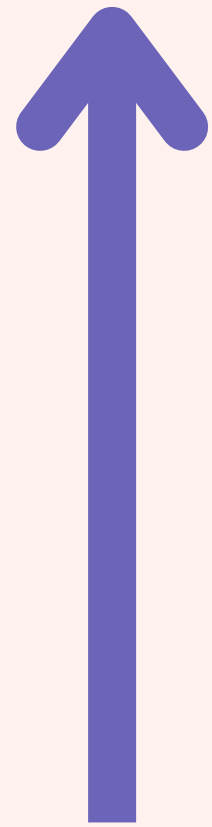


Query: 1 8, 1 2, 1 3

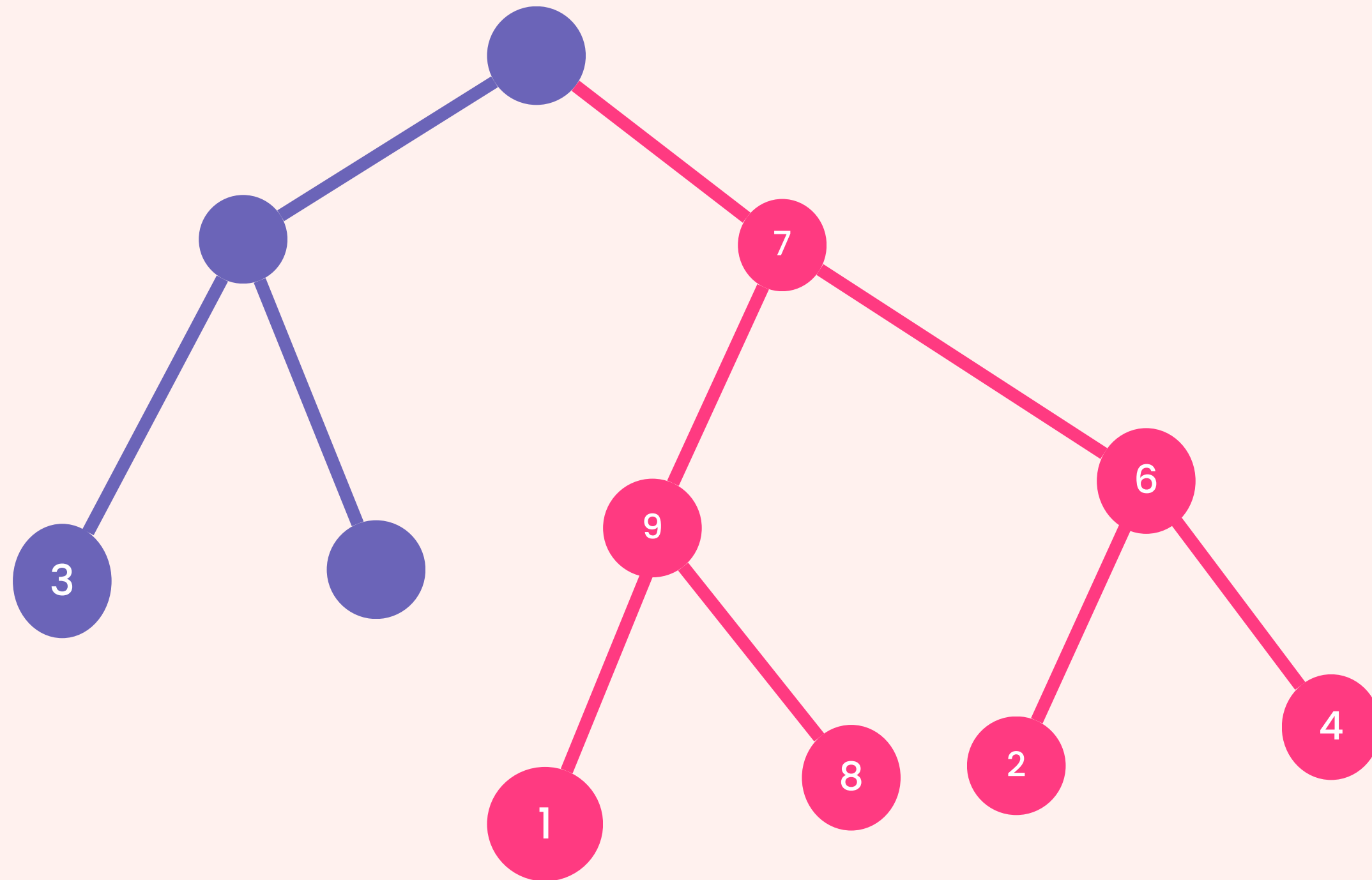
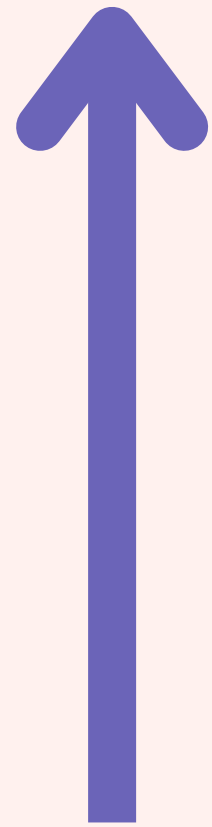
Root



Root



Root



## 4.10.2) CODE

```
void dfs(int v)
{
    visited[v] = true;
    ancestor[v] = v;
    for (int u : adj[v]) {
        if (!visited[u]) {
            dfs(u);
            union_sets(v, u);
            ancestor[find_set(v)] = v;
        }
    }
    for (int other_node : queries[v]) {
        if (visited[other_node])
            cout << "LCA of " << v << " and " << other_node
                << " is " << ancestor[find_set(other_node)] << ".\n";
    }
}

void compute_LCAs() {
    // initialize n, adj and DSU
    // for (each query (u, v)) {
    //     queries[u].push_back(v);
    //     queries[v].push_back(u);
    // }

    ancestor.resize(n);
    visited.assign(n, false);
    dfs(0);
}
```

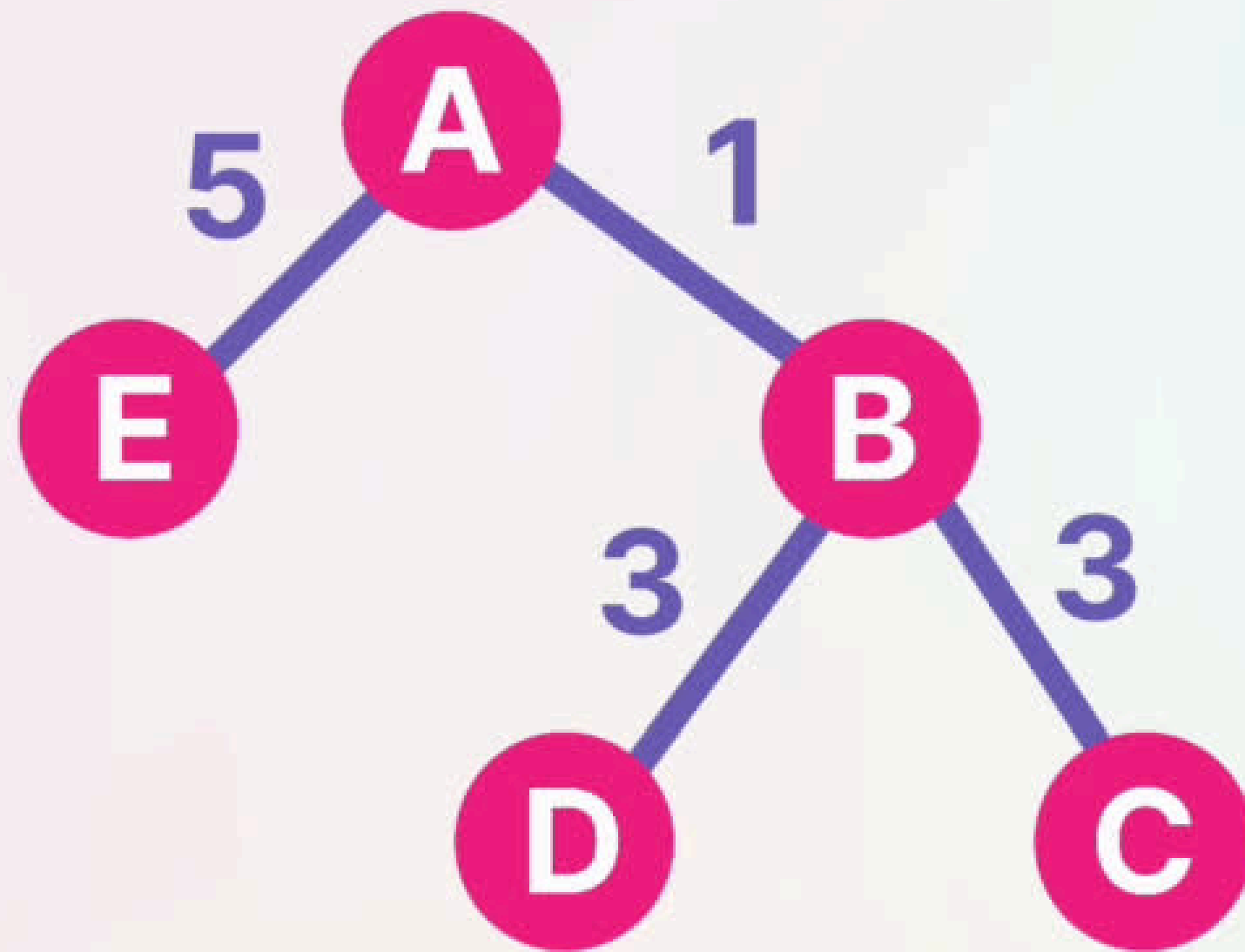
# APPLICATION IN TST 2023 - PROBLEM 1

- You are given a tree with  $n$  vertices and weighted edges. A limited number  $c$  is also given.
- You are asked to answer the following query: For a vertex  $u$ , how many vertices  $v$  are there such that the weight of the minimum edge between  $u$  and  $v$  is less than or equal to  $c$ ?
- How can this query be answered in  $O(\log n)$  time?

# APPLICATION IN TST 2023 - PROBLEM 1

- Let's discuss the concept of a DSU tree. I will construct a complete binary tree where the leaf nodes are the vertices of the original tree, and the remaining nodes represent the edge weights of the original tree. We need to ensure that every descendant  $v$  of  $u$  (except the leaves) satisfies  $v$  greater than or equal to  $u$ .
- From this tree, the smallest weight on the path from  $u$  to  $v$  will be the LCA of the two leaf nodes  $u$  and  $v$ .
- To construct this tree, I will apply DSU in the following way: (edit video animation).





**Sort the edge weights in descending order,**  
then process



# QNA SESSION 1

# QUIZ 1

What is the primary purpose of the Disjoint Set Union (DSU) data structure?

- A) To perform fast addition of elements to a set
- B) To quickly find and merge disjoint sets
- C) To sort elements in a set
- D) To search for an element in a sorted array



**B) TO QUICKLY FIND AND MERGE DISJOINT SETS**

# QUIZ 2

Which technique is used in DSU to keep the tree flat and optimize the union operation?

- A) Path compression
- B) Depth-first search
- C) Breadth-first search
- D) Binary search

## A) PATH COMPRESSION

# QUIZ 3

In the union by size heuristic, when joining two sets, which set is merged into which?

- A) The larger set is merged into the smaller set
- B) The sets are always of equal size
- C) The smaller set is merged into the larger set
- D) The sets are merged based on alphabetical order



**C) THE SMALLER SET IS MERGED INTO THE LARGER SET**







# QNA SESSION 2

# QUIZ 1

- In the set union technique, when performing the union operation in the disjoint set union (DSU) problem, if we ignore the condition of merging the smaller set into the larger set, what is the time complexity in the worst-case scenario? What is an example of such a case?

# QUIZ 1 - ANSWER

- The worst-case scenario is a path tree.
- Worst-case time complexity:  $O((n*(n - 1))/2*\log(n))$ .

# QUIZ 2

- Describe the idea to solve the following problem:
- Given a tree with  $n$  vertices, you receive  $q$  queries. For the  $i$ -th query, you are given two vertices  $u$  and  $v$  that are always directly connected by an edge, and you remove that edge from the tree. After this operation, what is the size of the largest connected component?

# QUIZ 2 - ANSWER

- We will review the queries from the end to the beginning, and perform the join\_set of the two edge pairs  $u v$  of  $i - 1$  to update the answer for  $i$ , and only need to use one variable to manage the largest size of the connected components. (Note that you must join\_set the edges that are not in the previous queries).

# QUIZ 3

- Describe the idea to solve the following problem:
- You are given a tree with  $n$  vertices where each vertex  $u$  has a color  $\text{color}[u]$ . For  $q$  queries, each query  $i$  asks for the number of distinct colors in the subtree rooted at vertex  $u$ .

# QUIZ 3 - ANSWER

- Using the set union technique, each set of connected components stores a set of different colors. When joining two sets, the smaller set is merged into the larger set.

The background is a light cream color. It features several decorative elements: a blue wavy shape in the top right corner with a yellow rounded square and three orange dots; an orange wavy shape in the bottom left corner with a yellow rounded square and two orange dots; a blue floral outline in the top left; a pink floral outline in the bottom right; and three dashed brown lines forming curved paths across the top, left, and bottom of the page.

# THANK YOU

- APCS K23 -



$$u \in V, Count \quad \{v \in V \mid \min_{P(u,v)} \text{weight}(P) \leq c\}$$