Graph (III)

Kelvin Chow {Lrt1088}

2023-03-25

Table of Contents

Overview & Revision

Prerequisite

- **Graph (I)**
  - **basic concepts, graph representation,** grid graph,
  - **depth first search,** flood fill, **breadth first search**
- Graph (II)
  - shortest path algorithms for weighted graphs,
  - minimum spanning tree
- **Data Structures (III)**
  - **sparse table, segment tree,**
  - **lazy propagation,** 1-d/2-d binary indexed tree
- **Dynamic Programming (I)**
  - **Knapsack Problems**

Overview

Tree:

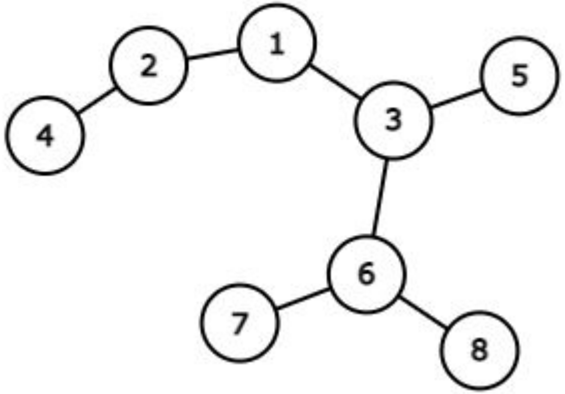
- Definition and properties

Algorithms on tree:

- Pre-order, in-order and post-order
- Tree diameter
- Lowest common ancestor
- (DAG and topological sort)

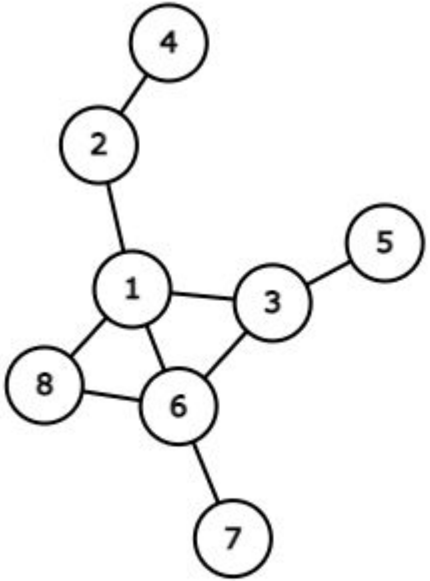
Revision: tree

- A tree is a connected graph with **no cycles**.
- Vertices of a tree are also called nodes.
- A tree can be either weighted or unweighted, and either rooted or unrooted.



← Tree

Not a tree →
∵ has cycles
(e.g. {1, 6, 8})



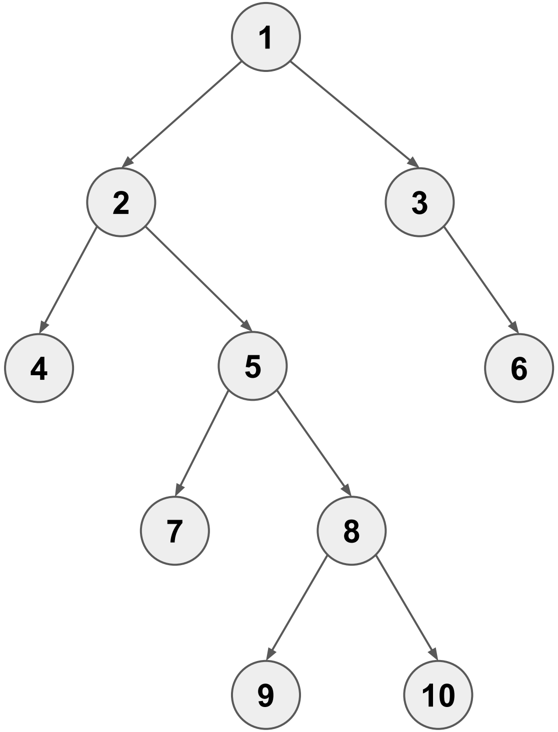
Revision: tree

- There are various equivalent definitions of a tree:
    - A connected graph with V vertices and $V - 1$ edges.
    - A connected graph with no cycles.
    - Between any two vertices on the graph, there is only one simple (also the shortest) path between them.
- These properties make problems easier to solve.

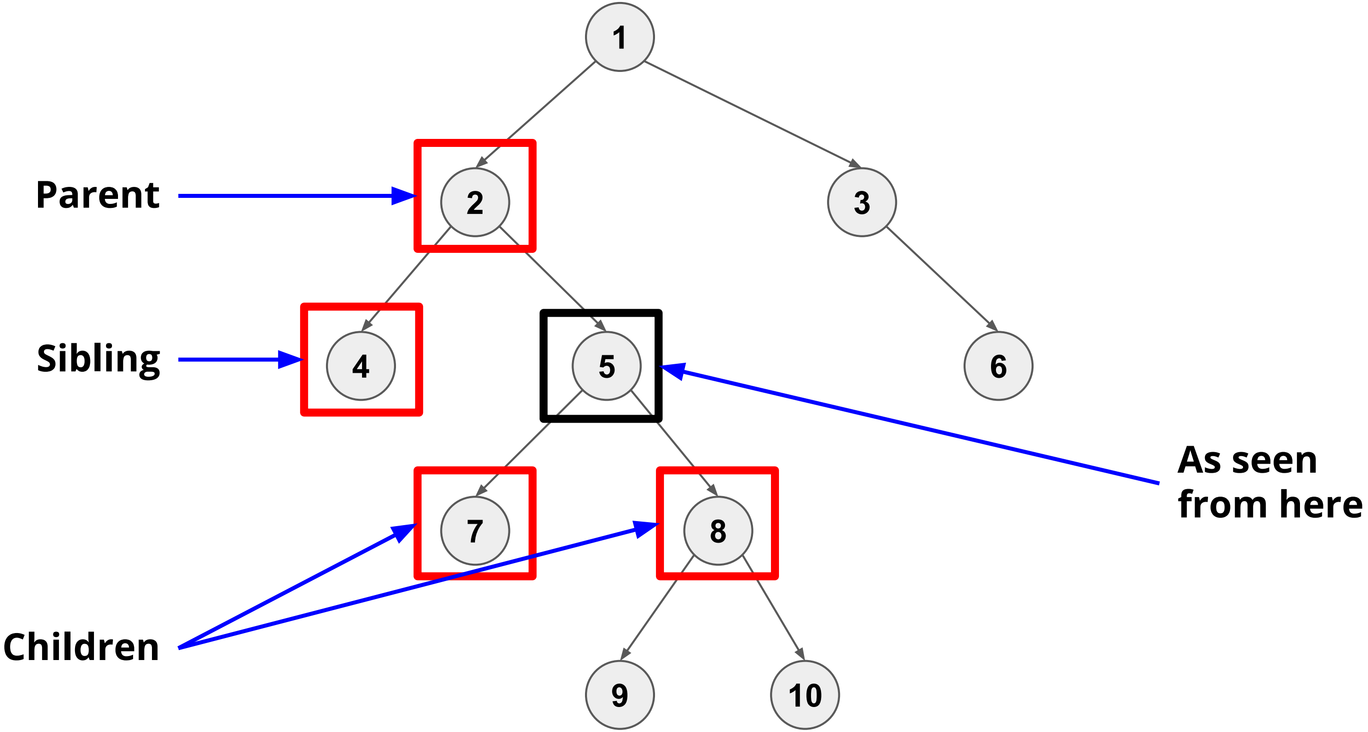
Rooted Tree

Rooted tree

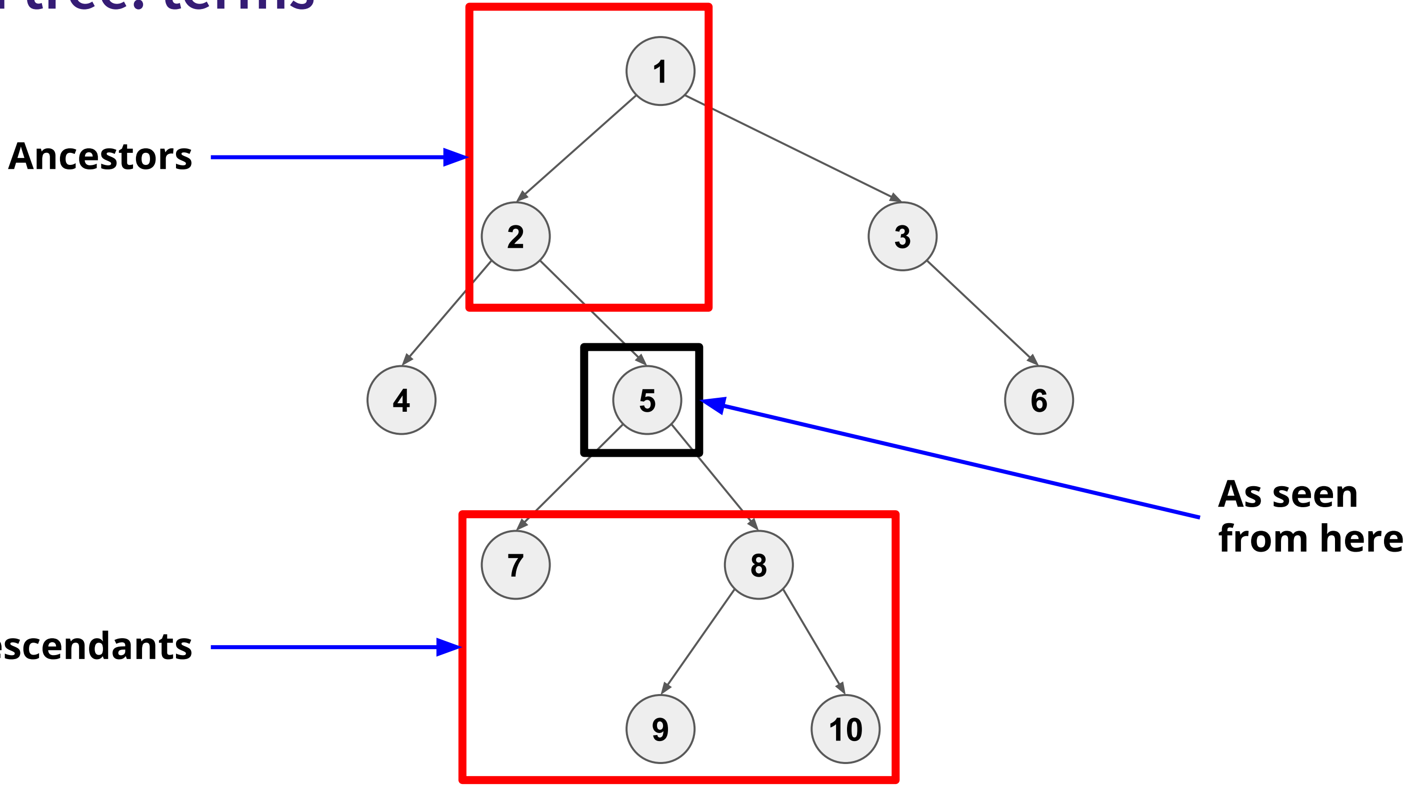
- Sometimes one of the nodes of the tree will be viewed as the **root** of the tree
- Then the tree becomes directed
- (Note: if there is no root, we sometimes choose any node as the root)



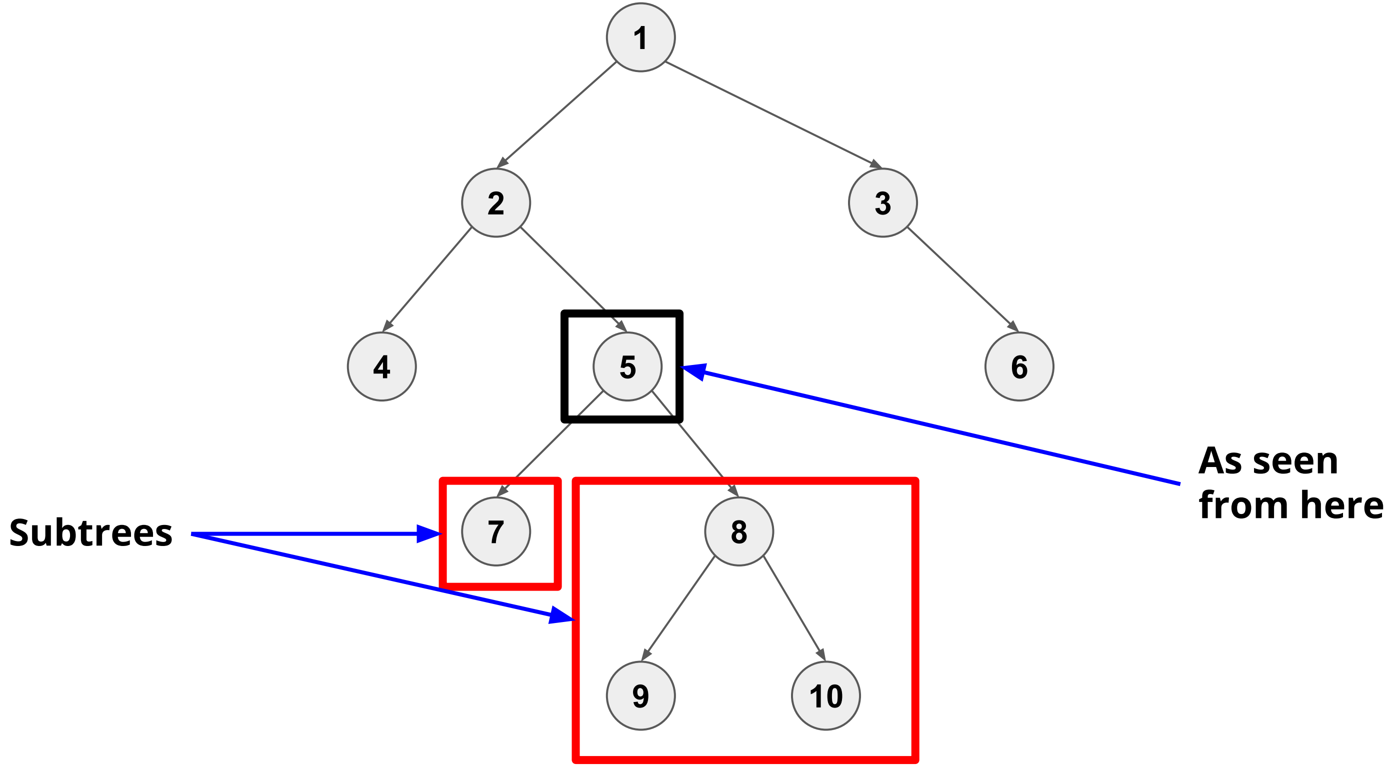
Rooted tree: terms



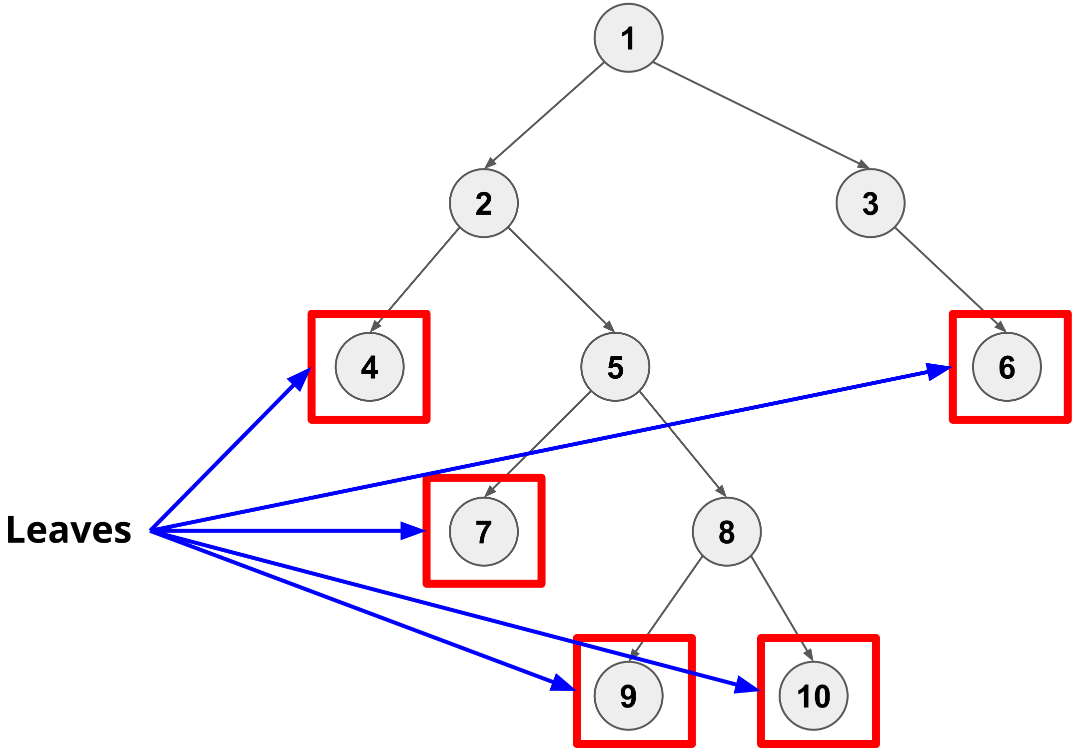
Rooted tree: terms



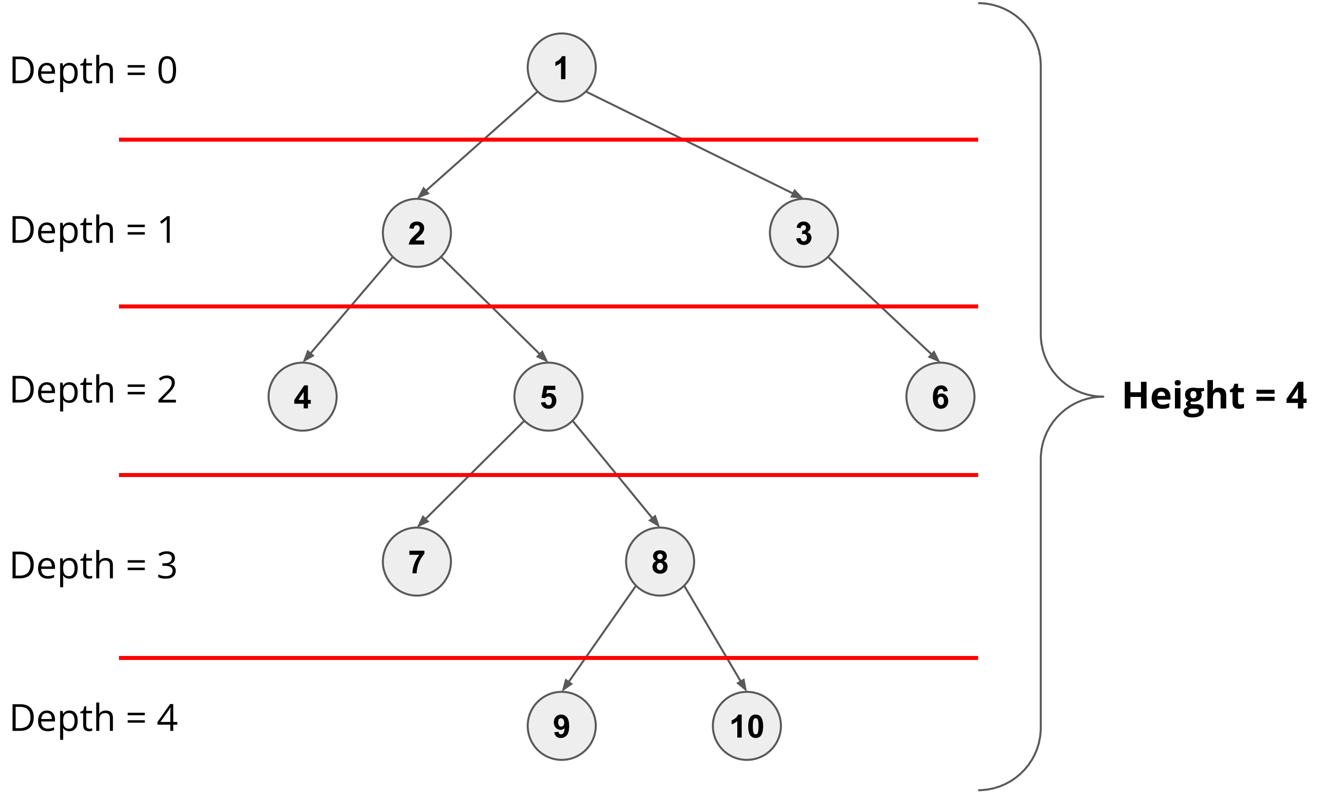
Rooted tree: terms



Rooted tree: terms



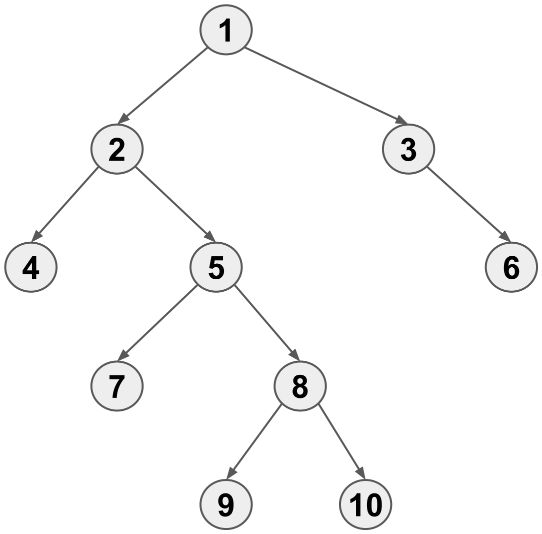
Rooted tree: terms



Tree: implementation

- Trees are graph so the same representations are used.
  - e.g. adjacency matrix, adjacency list, edge list
- For rooted tree, we can choose to store the parent and children separately.

Node	Parent	Child[0]	Child[1]
1		2	3
2	1	4	5
3	1	6	
4	2		
5	2	7	8
6	3		
7	5		
8	5	9	10
9	8		
10	8		



Tree: application

Some graph problems are trivial in trees.

- Shortest path between two nodes → the only path between two nodes
- Minimum spanning tree → the tree itself

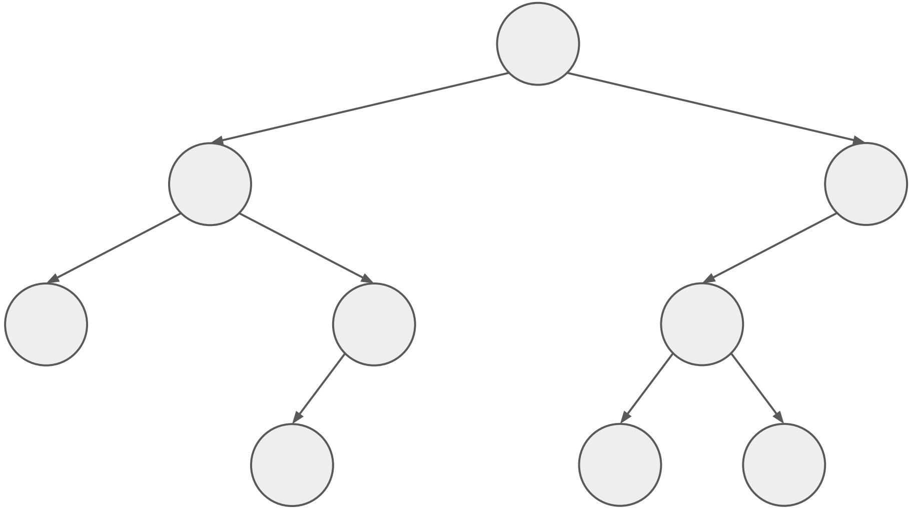
Trees are also used in data structures.

- Binary search tree
- Heap
- Trie
- Segment Tree
- Suffix Tree

Binary Tree

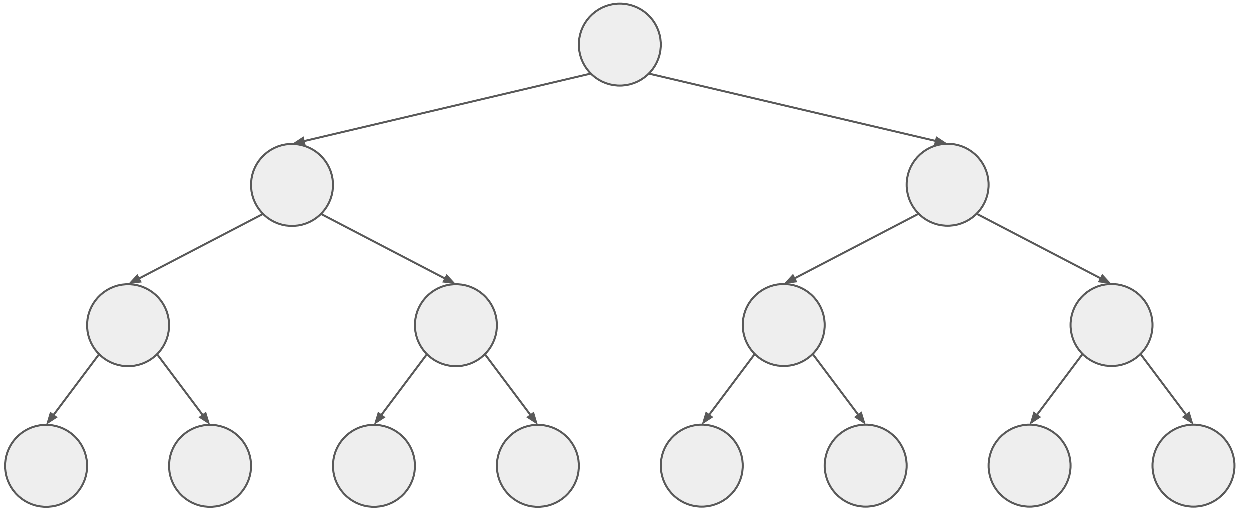
Binary Tree

A rooted tree where all vertices (nodes) have at most 2 children.



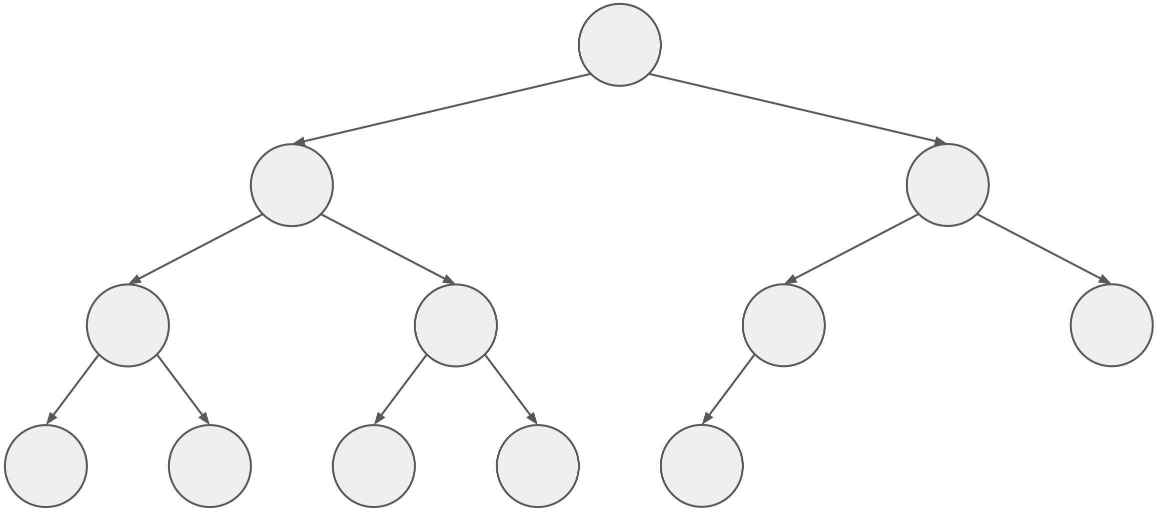
Perfect Binary Tree

A rooted tree where all vertices (nodes) have 2 children and all leaves have the same depth.



Complete Binary Tree

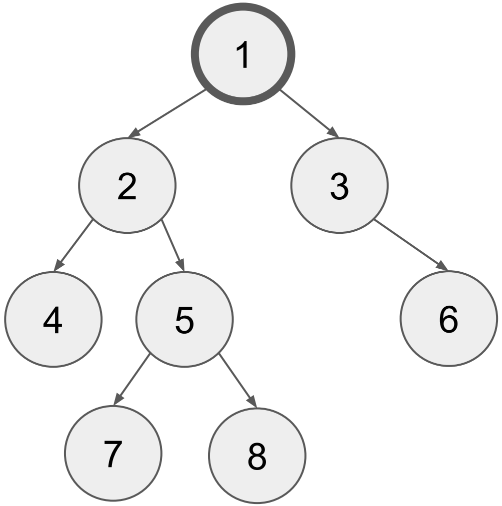
A perfect binary tree with some or all rightmost leaf nodes removed.



Tree traversal

Tree traversal

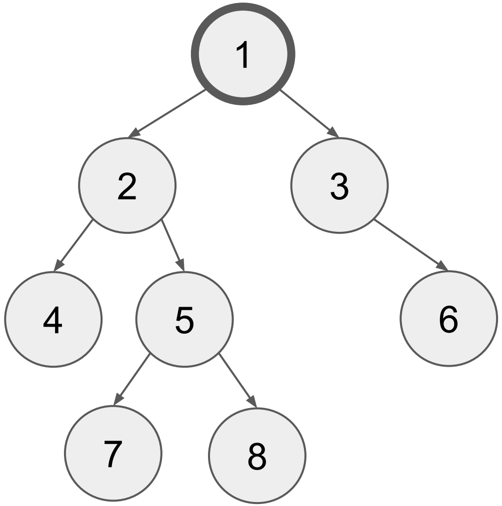
We can perform DFS on trees as we do on graph.




```
void dfs(int node) {
    for (auto child : children[node])
        dfs(child);
}
```

Tree traversal orders

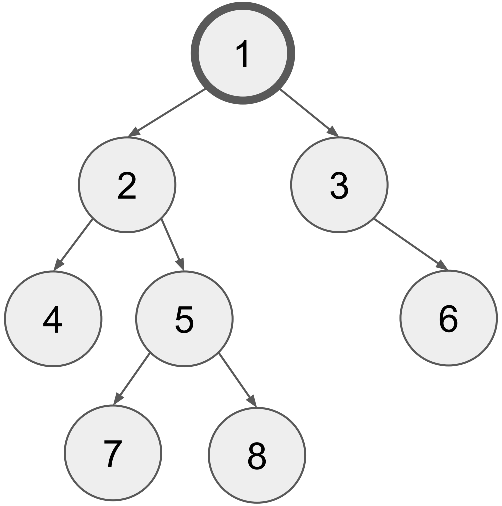
For binary trees, there are 3 common traversal orders:
Pre-order, **in-order** and **post-order**.



Pre-order	1	2	4	5	7	8	3	6
In-order	4	2	7	5	8	1	3	6
Post-order	4	7	8	5	2	6	3	1

Pre-order

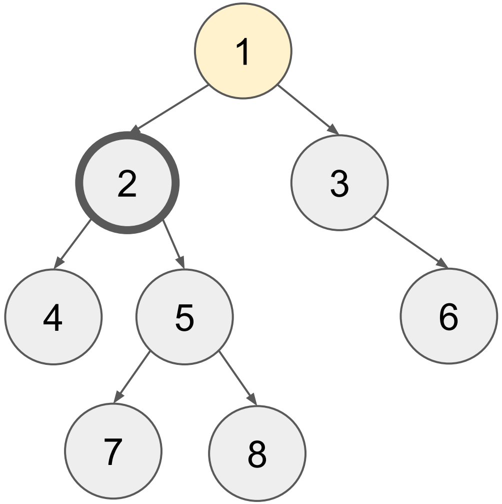
1							




```
void preorder_traversal(int node) {
    process(node);

    if (left_child[node])
        preorder_traversal(left_child[node]);

    if (right_child[node])
        preorder_traversal(right_child[node]);
}
```

Pre-order

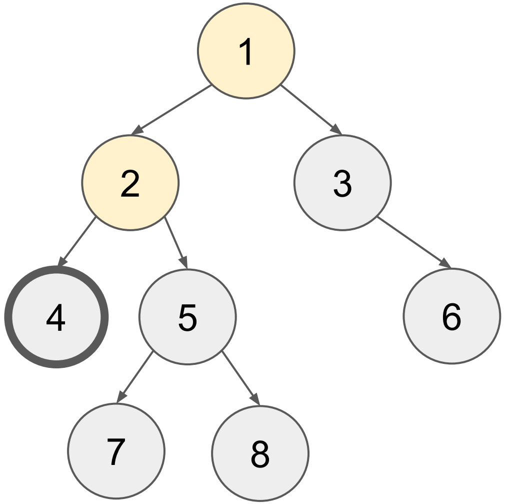
1	2						




```
void preorder_traversal(int node) {
    process(node);

    if (left_child[node])
        preorder_traversal(left_child[node]);

    if (right_child[node])
        preorder_traversal(right_child[node]);
}
```

Pre-order

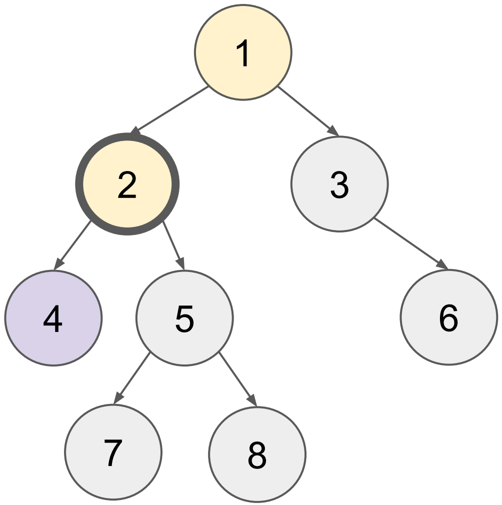
1	2	4					




```
void preorder_traversal(int node) {
    process(node);

    if (left_child[node])
        preorder_traversal(left_child[node]);

    if (right_child[node])
        preorder_traversal(right_child[node]);
}
```

Pre-order

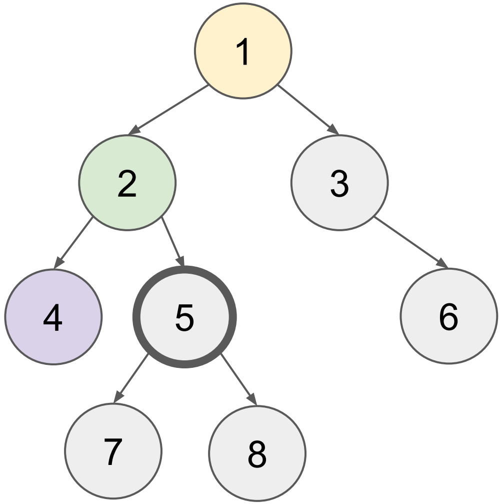
1	2	4					




```
void preorder_traversal(int node) {
    process(node);

    if (left_child[node])
        preorder_traversal(left_child[node]);

    if (right_child[node])
        preorder_traversal(right_child[node]);
}
```

Pre-order

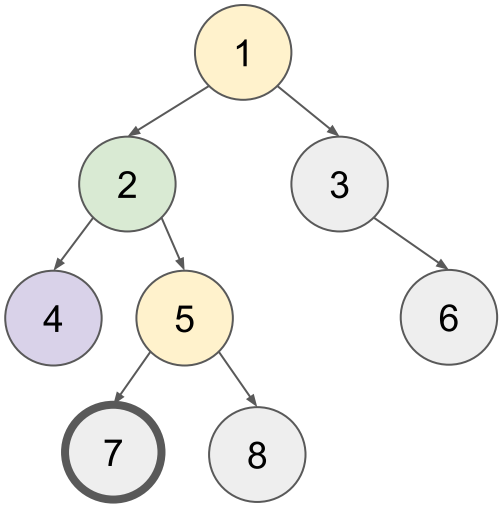
1	2	4	5				




```
void preorder_traversal(int node) {
    process(node);

    if (left_child[node])
        preorder_traversal(left_child[node]);

    if (right_child[node])
        preorder_traversal(right_child[node]);
}
```

Pre-order

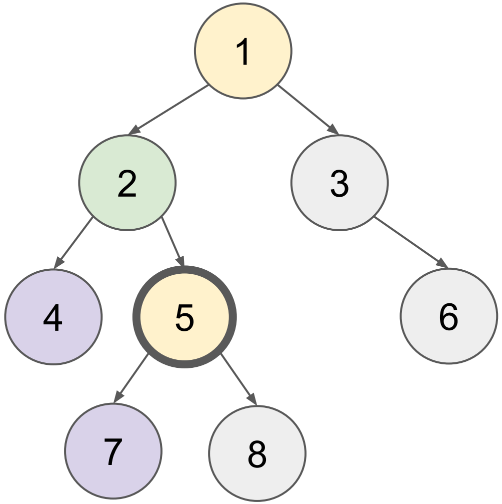
1	2	4	5	7			




```
void preorder_traversal(int node) {
    process(node);

    if (left_child[node])
        preorder_traversal(left_child[node]);

    if (right_child[node])
        preorder_traversal(right_child[node]);
}
```

Pre-order

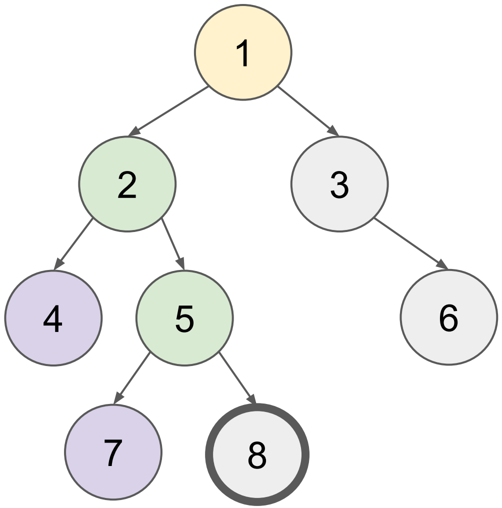
1	2	4	5	7			




```
void preorder_traversal(int node) {
    process(node);

    if (left_child[node])
        preorder_traversal(left_child[node]);

    if (right_child[node])
        preorder_traversal(right_child[node]);
}
```

Pre-order

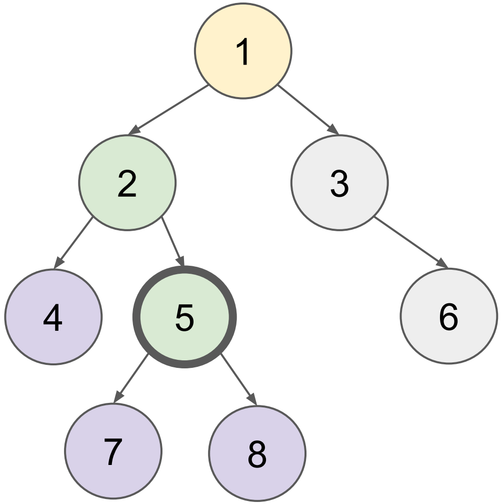
1	2	4	5	7	8		




```
void preorder_traversal(int node) {
    process(node);

    if (left_child[node])
        preorder_traversal(left_child[node]);

    if (right_child[node])
        preorder_traversal(right_child[node]);
}
```

Pre-order

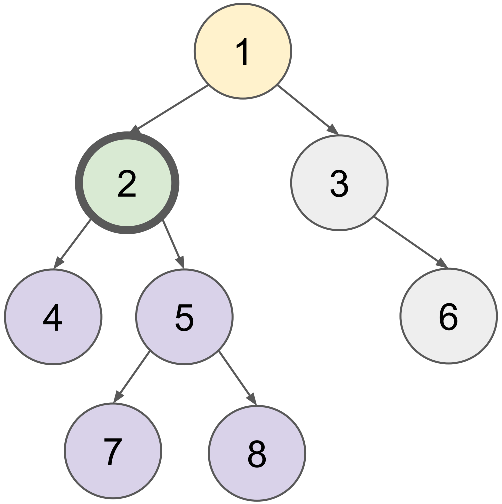
1	2	4	5	7	8		




```
void preorder_traversal(int node) {
    process(node);

    if (left_child[node])
        preorder_traversal(left_child[node]);

    if (right_child[node])
        preorder_traversal(right_child[node]);
}
```

Pre-order

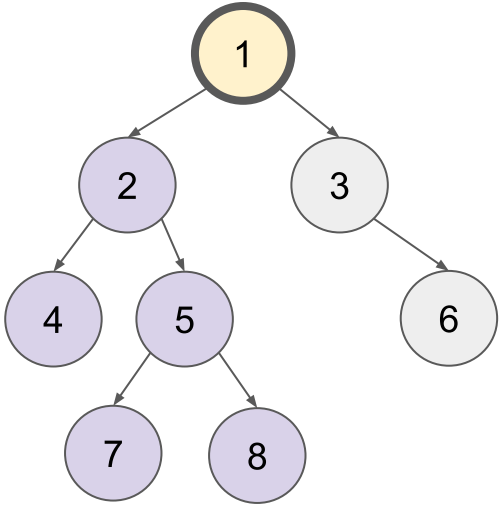
1	2	4	5	7	8		




```
void preorder_traversal(int node) {
    process(node);

    if (left_child[node])
        preorder_traversal(left_child[node]);

    if (right_child[node])
        preorder_traversal(right_child[node]);
}
```

Pre-order

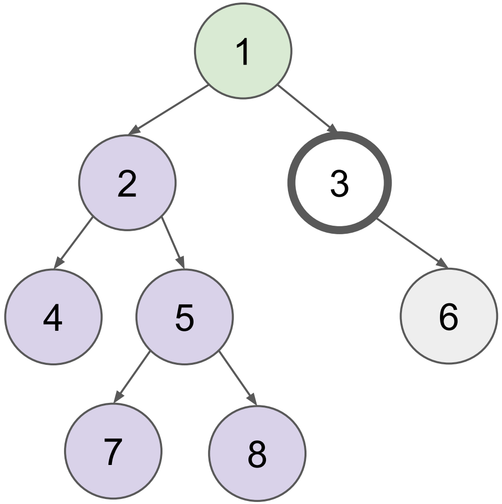
1	2	4	5	7	8		




```
void preorder_traversal(int node) {
    process(node);

    if (left_child[node])
        preorder_traversal(left_child[node]);

    if (right_child[node])
        preorder_traversal(right_child[node]);
}
```

Pre-order

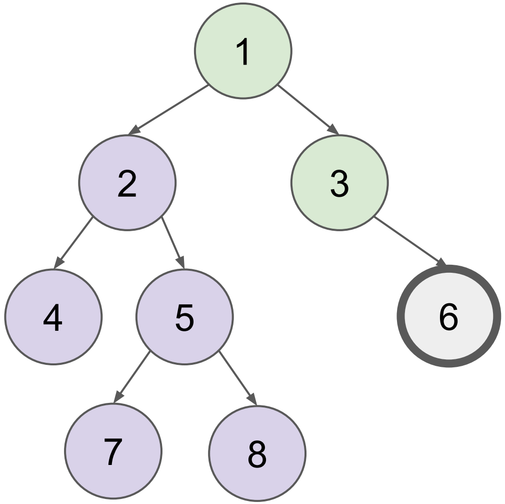
1	2	4	5	7	8	3	




```
void preorder_traversal(int node) {
    process(node);

    if (left_child[node])
        preorder_traversal(left_child[node]);

    if (right_child[node])
        preorder_traversal(right_child[node]);
}
```

Pre-order

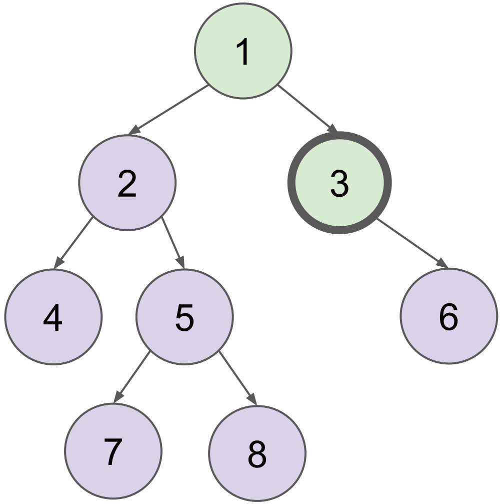
1	2	4	5	7	8	3	6




```
void preorder_traversal(int node) {
    process(node);

    if (left_child[node])
        preorder_traversal(left_child[node]);

    if (right_child[node])
        preorder_traversal(right_child[node]);
}
```

Pre-order

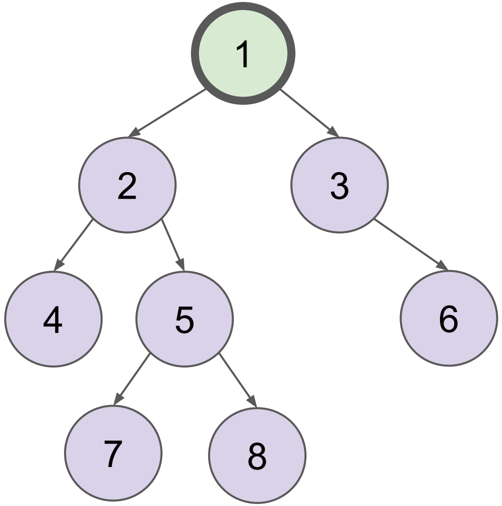
1	2	4	5	7	8	3	6




```
void preorder_traversal(int node) {
    process(node);

    if (left_child[node])
        preorder_traversal(left_child[node]);

    if (right_child[node])
        preorder_traversal(right_child[node]);
}
```

Pre-order

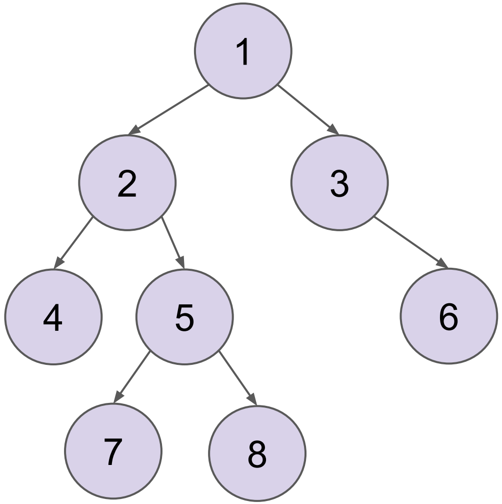
1	2	4	5	7	8	3	6




```
void preorder_traversal(int node) {
    process(node);

    if (left_child[node])
        preorder_traversal(left_child[node]);

    if (right_child[node])
        preorder_traversal(right_child[node]);
}
```

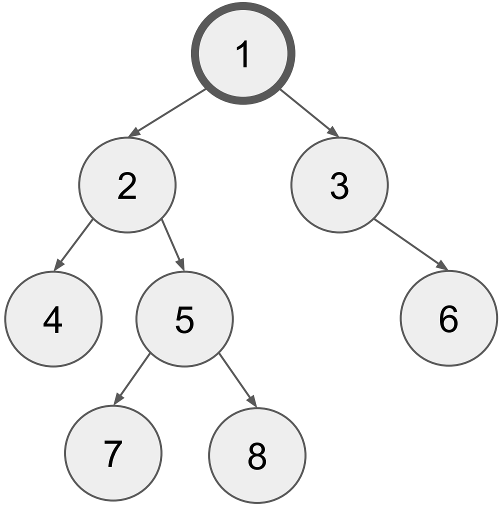
Pre-order

1	2	4	5	7	8	3	6



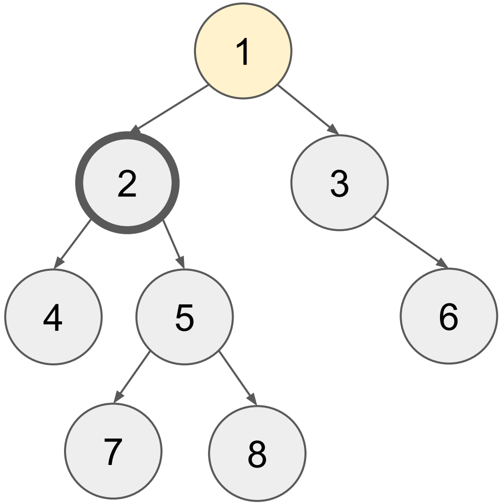

```
void preorder_traversal(int node) {
    process(node);

    if (left_child[node])
        preorder_traversal(left_child[node]);

    if (right_child[node])
        preorder_traversal(right_child[node]);
}
```

In-order




```
void inorder_traversal(int node) {
    if (left_child[node])
        inorder_traversal(left_child[node]);

    process(node);

    if (right_child[node])
        inorder_traversal(right_child[node]);
}
```

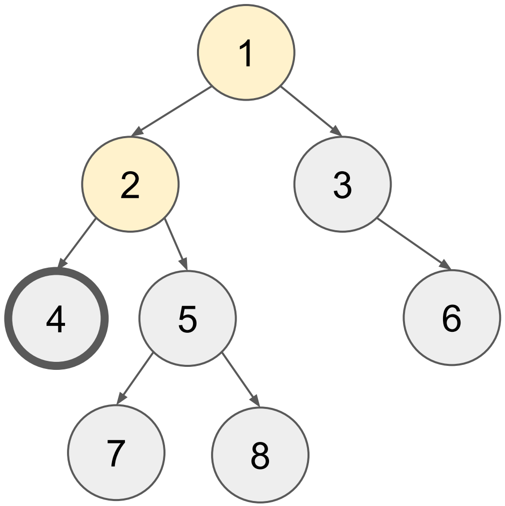
In-order




```
void inorder_traversal(int node) {
    if (left_child[node])
        inorder_traversal(left_child[node]);

    process(node);

    if (right_child[node])
        inorder_traversal(right_child[node]);
}
```

In-order

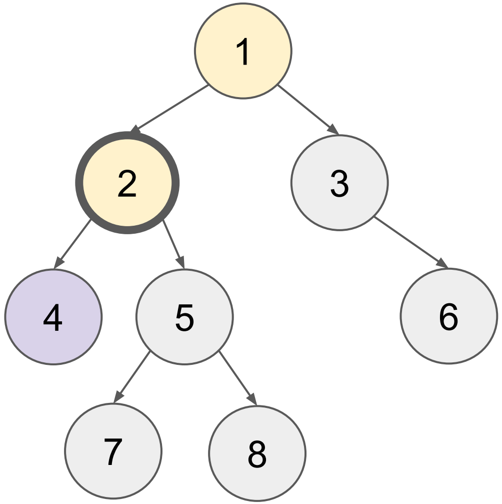
4							




```
void inorder_traversal(int node) {
    if (left_child[node])
        inorder_traversal(left_child[node]);

    process(node);

    if (right_child[node])
        inorder_traversal(right_child[node]);
}
```

In-order

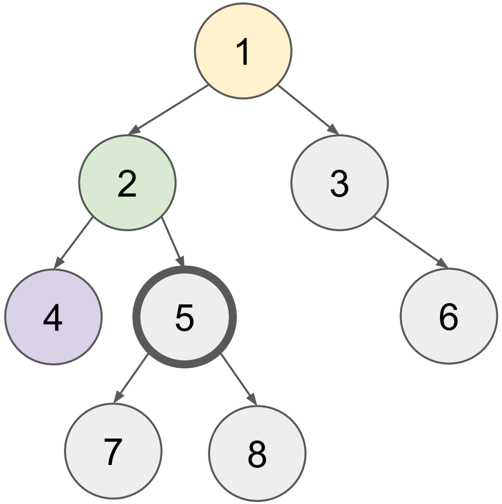
4	2						




```
void inorder_traversal(int node) {
    if (left_child[node])
        inorder_traversal(left_child[node]);

    process(node);

    if (right_child[node])
        inorder_traversal(right_child[node]);
}
```

In-order

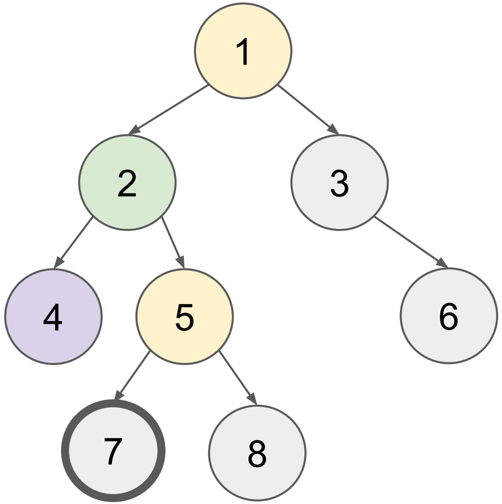
4	2						




```
void inorder_traversal(int node) {
    if (left_child[node])
        inorder_traversal(left_child[node]);

    process(node);

    if (right_child[node])
        inorder_traversal(right_child[node]);
}
```

In-order

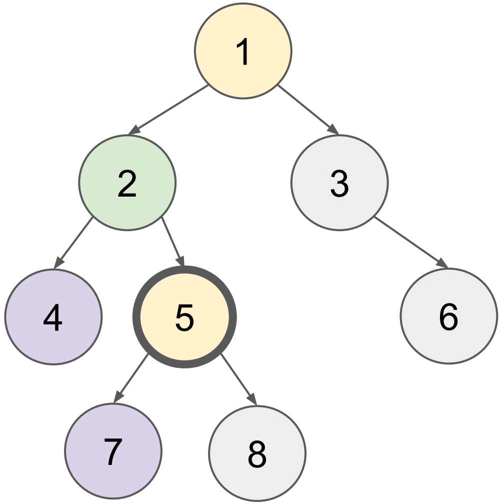
4	2	7					




```
void inorder_traversal(int node) {
    if (left_child[node])
        inorder_traversal(left_child[node]);

    process(node);

    if (right_child[node])
        inorder_traversal(right_child[node]);
}
```

In-order

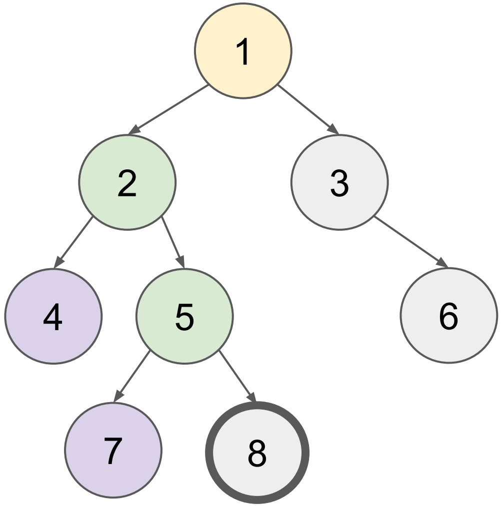
4	2	7	5				




```
void inorder_traversal(int node) {
    if (left_child[node])
        inorder_traversal(left_child[node]);

    process(node);

    if (right_child[node])
        inorder_traversal(right_child[node]);
}
```

In-order

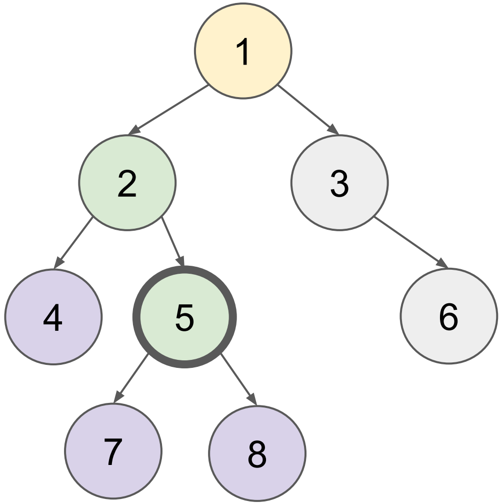
4	2	7	5	8			




```
void inorder_traversal(int node) {
    if (left_child[node])
        inorder_traversal(left_child[node]);

    process(node);

    if (right_child[node])
        inorder_traversal(right_child[node]);
}
```

In-order

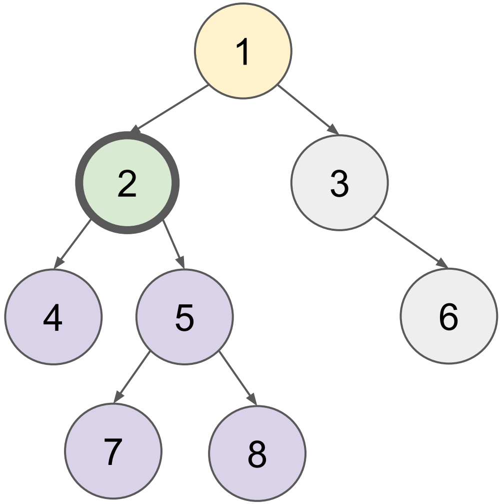
4	2	7	5	8			




```
void inorder_traversal(int node) {
    if (left_child[node])
        inorder_traversal(left_child[node]);

    process(node);

    if (right_child[node])
        inorder_traversal(right_child[node]);
}
```

In-order

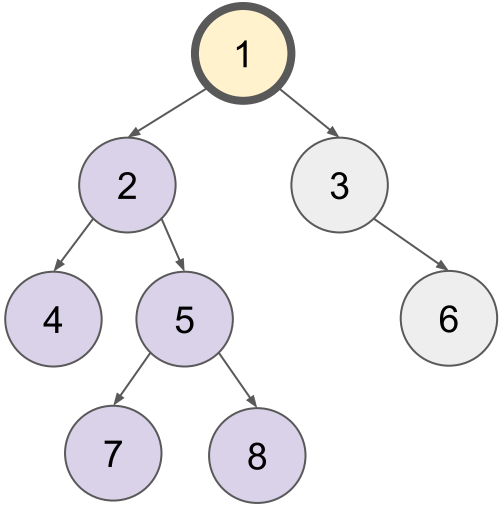
4	2	7	5	8			




```
void inorder_traversal(int node) {
    if (left_child[node])
        inorder_traversal(left_child[node]);

    process(node);

    if (right_child[node])
        inorder_traversal(right_child[node]);
}
```

In-order

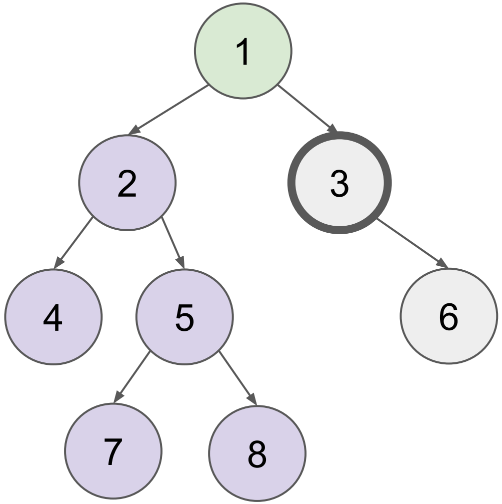
4	2	7	5	8	1		




```
void inorder_traversal(int node) {
    if (left_child[node])
        inorder_traversal(left_child[node]);

    process(node);

    if (right_child[node])
        inorder_traversal(right_child[node]);
}
```

In-order

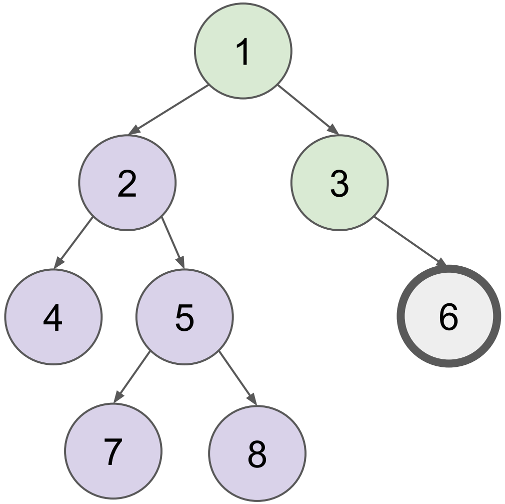
4	2	7	5	8	1	3	




```
void inorder_traversal(int node) {
    if (left_child[node])
        inorder_traversal(left_child[node]);

    process(node);

    if (right_child[node])
        inorder_traversal(right_child[node]);
}
```

In-order

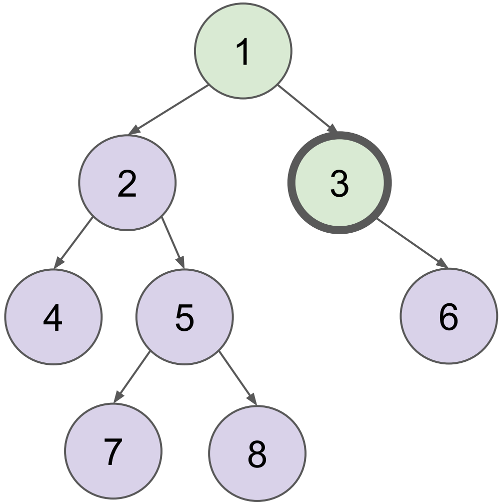
4	2	7	5	8	1	3	6




```
void inorder_traversal(int node) {
    if (left_child[node])
        inorder_traversal(left_child[node]);

    process(node);

    if (right_child[node])
        inorder_traversal(right_child[node]);
}
```

In-order

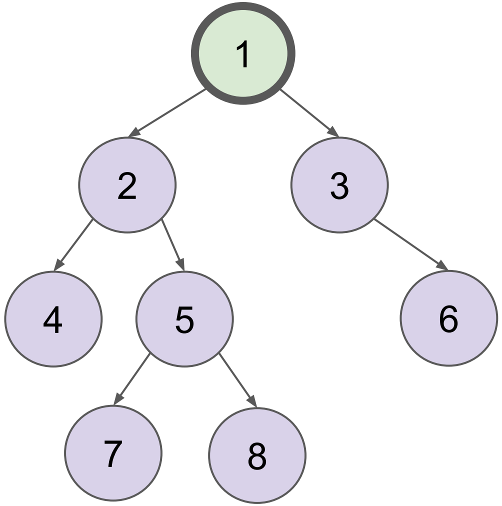
4	2	7	5	8	1	3	6




```
void inorder_traversal(int node) {
    if (left_child[node])
        inorder_traversal(left_child[node]);

    process(node);

    if (right_child[node])
        inorder_traversal(right_child[node]);
}
```

In-order

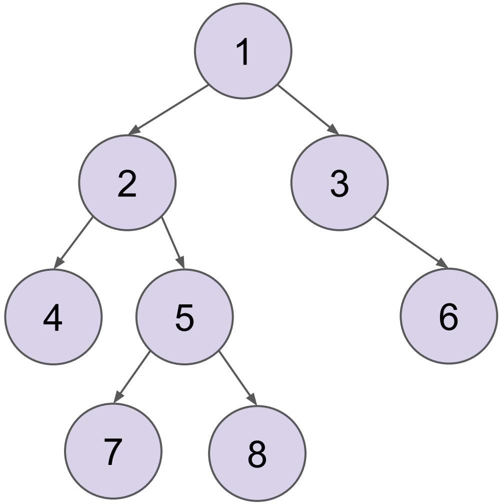
4	2	7	5	8	1	3	6




```
void inorder_traversal(int node) {
    if (left_child[node])
        inorder_traversal(left_child[node]);

    process(node);

    if (right_child[node])
        inorder_traversal(right_child[node]);
}
```

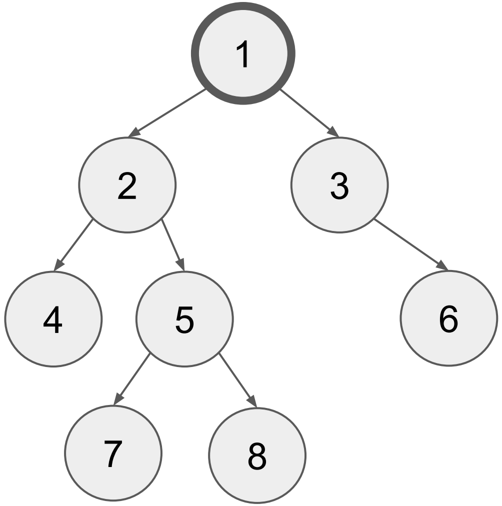
In-order

4	2	7	5	8	1	3	6



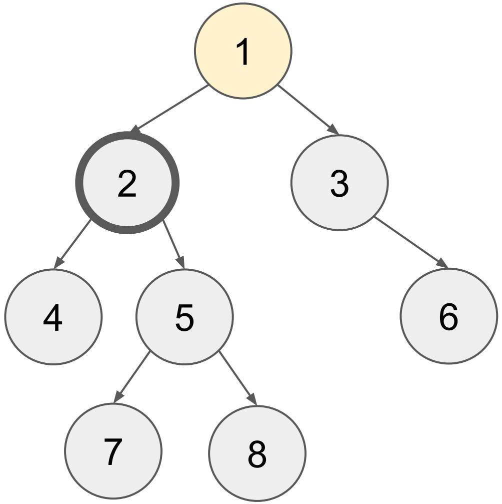

```
void inorder_traversal(int node) {
    if (left_child[node])
        inorder_traversal(left_child[node]);

    process(node);

    if (right_child[node])
        inorder_traversal(right_child[node]);
}
```

Post-order




```
void postorder_traversal(int node) {
    if (left_child[node])
        postorder_traversal(left_child[node]);

    if (right_child[node])
        postorder_traversal(right_child[node]);

    process(node);
}
```

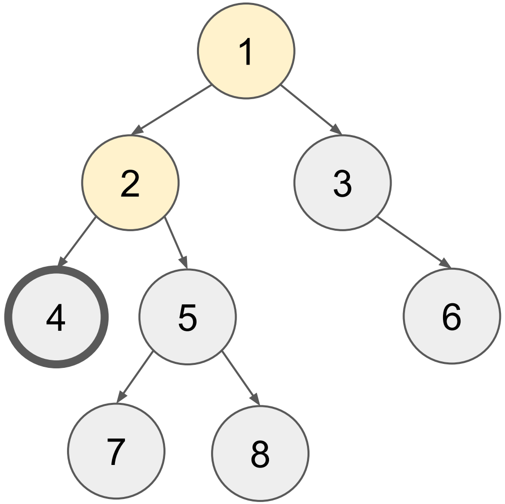
Post-order




```
void postorder_traversal(int node) {
    if (left_child[node])
        postorder_traversal(left_child[node]);

    if (right_child[node])
        postorder_traversal(right_child[node]);

    process(node);
}
```

Post-order

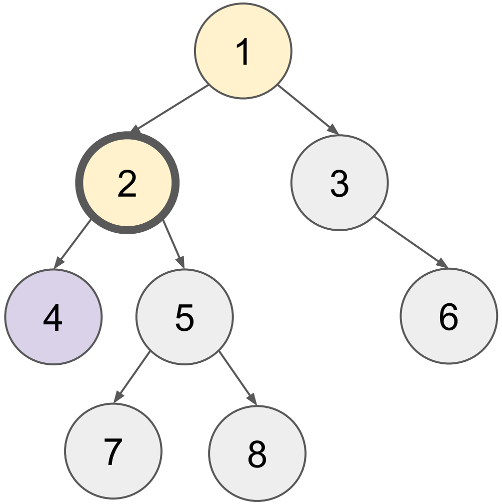
4							




```
void postorder_traversal(int node) {
    if (left_child[node])
        postorder_traversal(left_child[node]);

    if (right_child[node])
        postorder_traversal(right_child[node]);

    process(node);
}
```

Post-order

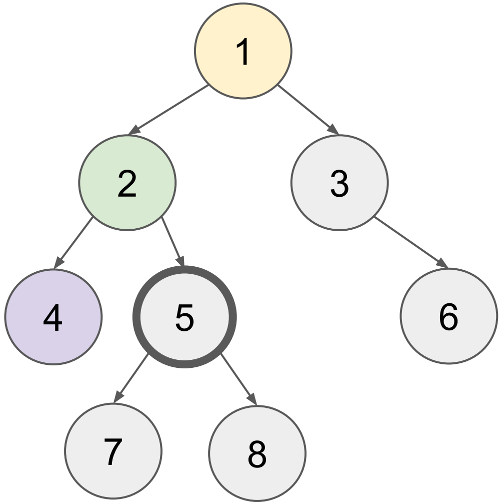
4							




```
void postorder_traversal(int node) {
    if (left_child[node])
        postorder_traversal(left_child[node]);

    if (right_child[node])
        postorder_traversal(right_child[node]);

    process(node);
}
```

Post-order

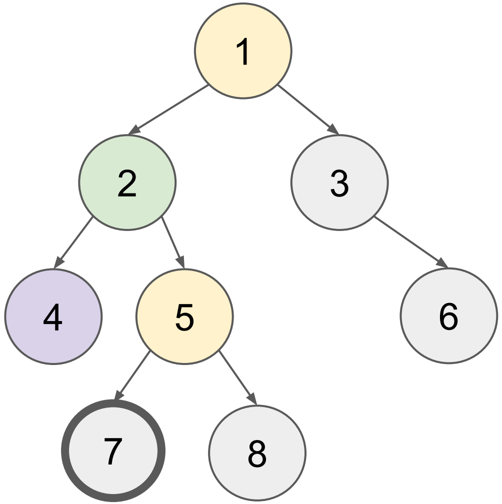
4							




```
void postorder_traversal(int node) {
    if (left_child[node])
        postorder_traversal(left_child[node]);

    if (right_child[node])
        postorder_traversal(right_child[node]);

    process(node);
}
```

Post-order

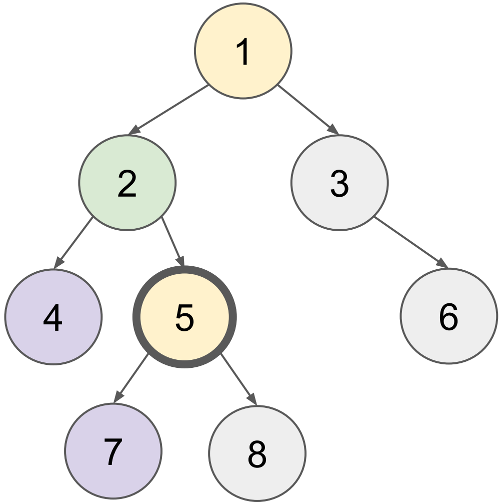
4	7						




```
void postorder_traversal(int node) {
    if (left_child[node])
        postorder_traversal(left_child[node]);

    if (right_child[node])
        postorder_traversal(right_child[node]);

    process(node);
}
```

Post-order

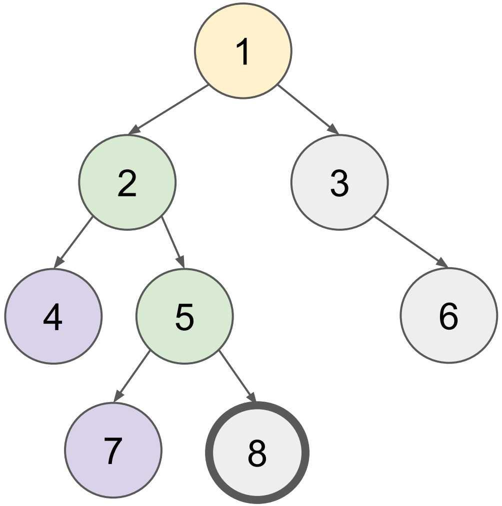
4	7						




```
void postorder_traversal(int node) {
    if (left_child[node])
        postorder_traversal(left_child[node]);

    if (right_child[node])
        postorder_traversal(right_child[node]);

    process(node);
}
```

Post-order

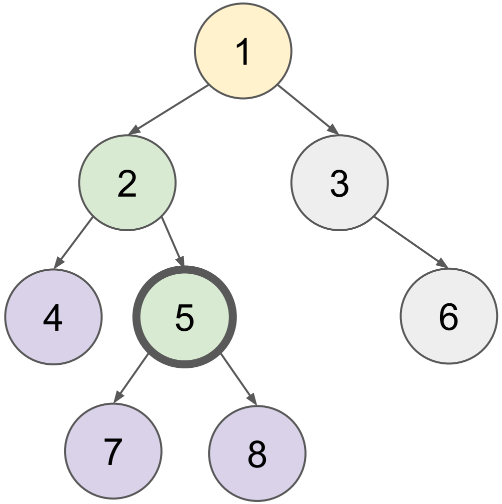
4	7	8					




```
void postorder_traversal(int node) {
    if (left_child[node])
        postorder_traversal(left_child[node]);

    if (right_child[node])
        postorder_traversal(right_child[node]);

    process(node);
}
```

Post-order

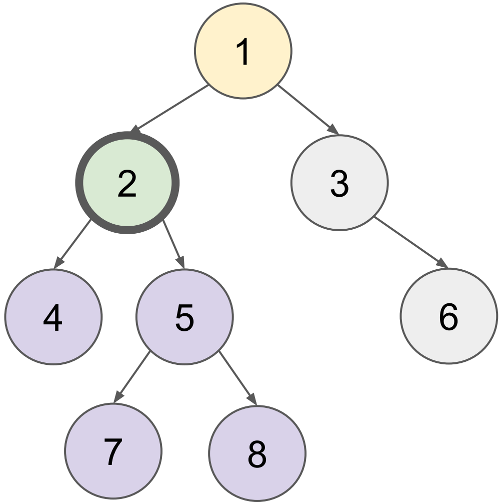
4	7	8	5				




```
void postorder_traversal(int node) {
    if (left_child[node])
        postorder_traversal(left_child[node]);

    if (right_child[node])
        postorder_traversal(right_child[node]);

    process(node);
}
```

Post-order

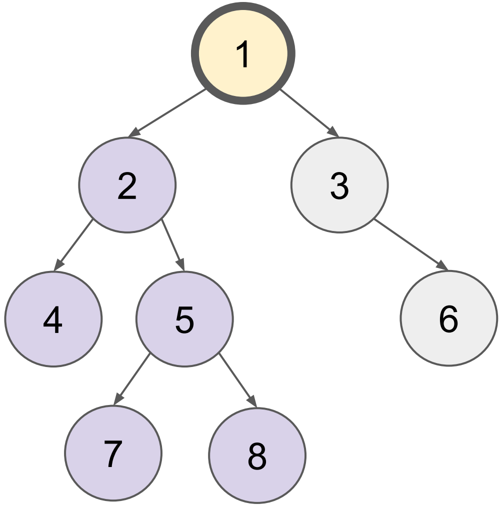
4	7	8	5	2			




```
void postorder_traversal(int node) {
    if (left_child[node])
        postorder_traversal(left_child[node]);

    if (right_child[node])
        postorder_traversal(right_child[node]);

    process(node);
}
```

Post-order

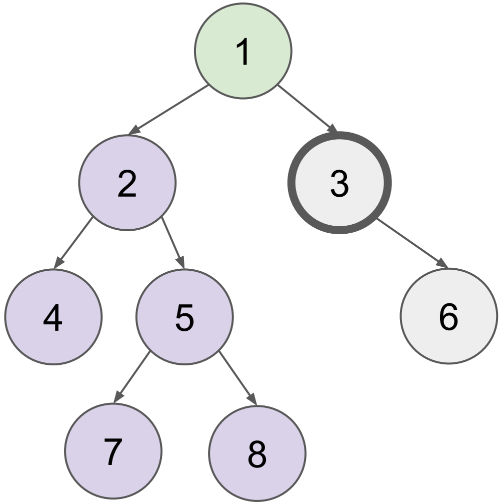
4	7	8	5	2			




```
void postorder_traversal(int node) {
    if (left_child[node])
        postorder_traversal(left_child[node]);

    if (right_child[node])
        postorder_traversal(right_child[node]);

    process(node);
}
```

Post-order

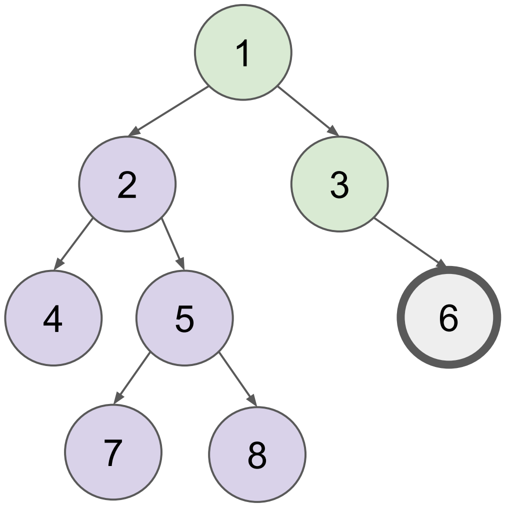
4	7	8	5	2			




```
void postorder_traversal(int node) {
    if (left_child[node])
        postorder_traversal(left_child[node]);

    if (right_child[node])
        postorder_traversal(right_child[node]);

    process(node);
}
```

Post-order

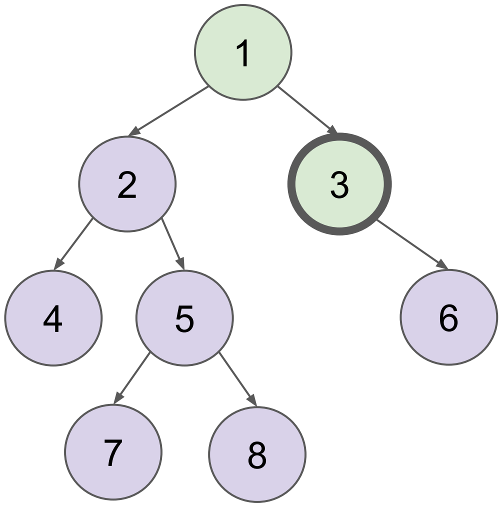
4	7	8	5	2	6		




```
void postorder_traversal(int node) {
    if (left_child[node])
        postorder_traversal(left_child[node]);

    if (right_child[node])
        postorder_traversal(right_child[node]);

    process(node);
}
```

Post-order

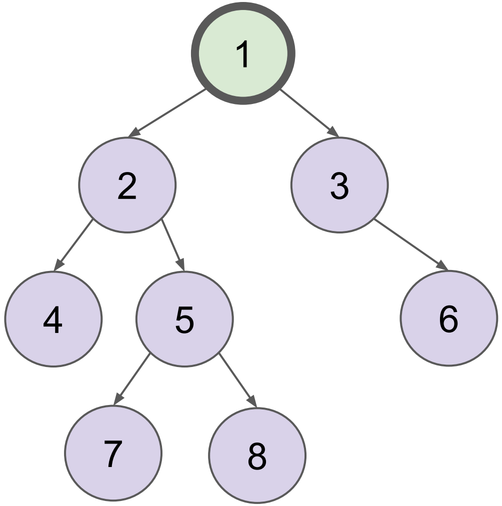
4	7	8	5	2	6	3	




```
void postorder_traversal(int node) {
    if (left_child[node])
        postorder_traversal(left_child[node]);

    if (right_child[node])
        postorder_traversal(right_child[node]);

    process(node);
}
```

Post-order

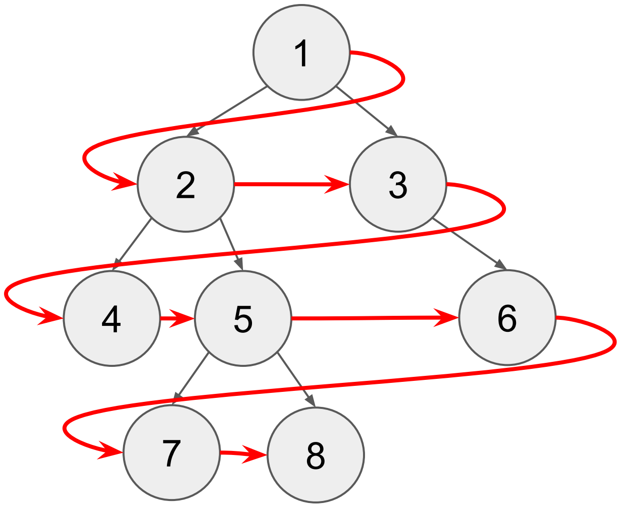
4	7	8	5	2	6	3	1




```
void postorder_traversal(int node) {
    if (left_child[node])
        postorder_traversal(left_child[node]);

    if (right_child[node])
        postorder_traversal(right_child[node]);

    process(node);
}
```

Breadth-first search on tree

1	2	3	4	5	6	7	8

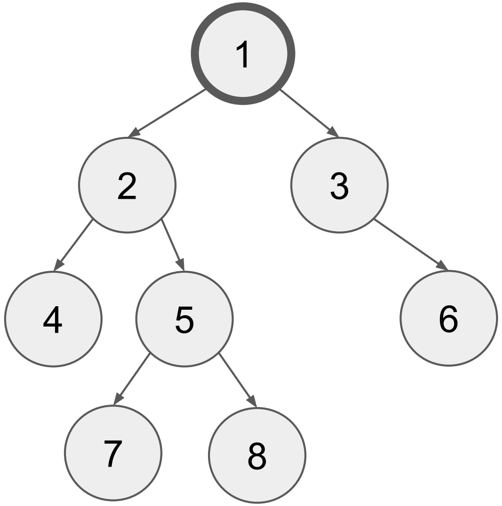



```
void bfs(int root) {
    queue<int> q;
    q.push(root);
    while (!q.empty()) {
        int node = q.front();
        q.pop();
        for (auto child : children[node])
            q.push(child);
    }
}
```

Pre-order, in-order and post-order: applications

A binary tree can be uniquely determined by (pre-order, in-order) or (post-order, in-order), **but not (pre-order, post-order)**.

[HKOJ 01040 Tree Recovery]



Pre-order	1	2	4	5	7	8	3	6
In-order	4	2	7	5	8	1	3	6
Post-order	4	7	8	5	2	6	3	1

Pre-order, in-order and post-order: applications

Problem 1:

Given a rooted tree with N nodes (node 1, 2, ..., N), find the size of all N subtrees with node *i* as root.

$1 \leq N \leq 10^5$

Pre-order, in-order and post-order: applications

Problem 1 Solution:

We can write a DFS similar to post-order traversal to calculate the size of subtree

size[i] = 1 + sum of *size[c]*

where *c* are all children of node *i*.

```
vector<int> size;
vector<vector<int>> children;

void dfs(int node) {
    size[node] = 1;
    for (auto child : children[node]) {
        dfs(child);
        size[node] += size[child];
    }
}
```

Pre-order, in-order and post-order: applications

Problem 2:

Given a rooted tree with N nodes (node 1, 2, ..., N) having initial value 0, there are two types of operation:

- *update(k, v)*: add *v* to every node in the subtree of node *k*
- *query(x)*: answer the value of node *x*.

Perform all *Q* operations.

$1 \leq N, Q \leq 10^5$

Pre-order, in-order and post-order: applications

Problem 2 Solution:

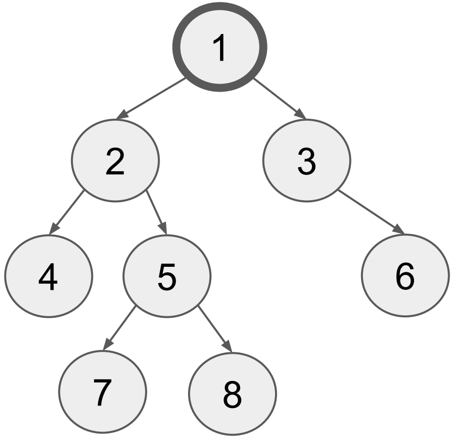
Store the value of nodes in an array using pre-order of the tree.
Then values of nodes in a subtree is contiguous in the array.

This reduces the problem to range update, point query problem, which can be solved with data structures like segment tree.

For example, the subtree of node 2 is shown below:

Index	0	**1**	2	3	4	5	6	7
Pre-order	1	**2**	4	5	7	8	3	6
Value	0	**0**	0	0	0	0	0	0

width = subtree size



Pre-order, in-order and post-order: applications

Problem 3: (Knapsack on Tree)

Given a rooted tree with N nodes (node 1, 2, ..., N), each of the edge has a digging cost and each of the node has a reward.

You can dig some edges that all digged edges are connected to node 1 by some digged edges. The sum of digging costs cannot exceed the digging budget B. Find the maximum sum of reward.

$1 \le N, B \le 10^3$

(from https://codeforces.com/blog/entry/13168)

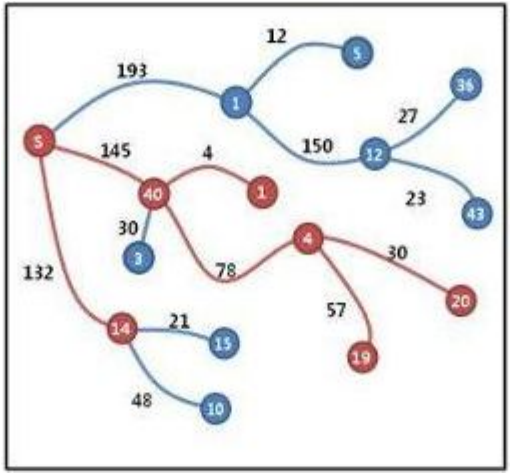
Pre-order, in-order and post-order: applications

Problem 3 Solution:

For this knapsack DP, we want to process the subtree of node i before node i.

In post-order, the subtree of node i is ensured to be in front of node i.

So we can use post-order as the DP order.



Pre-order, in-order and post-order: applications

Problem 3 Solution:

Let dp[i][j] = the maximum reward of same node that the post order is smaller then node i, the parent of the chosen node must be chose except the ancestors of i, with j budget.

Write a DFS , before traveling, copy dp[parent[i]] to dp[i].
After traveling, update dp[parent[i]] with every possible budget.

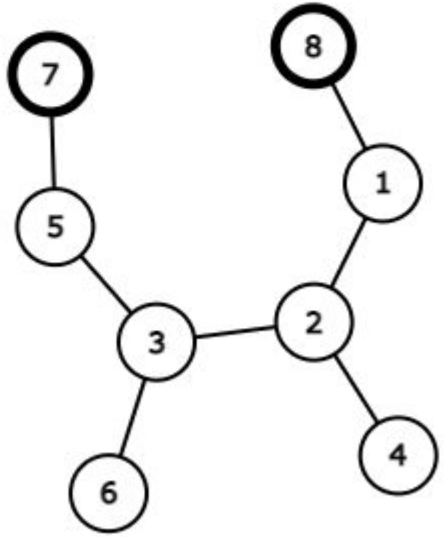
```
vector<vector<int>> children;
vector<vector<int>> dp;
vector<int> cost, reward;

void dfs(int node, int parent) {
    dp[node] = dp[parent];
    for (auto child : children[node]) {
        dfs(child, node);
    }

    for (int j=0; j+cost[node] <= B; j++)
    {
        dp[parent][j + cost[node]] = max(
        dp[parent][j + cost[node]],
        dp[node][j] + reward[node])
    }
}
```

Tree diameter

Tree diameter

The diameter of a tree is the furthest distance between any two nodes in the tree.

For example, in the tree on the right, the diameter of the tree is *5*.



Tree diameter: algorithm

It is obvious that we can run DFS/BFS from every node once and take the maximum distance from each run, resulting in an algorithm with complexity $O(V^2)$, where V is the number of nodes.

However, we can do better.

Tree diameter: algorithm

We do not have to run a DFS/BFS from every node: we only need to do it twice.

- First, we run a DFS/BFS from any node, recording the depth of each node from the starting node.
- Among all the nodes with the maximum depth, choose any of them, and run another DFS/BFS while recording the depth of each node.
- The maximum depth in the second DFS/BFS is the tree diameter.

This results in an algorithm with time complexity $O(V)$, where V is the number of nodes.

Tree diameter: implementation

```
void dfs(int u, int parent, int depth) {
    if (max_depth < depth) {
        deepest_node = u;
        max_depth = depth;
    }
    for (auto v : edges[u])
        if (v != parent) dfs(v, u, depth + 1);
}

int diameter() {
    max_depth = 0;
    dfs(1, -1, 0);
    max_depth = 0;
    dfs(deepest_node, -1, 0);
    return max_depth;
}
```

Tree diameter: proof

How does it work?

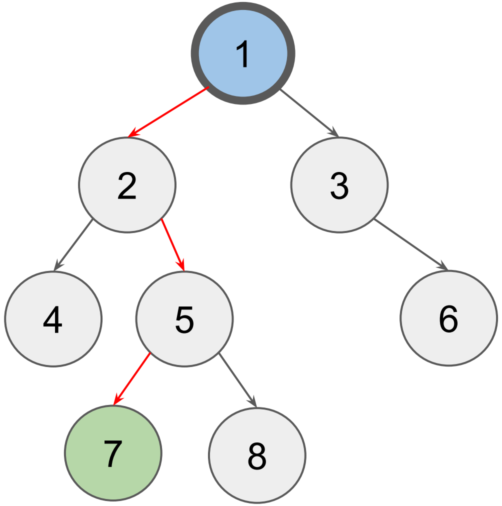
Proof by contradiction:

Assume the longest path is not from the deepest node from the root, you can always extend the path by changing one side to the deepest node

Tree diameter: example 1

First, we run a DFS from node 1. (This can be from any node.)

First DFS								
Node	**1**	2	3	4	5	6	7	8
Depth	0	1	1	2	2	2	3	3



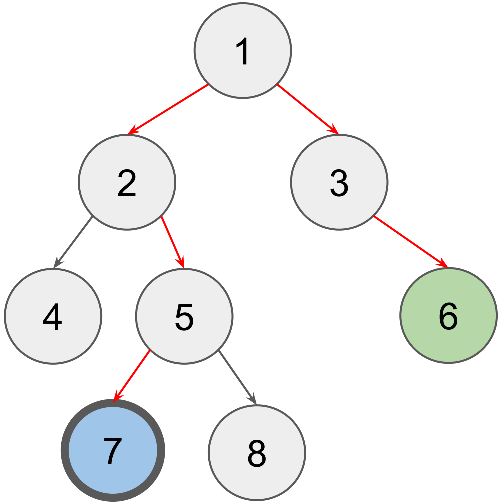
Tree diameter: example 1

As node 7 and 8 have the same, maximum depth, we can choose any of them.

Here, we start from node 7.

First DFS								
Node	**1**	2	3	4	5	6	7	8
Depth	0	1	1	2	2	2	3	3

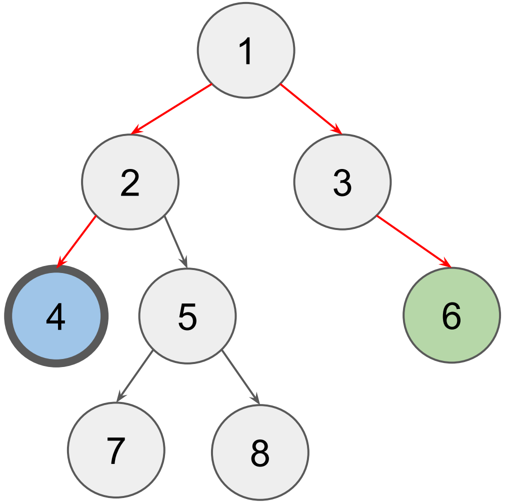
Second DFS								
Node	1	2	3	4	5	6	**7**	8
Depth	3	2	4	3	1	5	0	2



Tree diameter: example 2

First, we run a DFS from node 4. (This can be from any node.)

First DFS								
Node	**1**	2	3	4	5	6	7	8
Depth	2	1	3	0	2	4	3	3

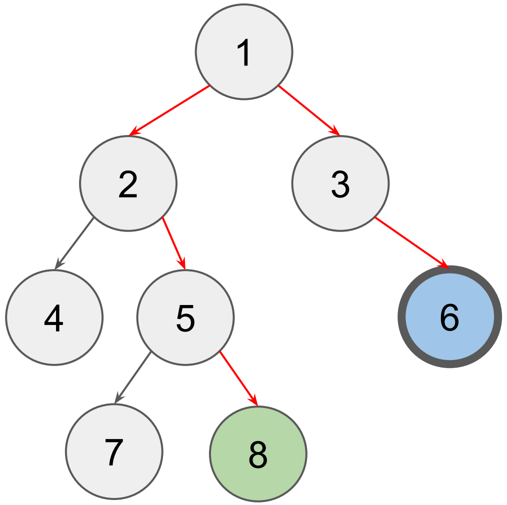


Tree diameter: example 2

As node 6 is the deepest node, we start from node 6.

First DFS								
Node	**1**	2	3	4	5	6	7	8
Depth	2	1	3	0	2	4	3	3

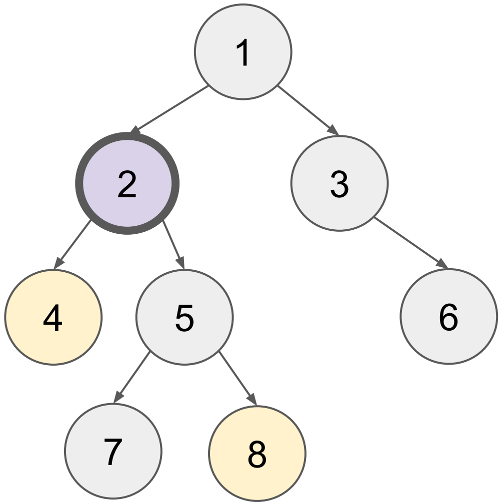
Second DFS								
Node	1	2	3	4	5	6	**7**	8
Depth	2	3	1	4	4	0	5	5



Lowest common ancestor (LCA)

Lowest common ancestor (LCA)

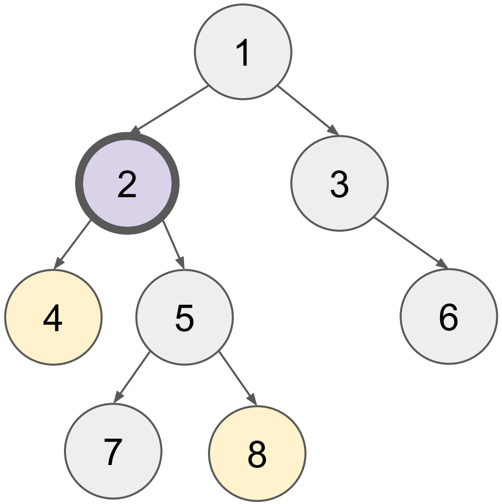
- In a rooted tree, the lowest common ancestor of two nodes *u* and *v* is the node that is the ancestor of both *u* and *v* and has the highest depth.
- If one of the nodes is the ancestor of another, it is the LCA.
- e.g. LCA(4, 8) = 2 (as shown on the right)
  LCA(5, 7) = 5



Lowest common ancestor (LCA)

Naive solution:

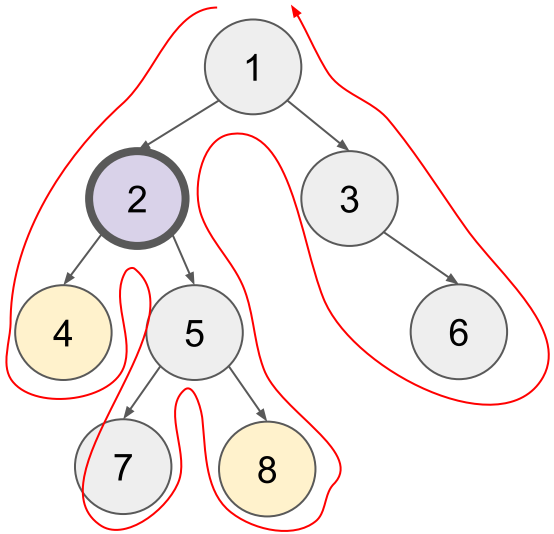
1. Check the ancestors of the given nodes and return the lowest common one
   - Time complexity per query: *O(N)*
2. Precompute the answers of all pairs by running DFS on each node
   - Time complexity per query: O(1)
   - Time complexity for precomputation: O(N^2)



Lowest common ancestor (LCA)

Solution 1:

- Perform DFS once to generate "Euler tour" of the tree:
  - Insert node once when first visiting the node
  - Insert node once when one of its children has been visited
  - Insert node once when leaving the node

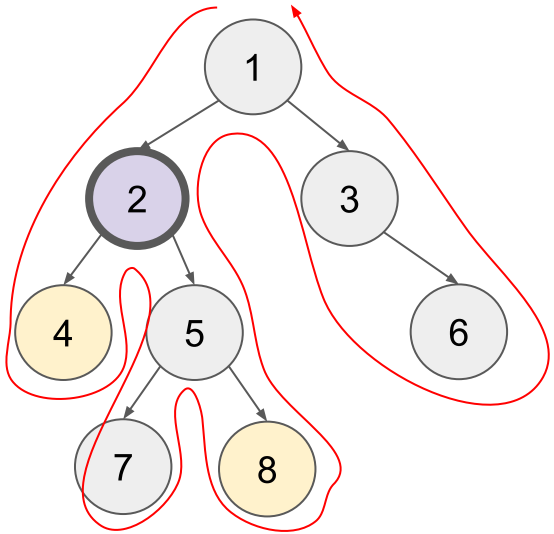


1	2	4	4	2	5	7	7	5	8	8	5	2	1	3	6	6	3	1

Lowest common ancestor (LCA)

Solution 1:

- For each node, compute the depth of the node.
- The LCA always appear between them in the euler tour.
- The one with the smallest depth is the LCA.



1	2	4	4	2	5	7	7	5	8	8	5	2	1	3	6	6	3	1
0	1	2	2	1	2	3	3	2	3	3	2	1	0	1	2	2	1	0

Lowest common ancestor (LCA)

Solution 1:

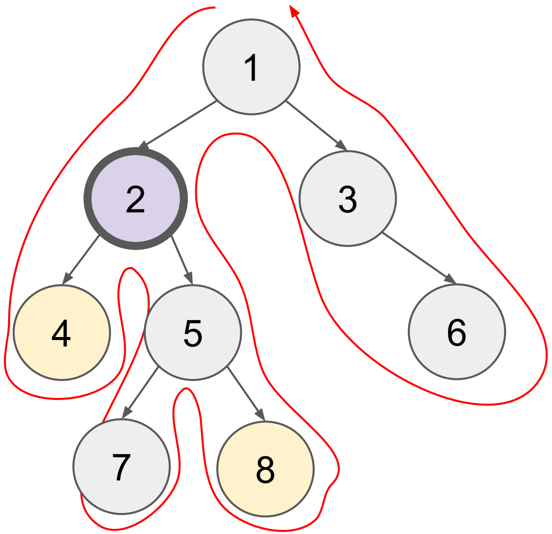
- Thus the problem is reduced to Range Minimum Query problem
- Can be solved with segment tree or **sparse table**
- Complexity:
    - Time, precomputation: *O(N)* for segment tree, *O(N log N)* for sparse table
    - Time, query: *O(log N)* for segment tree, *O(1)* for sparse table
    - Space: *O(N)* for segment tree, *O(N log N)* for sparse table

1	2	4	4	2	5	7	7	5	8	8	5	2	1	3	6	6	3	1
0	1	2	2	1	2	3	3	2	3	3	2	1	0	1	2	2	1	0

Lowest common ancestor (LCA)

Solution 1:

- For each node, compute the depth and **first occurrence** of the node.
- The LCA always appear between them in the euler tour.
- The one with the smallest depth is the LCA.

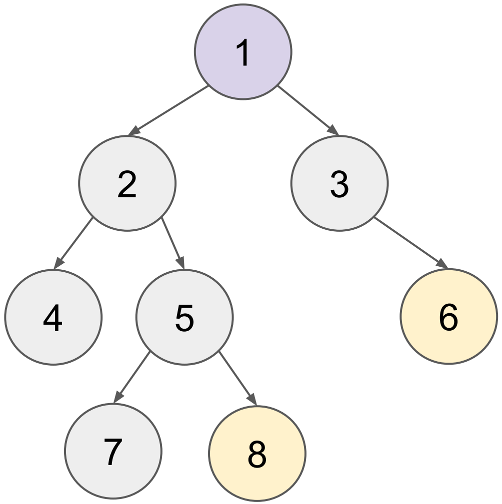


1	2	4	4	2	5	7	7	5	8	8	5	2	1	3	6	6	3	1
0	1	2	2	1	2	3	3	2	3	3	2	1	0	1	2	2	1	0

Lowest common ancestor (LCA)

Solution 2:

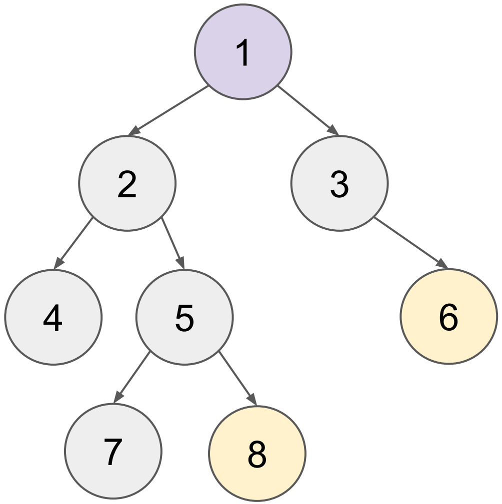
- We can attempt to optimize the naive solution of checking all ancestors with **binary lifting**.
- From now on, let us denote *ancestor(u, k)* as the *k*-th ancestor of node *u*.
- For example, *ancestor(8, 1) = 5*, *ancestor(5, 2) = 1*, *ancestor(2, 3) = 1**
- (For simplicity, we assume the parent of the root node to be itself.)



Lowest common ancestor (LCA)

Solution 2:

- First we precompute a table of *ancestor(u, 2^i)* for i = 0, 1, 2, …, $\lfloor \log_2 V \rfloor$
  - *ancestor(u, 2^0 = 1)* = the parent of *u*
  - *ancestor(u, 2^{i+1}) = ancestor(ancestor(u, 2^i), 2^i)*
- We can compute the table in *O(N log N)*.

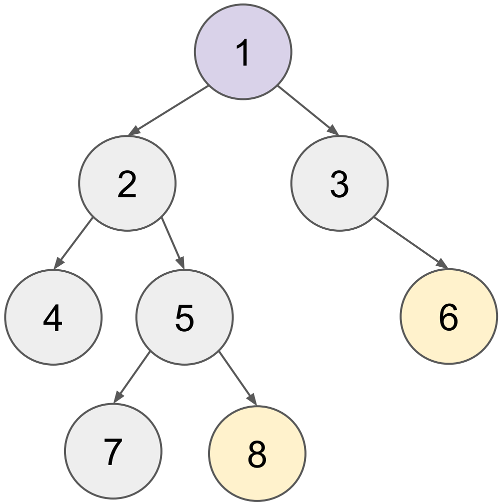


Node	1	2	3	4	5	6	7	8
i = 0	1	1	1	2	2	3	5	5
i = 1	1	1	1	1	1	1	2	2
i = 2	1	1	1	1	1	1	1	1
i = 3	1	1	1	1	1	1	1	1

Lowest common ancestor (LCA)

Solution 2:

- With the table precomputed, we can find *ancestor(u, x)* in *O(log N)*.
- For example,
  $ancestor(u, 11_{(10)} = 1011_{(2)})$
  $= ancestor(ancestor(ancestor(u, 2^3), 2^1), 2^0)$

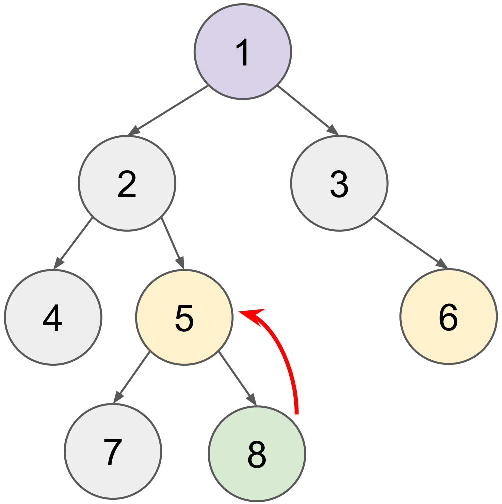


Node	1	2	3	4	5	6	7	8
i = 0	1	1	1	2	2	3	5	5
i = 1	1	1	1	1	1	1	2	2
i = 2	1	1	1	1	1	1	1	1
i = 3	1	1	1	1	1	1	1	1

Lowest common ancestor (LCA)

Solution 2:

- We will call the nodes *u*, *v* and assume *depth[u] ≥ depth[v]*.
- First, move *u* to the same level as *v* by *u ← ancestor(u, depth[v] - depth[u])*
- Then, we can use binary lifting to move *u* and *v* to be **one level below** the LCA of *u* and *v*.



Node	1	2	3	4	5	6	7	8
i = 0	1	1	1	2	2	3	5	5
i = 1	1	1	1	1	1	1	2	2
i = 2	1	1	1	1	1	1	1	1
i = 3	1	1	1	1	1	1	1	1

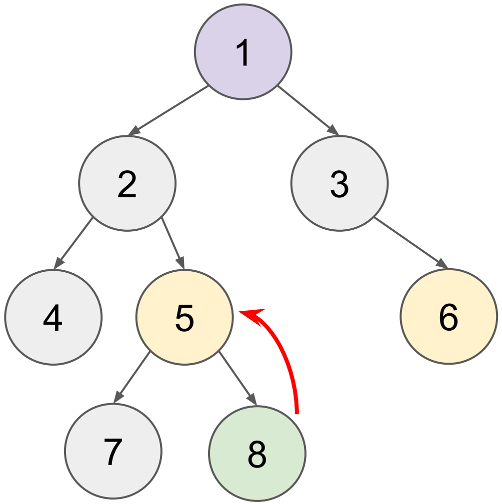
Lowest common ancestor (LCA)

Solution 2:

- The algorithm is as follows:

```
for i = ⌊log₂V⌋, ⌊log₂V⌋ - 1, …, 1, 0:
  if ancestor(u, 2^i) ≠ ancestor(v, 2^i):
    u ← ancestor(u, 2^i)
    v ← ancestor(v, 2^i)
```

- After this, the parent of *u* = the parent of *v* is the LCA of *u* and *v*.

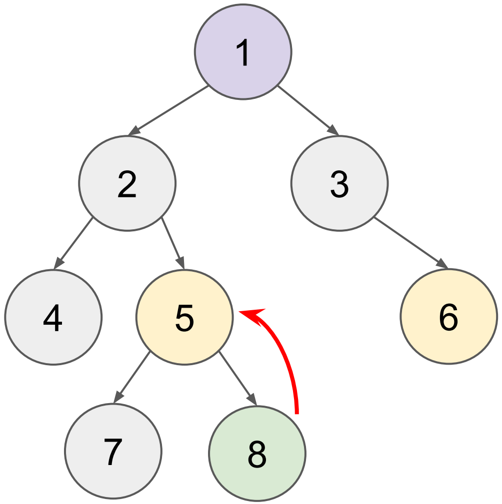


Node	1	2	3	4	5	6	7	8
i = 0	1	1	1	2	2	3	5	5
i = 1	1	1	1	1	1	1	2	2
i = 2	1	1	1	1	1	1	1	1
i = 3	1	1	1	1	1	1	1	1

Lowest common ancestor (LCA)

Solution 2:

- For example, when $u = 5$ and $v = 6$,
- $ancestor(u, 2^3) = 1$, $ancestor(v, 2^3) = 1 \Rightarrow$ no change
- $ancestor(u, 2^2) = 1$, $ancestor(v, 2^2) = 1 \Rightarrow$ no change
- $ancestor(u, 2^1) = 1$, $ancestor(v, 2^1) = 1 \Rightarrow$ no change
- $ancestor(u, 2^0) = 2$, $ancestor(v, 2^0) = 3$
  $\Rightarrow u \leftarrow ancestor(u, 2^0)$, $v \leftarrow ancestor(u, 2^0)$
- Now $u = 2$, $v = 3$ and $parent(u) = parent(v) = 1$, which is the LCA of 5 and 6.



Node	1	2	3	4	5	6	7	8
i = 0	1	1	1	2	2	3	5	5
i = 1	1	1	1	1	1	1	2	2
i = 2	1	1	1	1	1	1	1	1
i = 3	1	1	1	1	1	1	1	1

Lowest common ancestor (LCA)

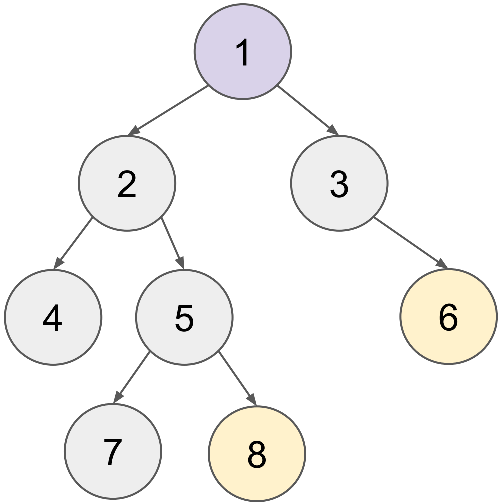
Solution 2: proof for correctness

- Assume the LCA of u and v is $ancestor(u, k)$, and $depth[u] = depth[v]$
  - i.e. we have already lifted u to the same depth as v
- $ancestor(u, a + b) = ancestor(ancestor(u, a), b)$ for all node u, $a \geq 0$, $b \geq 0$
  - $ancestor(u, 0) = u$
- For all $i < k$, $ancestor(u, i) \neq ancestor(v, i)$ because of contradiction (**lowest** common ancestor)
- For all $i \geq k$, $ancestor(u, i) = ancestor(ancestor(u, k), i - k)$
  $ancestor(v, i) = ancestor(ancestor(v, k), i - k)$
  As $ancestor(u, k) = ancestor(v, k)$ and $i - k \geq 0$, $ancestor(u, i) = ancestor(v, i)$
- i.e. the function $f(x) = \mathit{if}\ ancestor(u, x) = ancestor(v, x)\ \mathit{then}\ 1\ \mathit{else}\ 0$ is increasing or, in other words, you can binary search on f(x)

Lowest common ancestor (LCA)

Solution 2:

- Time complexity:
  - Precomputation: *O(N log N)*
  - Query: *O(log N)*
- Space complexity: *O(N log N)*

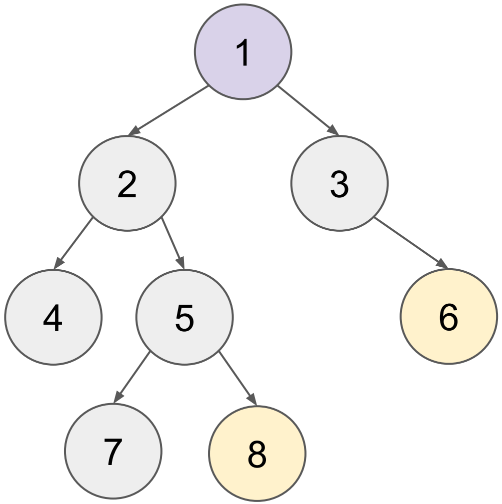


Node	1	2	3	4	5	6	7	8
i = 0	1	1	1	2	2	3	5	5
i = 1	1	1	1	1	1	1	2	2
i = 2	1	1	1	1	1	1	1	1
i = 3	1	1	1	1	1	1	1	1

Lowest common ancestor (LCA): application

Given a unrooted tree, answer distance between two given nodes.

e.g. *query(1, 2) = 1, query(5, 6) = 4*



Lowest common ancestor (LCA): application

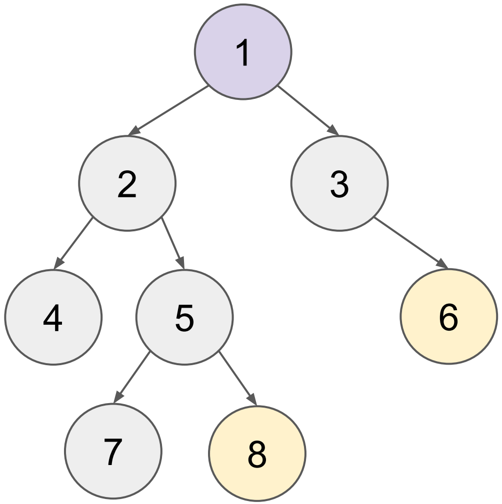
Answer:

Pick any node as the root. Then,

query(u, v)
= (depth[u] - depth[m]) + (depth[v] - depth[m])

where *m* is the LCA of *u* and *v*

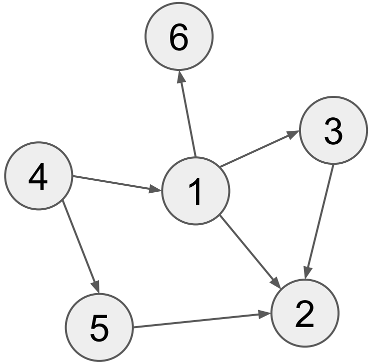
We can use any of the methods to compute LCA introduced just now.



DAG and Topological Sort

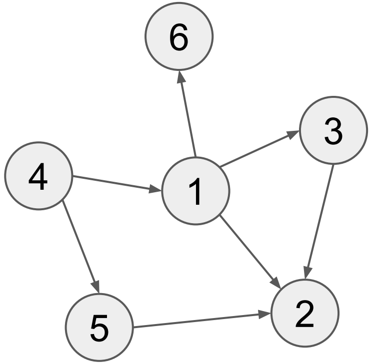
DAG and Topological Sort

- Directed Acyclic Graph (DAG) is a directed graph with no cycles.
- A rooted tree is also a DAG.



Topological Sort

- A topological ordering is an order of vertices in a graph where
  if there is an edge A → B, then A appears before B.
- For example, in the graph on the right, one of the topological orderings is
  [4, 5, 1, 6, 3, 2]
- A graph has a topological ordering
  if and only if the graph is a DAG.
- A graph can have more than one topological ordering.
  For example, [4, 1, 5, 3, 2, 6] is also a valid topological ordering for the graph.



Topological Sort

- To obtain a topological order, we can repeat the process of removing nodes with no incoming edges and all edges from that node.
- The code is shown on the right.
- We will demonstrate with samples below.

```
vector<pair<int, int>> edge_list;
vector<vector<int>> edges;
vector<int> topological_sort() {
    int n = edge_list.size();
    for (auto [u, v] : edge_list) in_drgree[v]++;
    queue<int> q;
    for (int i = 0; i < n; i++)
        if (in_drgree[i] == 0) q.push(i);
    vector<int> order;
    while (!q.empty()) {
        int u = q.front();
        q.pop();
        order.push_back(u);
        for (auto v : edges[u]) {
            in_drgree[v]--;
            if (in_drgree[v] == 0) q.push(v);
        }
    }
    return order;
}
```

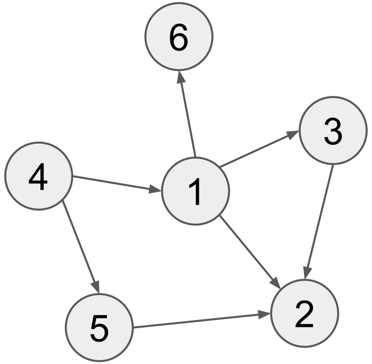
Topological Sort: Example 1

- First, the in-degree is calculated.

Node	1	2	3	4	5	6
In-degree	1	3	1	0	1	1

Queue						

Order						




```
vector<pair<int, int>> edge_list;
vector<vector<int>> edges;
vector<int> topological_sort() {
    int n = edge_list.size();
    for (auto [u, v] : edge_list) in_drgree[v]++;
    queue<int> q;
    for (int i = 0; i < n; i++)
        if (in_drgree[i] == 0) q.push(i);
    vector<int> order;
    while (!q.empty()) {
        int u = q.front();
        q.pop();
        order.push_back(u);
        for (auto v : edges[u]) {
            in_drgree[v]--;
            if (in_drgree[v] == 0) q.push(v);
        }
    }
    return order;
}
```

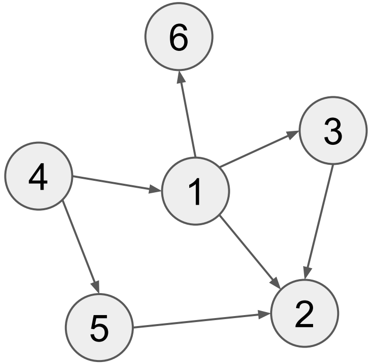
Topological Sort: Example 1

- Then, the node(s) with no incoming edges are pushed into the queue.

Node	1	2	3	4	5	6
In-degree	1	3	1	0	1	1

Queue	4					

Order						




```
vector<pair<int, int>> edge_list;
vector<vector<int>> edges;
vector<int> topological_sort() {
    int n = edge_list.size();
    for (auto [u, v] : edge_list) in_drgree[v]++;
    queue<int> q;
    for (int i = 0; i < n; i++)
        if (in_drgree[i] == 0) q.push(i);
    vector<int> order;
    while (!q.empty()) {
        int u = q.front();
        q.pop();
        order.push_back(u);
        for (auto v : edges[u]) {
            in_drgree[v]--;
            if (in_drgree[v] == 0) q.push(v);
        }
    }
    return order;
}
```

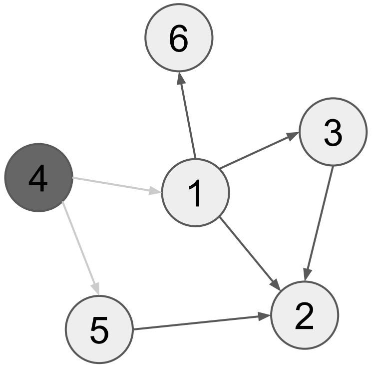
Topological Sort: Example 1

- Then, we repeat the process of removing nodes in the queue and pushing new nodes with no incoming edges into the queue.

Node	1	2	3	4	5	6
In-degree	0	3	1	0	0	1

Queue	~~4~~	1	5			

Order	4					




```
vector<pair<int, int>> edge_list;
vector<vector<int>> edges;
vector<int> topological_sort() {
    int n = edge_list.size();
    for (auto [u, v] : edge_list) in_drgree[v]++;
    queue<int> q;
    for (int i = 0; i < n; i++)
        if (in_drgree[i] == 0) q.push(i);
    vector<int> order;
    while (!q.empty()) {
        int u = q.front();
        q.pop();
        order.push_back(u);
        for (auto v : edges[u]) {
            in_drgree[v]--;
            if (in_drgree[v] == 0) q.push(v);
        }
    }
    return order;
}
```

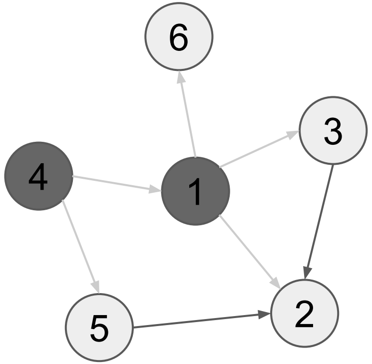
Topological Sort: Example 1

- Then, we repeat the process of removing nodes in the queue and pushing new nodes with no incoming edges into the queue.

Node	1	2	3	4	5	6
In-degree	0	2	0	0	0	0

Queue	~~4~~	~~1~~	5	3	6	

Order	4	1				




```
vector<pair<int, int>> edge_list;
vector<vector<int>> edges;
vector<int> topological_sort() {
    int n = edge_list.size();
    for (auto [u, v] : edge_list) in_drgree[v]++;
    queue<int> q;
    for (int i = 0; i < n; i++)
        if (in_drgree[i] == 0) q.push(i);
    vector<int> order;
    while (!q.empty()) {
        int u = q.front();
        q.pop();
        order.push_back(u);
        for (auto v : edges[u]) {
            in_drgree[v]--;
            if (in_drgree[v] == 0) q.push(v);
        }
    }
    return order;
}
```

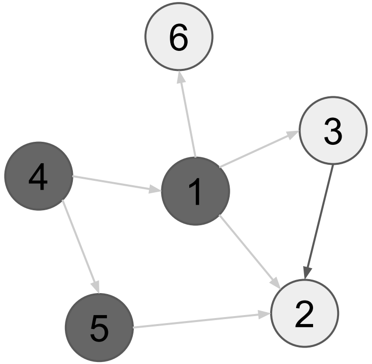
Topological Sort: Example 1

- Then, we repeat the process of removing nodes in the queue and pushing new nodes with no incoming edges into the queue.

Node	1	2	3	4	5	6
In-degree	0	1	0	0	0	0

Queue	~~4~~	~~1~~	~~5~~	3	6	

Order	4	1	5			




```
vector<pair<int, int>> edge_list;
vector<vector<int>> edges;
vector<int> topological_sort() {
    int n = edge_list.size();
    for (auto [u, v] : edge_list) in_drgree[v]++;
    queue<int> q;
    for (int i = 0; i < n; i++)
        if (in_drgree[i] == 0) q.push(i);
    vector<int> order;
    while (!q.empty()) {
        int u = q.front();
        q.pop();
        order.push_back(u);
        for (auto v : edges[u]) {
            in_drgree[v]--;
            if (in_drgree[v] == 0) q.push(v);
        }
    }
    return order;
}
```

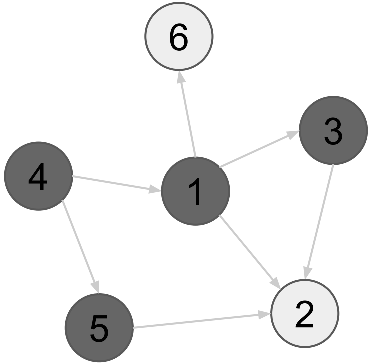
Topological Sort: Example 1

- Then, we repeat the process of removing nodes in the queue and pushing new nodes with no incoming edges into the queue.

Node	1	2	3	4	5	6
In-degree	0	0	0	0	0	0

Queue	~~4~~	~~1~~	~~5~~	~~3~~	6	2

Order	4	1	5	3		



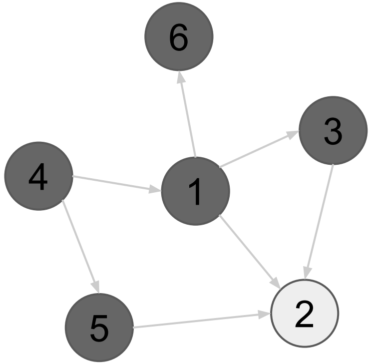

```cpp
vector<pair<int, int>> edge_list;
vector<vector<int>> edges;
vector<int> topological_sort() {
    int n = edge_list.size();
    for (auto [u, v] : edge_list) in_drgree[v]++;
    queue<int> q;
    for (int i = 0; i < n; i++)
        if (in_drgree[i] == 0) q.push(i);
    vector<int> order;
    while (!q.empty()) {
        int u = q.front();
        q.pop();
        order.push_back(u);
        for (auto v : edges[u]) {
            in_drgree[v]--;
            if (in_drgree[v] == 0) q.push(v);
        }
    }
    return order;
}
```

Topological Sort: Example 1

- Then, we repeat the process of removing nodes in the queue and pushing new nodes with no incoming edges into the queue.

Node	1	2	3	4	5	6
In-degree	0	0	0	0	0	0

Queue	~~4~~	~~1~~	~~5~~	~~3~~	~~6~~	2
Order	4	1	5	3	6	




```
vector<pair<int, int>> edge_list;
vector<vector<int>> edges;
vector<int> topological_sort() {
    int n = edge_list.size();
    for (auto [u, v] : edge_list) in_drgree[v]++;
    queue<int> q;
    for (int i = 0; i < n; i++)
        if (in_drgree[i] == 0) q.push(i);
    vector<int> order;
    while (!q.empty()) {
        int u = q.front();
        q.pop();
        order.push_back(u);
        for (auto v : edges[u]) {
            in_drgree[v]--;
            if (in_drgree[v] == 0) q.push(v);
        }
    }
    return order;
}
```

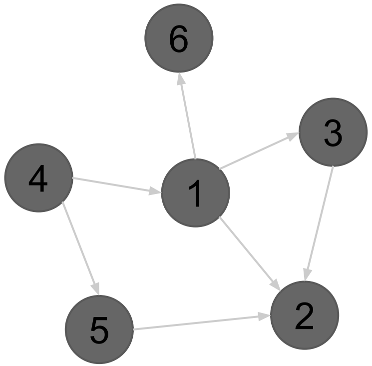
Topological Sort: Example 1

- The process is finished when the queue is empty.

Node	1	2	3	4	5	6
In-degree	0	0	0	0	0	0

Queue	~~4~~	~~1~~	~~5~~	~~3~~	~~6~~	~~2~~

Order	4	1	5	3	6	2




```
vector<pair<int, int>> edge_list;
vector<vector<int>> edges;
vector<int> topological_sort() {
    int n = edge_list.size();
    for (auto [u, v] : edge_list) in_drgree[v]++;
    queue<int> q;
    for (int i = 0; i < n; i++)
        if (in_drgree[i] == 0) q.push(i);
    vector<int> order;
    while (!q.empty()) {
        int u = q.front();
        q.pop();
        order.push_back(u);
        for (auto v : edges[u]) {
            in_drgree[v]--;
            if (in_drgree[v] == 0) q.push(v);
        }
    }
    return order;
}
```

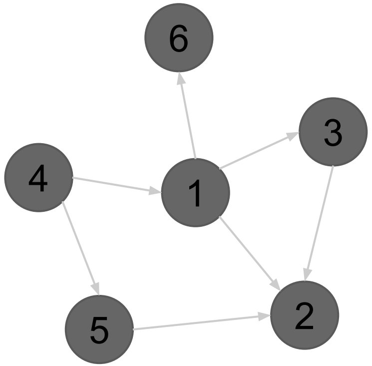
Topological Sort: Example 1

- The process is finished when the queue is empty.

Node	1	2	3	4	5	6
In-degree	0	0	0	0	0	0

Queue	~~4~~	~~1~~	~~5~~	~~3~~	~~6~~	~~2~~

Order	4	1	5	3	6	2




```
vector<pair<int, int>> edge_list;
vector<vector<int>> edges;
vector<int> topological_sort() {
    int n = edge_list.size();
    for (auto [u, v] : edge_list) in_drgree[v]++;
    queue<int> q;
    for (int i = 0; i < n; i++)
        if (in_drgree[i] == 0) q.push(i);
    vector<int> order;
    while (!q.empty()) {
        int u = q.front();
        q.pop();
        order.push_back(u);
        for (auto v : edges[u]) {
            in_drgree[v]--;
            if (in_drgree[v] == 0) q.push(v);
        }
    }
    return order;
}
```

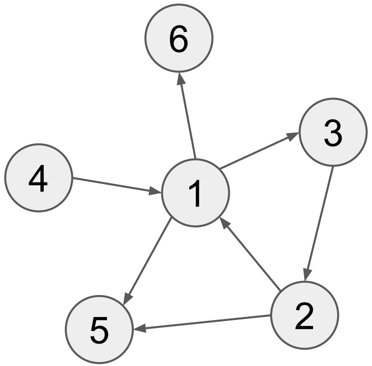
Topological Sort: Example 2

- First, the in-degree is calculated.

Node	1	2	3	4	5	6
In-degree	**1**	**1**	**1**	**0**	**2**	**1**

Queue	4					

Order						




```
vector<pair<int, int>> edge_list;
vector<vector<int>> edges;
vector<int> topological_sort() {
    int n = edge_list.size();
    for (auto [u, v] : edge_list) in_drgree[v]++;
    queue<int> q;
    for (int i = 0; i < n; i++)
        if (in_drgree[i] == 0) q.push(i);
    vector<int> order;
    while (!q.empty()) {
        int u = q.front();
        q.pop();
        order.push_back(u);
        for (auto v : edges[u]) {
            in_drgree[v]--;
            if (in_drgree[v] == 0) q.push(v);
        }
    }
    return order;
}
```

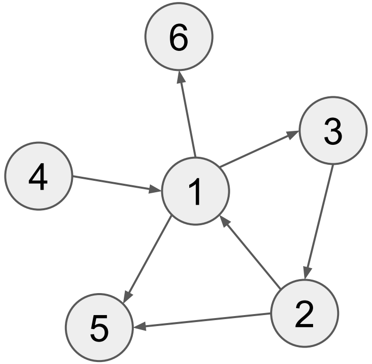
Topological Sort: Example 2

- Then, the node(s) with no incoming edges are pushed into the queue.

Node	1	2	3	4	5	6
In-degree	1	1	1	0	2	1

Queue	4					

Order						




```
vector<pair<int, int>> edge_list;
vector<vector<int>> edges;
vector<int> topological_sort() {
    int n = edge_list.size();
    for (auto [u, v] : edge_list) in_drgree[v]++;
    queue<int> q;
    for (int i = 0; i < n; i++)
        if (in_drgree[i] == 0) q.push(i);
    vector<int> order;
    while (!q.empty()) {
        int u = q.front();
        q.pop();
        order.push_back(u);
        for (auto v : edges[u]) {
            in_drgree[v]--;
            if (in_drgree[v] == 0) q.push(v);
        }
    }
    return order;
}
```

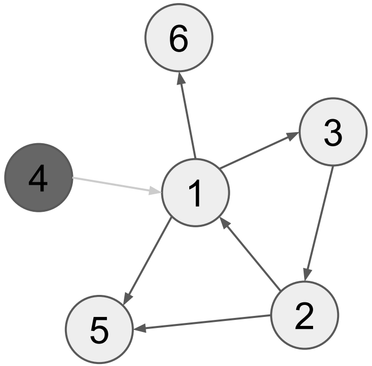
Topological Sort: Example 2

- Then, we repeat the process of removing nodes in the queue and pushing new nodes with no incoming edges into the queue.

Node	1	2	3	4	5	6
In-degree	1	1	1	0	2	1

Queue	4					

Order	4					




```
vector<pair<int, int>> edge_list;
vector<vector<int>> edges;
vector<int> topological_sort() {
    int n = edge_list.size();
    for (auto [u, v] : edge_list) in_drgree[v]++;
    queue<int> q;
    for (int i = 0; i < n; i++)
        if (in_drgree[i] == 0) q.push(i);
    vector<int> order;
    while (!q.empty()) {
        int u = q.front();
        q.pop();
        order.push_back(u);
        for (auto v : edges[u]) {
            in_drgree[v]--;
            if (in_drgree[v] == 0) q.push(v);
        }
    }
    return order;
}
```

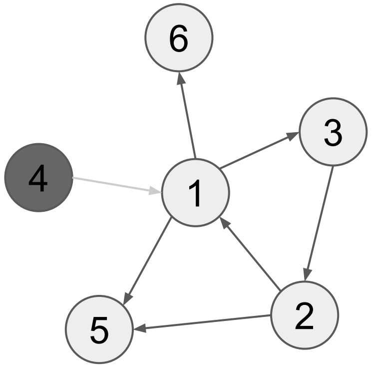
Topological Sort: Example 2

- The queue is empty, but we have not processed the whole graph. This means the graph has a **cycle**.

Node	1	2	3	4	5	6
In-degree	1	1	1	0	2	1

Queue	4					

Order	4					




```
vector<pair<int, int>> edge_list;
vector<vector<int>> edges;
vector<int> topological_sort() {
    int n = edge_list.size();
    for (auto [u, v] : edge_list) in_drgree[v]++;
    queue<int> q;
    for (int i = 0; i < n; i++)
        if (in_drgree[i] == 0) q.push(i);
    vector<int> order;
    while (!q.empty()) {
        int u = q.front();
        q.pop();
        order.push_back(u);
        for (auto v : edges[u]) {
            in_drgree[v]--;
            if (in_drgree[v] == 0) q.push(v);
        }
    }
    return order;
}
```

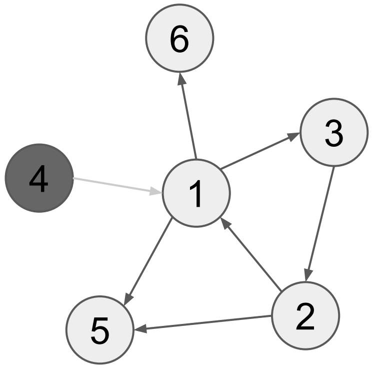
Topological Sort: Example 2

We can use topological sort to **detect cycles** in a directed graph!

Node	1	2	3	4	5	6
In-degree	1	1	1	0	2	1

Queue	4					

Order	4					




```
vector<pair<int, int>> edge_list;
vector<vector<int>> edges;
vector<int> topological_sort() {
    int n = edge_list.size();
    for (auto [u, v] : edge_list) in_drgree[v]++;
    queue<int> q;
    for (int i = 0; i < n; i++)
        if (in_drgree[i] == 0) q.push(i);
    vector<int> order;
    while (!q.empty()) {
        int u = q.front();
        q.pop();
        order.push_back(u);
        for (auto v : edges[u]) {
            in_drgree[v]--;
            if (in_drgree[v] == 0) q.push(v);
        }
    }
    return order;
}
```

Practice Problems

- HKOJ 01038 - Preorder Tree Traversal
- HKOJ 01039 - Postorder Tree Traversal
- HKOJ S042 - Teacher's Problem
- HKOJ M0642 - Cells
- HKOJ T114 - Current Flow
- HKOJ T172 - City Reform
- Codeforces 191C - Fools and Roads
- Codeforces 208E - Blood Cousins
- AtCoder nikkei2019_qual_d - Restore the Tree
- AtCoder past201912_k - Conglomerate

Reference

https://assets.hkoi.org/training2019/g-iii.pdf

https://assets.hkoi.org/training2021/g-iii.pdf

https://assets.hkoi.org/training2022/g-iii.pdf

https://codeforces.com/blog/entry/13168