Final exam

- Next week class will be final exam
  - Same format as midterm, 3 regular (100 pt each) + 2 bonus (10 pt each)
  - Your best-performed 3 problems will be regular
  - Do not open test page unless you are ready for the test. Two Hours Timer will start immediately
- Bonus points for 5 minute video competition
  - Participation: + 10 points to final exam
  - Winner: +20 points
- How to prepare for final exam
  - Review/redo homework

Homework Review

The Great Revegetation - P15373

- Build constraint graph
- Undirected graph
- Each node (pasture) has 4 options (1, 2, 3, 4)
- Start from first node
- Remove options if there is a constraint for it
- Take the smallest in the left options

SAMPLE INPUT:

```
5 6
4 1
4 2
4 3
2 5
1 2
1 5
```

SAMPLE OUTPUT:

```
12133
```

The Great Revegetation - P15373

```
int main () {
  int n, m, p1, p2;
  cin >> n >> m;

  int type[n+1]; // grass type of pastures
  memset(type, 0, sizeof(type));

  vector<int> adj[n+1];
  for (int i=0; i<m; i++) {
    cin >> p1 >> p2;
    adj[p1].push_back(p2);
    adj[p2].push_back(p1);
  }

  for (int i=1; i<=n; i++) {
    set<int> usable_types = {1, 2, 3, 4};
    // go through all adj pastures, if they already
    // have a grass type, remove from usable type
    // default is 0, removing 0 has no impact
    for (auto x : adj[i])
      usable_types.erase(type[x]);
    type[i] = *usable_types.begin();
    cout << type[i] << endl;
  }
}
```

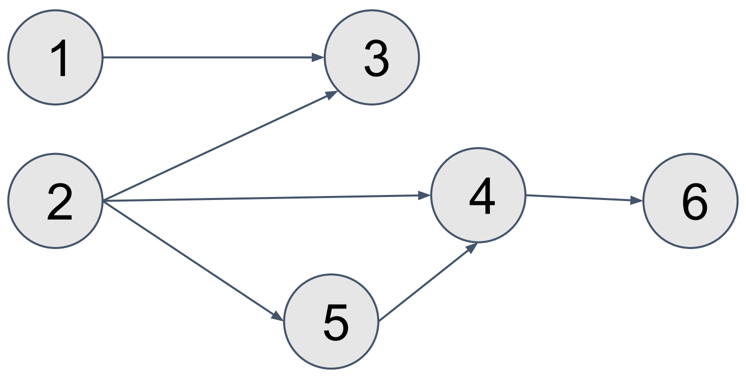
SAMPLE INPUT:

5 6
4 1
4 2
4 3
2 5
1 2
1 5

SAMPLE OUTPUT:

12133

Topological sequence (P1064)



1. Use the algorithm we discussed in class
2. Smallest topology sequence
3. Can we use queue?
4. If not, what data structure should we use?

Sample input:

6 8

1 3

2 3

2 4

2 5

3 4

3 6

4 6

5 4

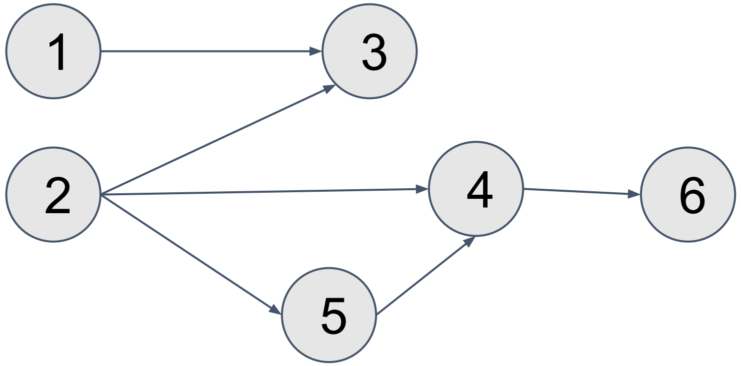
Sample output:

1 2 3 5 4 6

Topological sequence (P1064)

```
void cal_order(vector<vector<int>>& arr) {
  vector<int> in(arr.size(), 0);
  set<int> myset;
  vector<int> solution;
  int size = arr.size();

  // calc in-degree
  for (int i = 0; i < size; i++)
    for (int j = 0; j < arr[i].size(); j++)
      in[j] += arr[i][j];

  // add nodes with in-degree of 0
  for (int i = 0; i < size; i++)
    if (in[i] == 0)
      myset.insert(i); // add to set

  while (!myset.empty()) {
    auto top = myset.begin();
    int current = *top;
    myset.erase(top);
    solution.push_back(current);

    size = arr[current].size();
    for (int j = 0; j < size; j++) {
      if (arr[current][j] == 0)
        continue;
      in[j] -= arr[current][j]; // re-calc in-degree
      if (in[j] == 0)
        myset.insert(j); // add to set
    }
  }

  for (auto it = begin(solution); it != end(solution); ++it)
    cout << *it + 1 << " ";
}
```

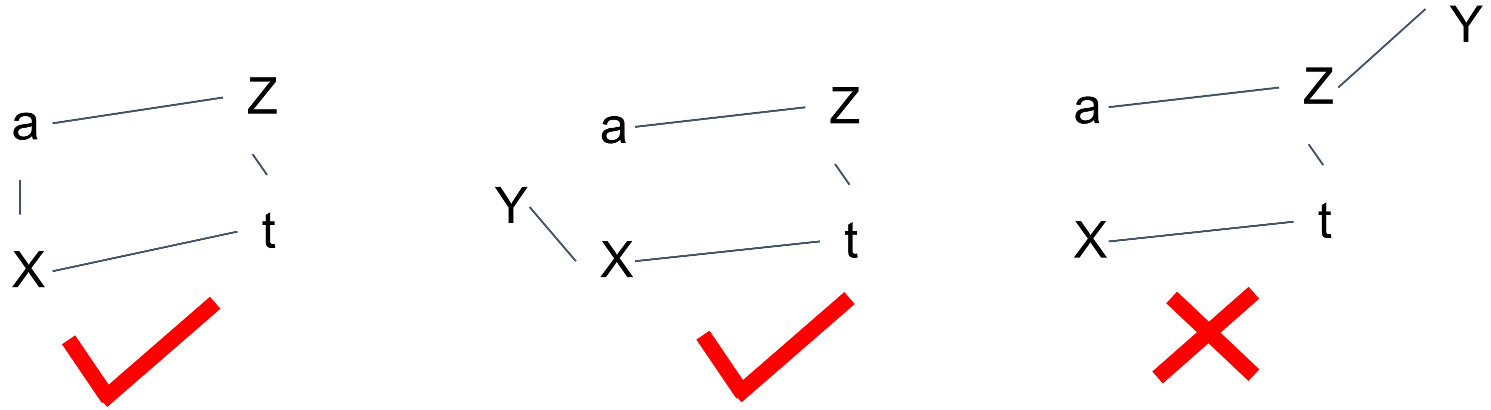
```
int main(void)
{
  int n, m, a, b;

  cin >> n >> m;
  vector<vector<int>> arr(n, vector<int>(n, 0));
  for (int i = 1; i <= m; i++) { // read matrix
    cin >> a >> b;
    arr[a - 1][b - 1] = 1;
  }
  cal_order(arr);
  return 0;
}
```



Sample output:

1 2 3 5 4 6

Unordered letter pairs (P8264)



1. Count node number with odd degree
2. If and only if number is 0 or 2, there is solution
3. If 0, select overall smallest node
4. If 2, select the smaller end
5. Start dfs, keep going until all connections are checked

Sample input 1:

4
aZ
tZ
Xt
aX

Sample output 2:

XaZtX

Unordered letter pairs (P8264)

```cpp
int char_counts[60];
bool ajacency_matrix[105][105];
int min_char = 58; // from 'A' to 'z'
string ans;

void dfs(int i) {
  ans += (char)(i + 'A');
  for (int j = 0; j < 58; j++) {
    if (ajacency_matrix[i][j]) {
      ajacency_matrix[i][j] = ajacency_matrix[j][i] = 0;
      dfs(j);
      return;
    }
  }
}

int main() {
  int n;
  cin >> n;
  for (int i = 0; i < n; i++) {
    char a, b;
    cin >> a >> b;
    int x = a - 'A';
    int y = b - 'A';
    ajacency_matrix[x][y] = ajacency_matrix[y][x] = 1;
    min_char = min(min_char, min(x, y));
    char_counts[x]++;
    char_counts[y]++;
  }
  // Number of odd chars
  // since we have n pairs and want n+1 length of string
  // num of odd must be 0 or 2
  int odd_count = 0;
  for (int i = 0; i < 58; i++) {
    if ((char_counts[i] % 2) != 0) {
      odd_count++;
      switch (odd_count) {
      case 1:
        min_char = i;
        break;
      case 2:
        min_char = min(min_char, i);
        break;
      default: // odd_count > 2
        cout << "No Solution";
        return 0;
      }
    }
  }

  if (odd_count == 0 || odd_count == 2) {
    dfs(min_char);
    cout << ans;
  } else {
    cout << "No Solution";
  }
}
```

Connected blocks (P8253)

```
int main() {
  int n, m;
  cin >> n >> m;
  visited = vector<int> (n+1,0);
  adjList = vector<vector<int>> (n+1);

  int components = 0;
  for(int i = 0; i < m; i++) {
    int n1, n2;
    cin >> n1 >> n2;
    adjList[n1].push_back(n2);
    adjList[n2].push_back(n1);
  }

  for(int i = 1; i < n+1; i++) {
    if(!visited[i]) {
      components++;
      dfs(i);
    }
  }
```

```
vector<int> visited;
vector<vector<int>> adjList;

void dfs(int i) {
  if(visited[i])
    return;

  visited[i] = 1;
  for(auto it = adjList[i].begin();
    it != adjList[i].end(); it++)
    dfs(*it);
}

```

Sample input:

5 6
1 2
3 4
5 2
5 1
1 5
4 4

Sample output:

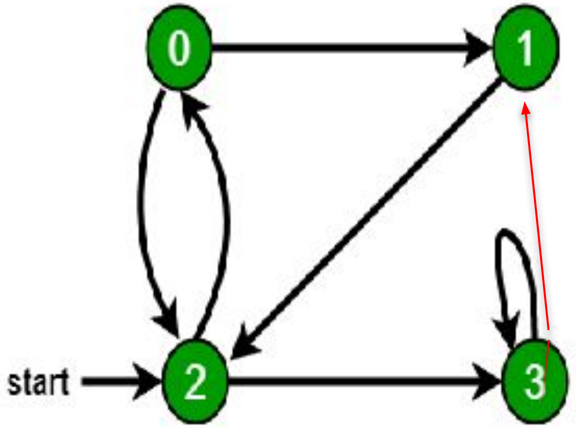
2

What did we learn in last week?

- Detect a cycle
- Topological sort
- Euler path/circuit

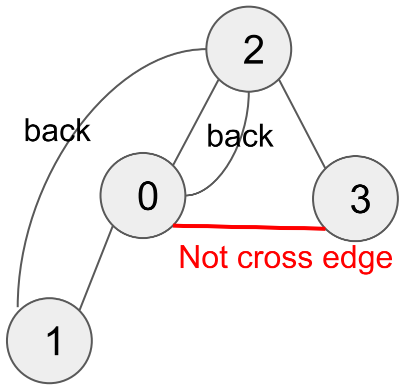
Detect Cycle in a Directed Graph

```
bool isCyclic(int v, bool visited[], bool recStack[])
{
        // Mark the current node as visited and part of recursion stack
        visited[v] = true;
        recStack[v] = true;
        list<int>::iterator i;
         // Recur for all the vertices adjacent to this vertex
        for(i = adj[v].begin(); i != adj[v].end(); ++i)
        {
            if ( !visited[*i] && isCyclic(*i, visited, recStack) )
                return true;
            else if (recStack[*i]) // can't just check visited, red arrow
                return true;
        }
        recStack[v] = false;  // remove the vertex from recursion stack, back tracking
        return false;
}
```



Detect Cycle in an Simple Undirected Graph

```
// A recursive function that uses visited[] and parent to detect
// cycle in subgraph reachable from vertex v.
bool isCyclic(int v, bool visited[], int parent)
{
   // Mark the current node as visited
   visited[v] = true;
    // Recur for all the vertices adjacent to this vertex
   list<int>::iterator i;
   for (i = adj[v].begin(); i != adj[v].end(); ++i)
   {
       // If an adjacent is not visited, then recur for that adjacent
       if (!visited[*i])
       {
          if (isCyclic(*i, visited, v))
             return true;
       }
        // If an adjacent is visited and not parent of current vertex,
       // then there is a cycle.
       else if (*i != parent)
          return true;
   }
   return false;
}
```



Topological Sorting Algorithm (BFS based)

Step-1: Compute in-degree (number of incoming edges) for each of the vertex present in the DAG and initialize the count of visited nodes as 0.

Step-2: Pick all the vertices with in-degree as 0 and add them into a queue (Enqueue operation)

Step-3: Remove a vertex from the queue (Dequeue operation) and then.

1. Increment count of visited nodes by 1.
2. Decrease in-degree by 1 for all its neighboring nodes.
3. If in-degree of a neighboring nodes is reduced to zero, then add it to the queue.

Step 4: Repeat Step 3 until the queue is empty.

If count of visited nodes is **not** equal to the number of nodes in the graph then the topological sort is not possible for the given graph. Why?

Definition for Euler circuit and Euler path

An **Euler circuit** is a **circuit** starts and ends at the same vertices, and uses every edge of a graph exactly once.

An **Euler path** starts and ends at different vertices.

Existence of Euler Circuit and Path

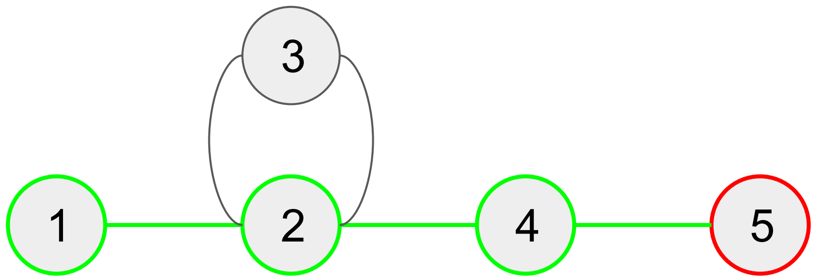
Existence of Euler Circuit: **Every** vertex has **even** degree.

Existence of Euler Path: Every vertex has even degree except two odd vertices. (They will be the start and end vertex)

Algorithm to Find Euler Path and Euler Circuit

For Euler Path, start at one of the odd vertex.

1. DFS, Keep following unused edges and removing them until we get stuck.
2. Once we get stuck, we backtrack to the nearest vertex in our current path that has unused edges, and we repeat the process until all the edges have been used. We can use another container to maintain the final path



Start from node 1 and get final Solution: [1, 2, 3, 2, 4, 5]

X - Camp

Week 11

Disjoint Set Union

Goal: To understand the need and use of collections of sets and its applications in relevant problems.

Introduction

What is a set?

Note this is not referring to std::set in C++. Rather, it is a mathematical concept.
Some examples for sets

What properties does a set have?

What makes two sets disjoint?

What is “disjoint”?

Definition

A **disjoint - set union (DSU)**, also called **union - find** or **merge - find**, is a data structure that operates with a set partitioned in several disjoint subsets.

★ **Find**: Given a particular element of the set, it identifies the subset of the element

★ **Unite**: Joins two subsets into a single subset

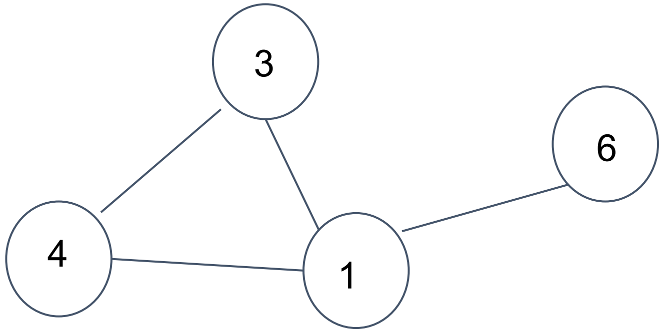
Note: DSU allows you to add edges to an initially empty graph and test whether two vertices of the graph are connected.

Application

Example: Epidemiology - Suppose groups of people are infected with the same disease, we can keep track of these connections and know which people are affected by the same transmission cluster.

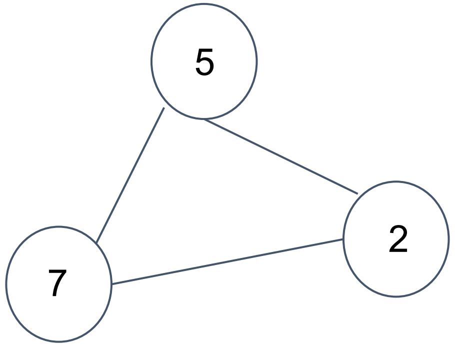
Let’s think of some more!

Connected Components



For each query to check friendship, check to see if they belong to the same connected component

Check:
1 and 3
5 and 7
2 and 6



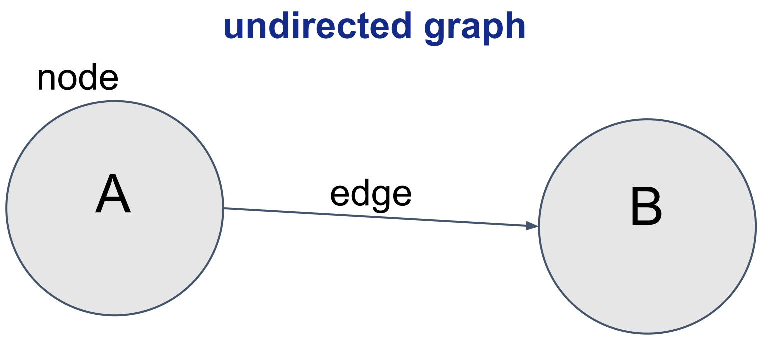
What algorithm do we know that we could use for find?
Are there any downsides to this method?

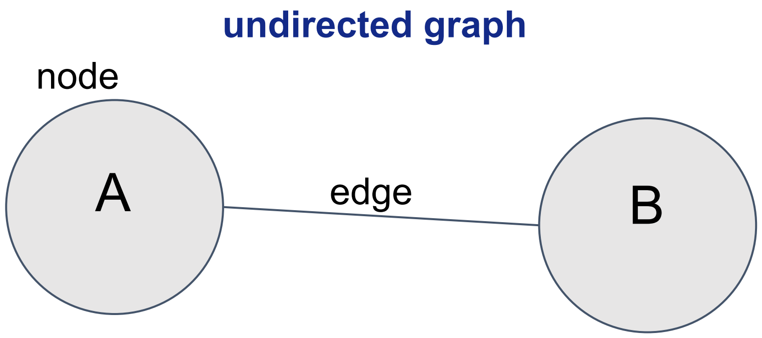
Graphs

What is a graph? Let's do a quick review.

A collection of nodes (things) and edges (connections or links)

Graphs can be directed (usually drawn with arrow to show direction) or undirected

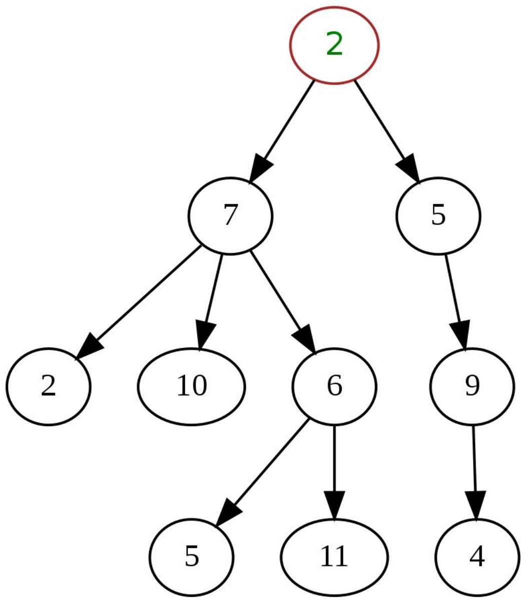


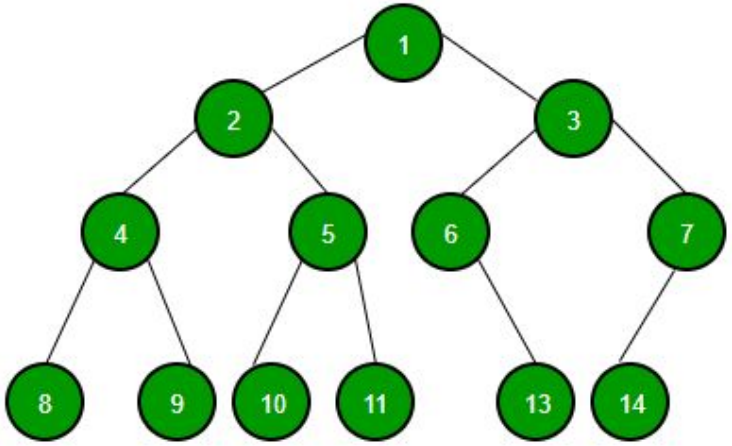


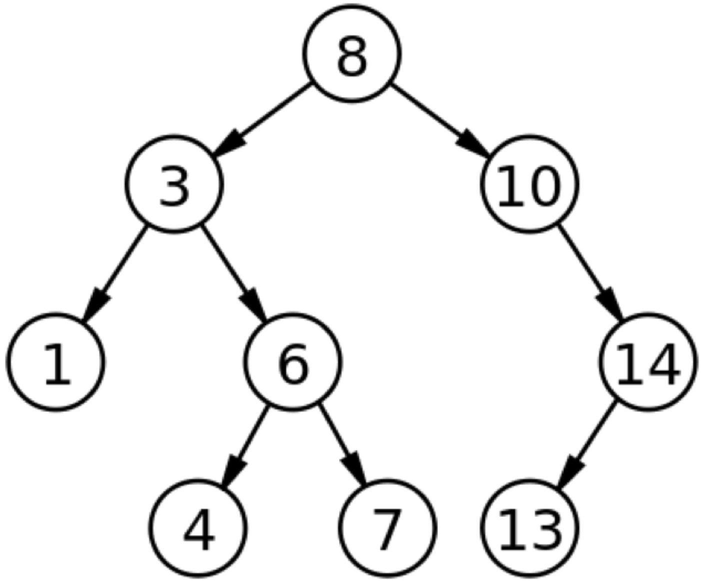
Quick Introduction to Tree

A tree is a widely used abstract data type that represents a hierarchical structure with a set of connected nodes.

Binary trees, Binary search trees, etc.







Up - Trees

An **up - tree** is a specific type of directed graph where each node points “upward” to another node, unless the node is at the very top, which we call the sentinel node (or root).

Each node has **only** 1 parent “pointer”

Let’s redefine our find and unite functions…

★ Add - add a new disjoint subset

★ Find - return the sentinel node of the element in question

★ Unite/Union - Given two elements A and B, find the sentinel node of each and make one “point” to the other

Representation

How can we represent this structure with code?

1. A collection of nodes
2. Each node points to its parent
3. Good performance

Representation

How can we represent this structure with code?

Arrays!

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

Using the above array, please draw the up - tree

Basic Functions

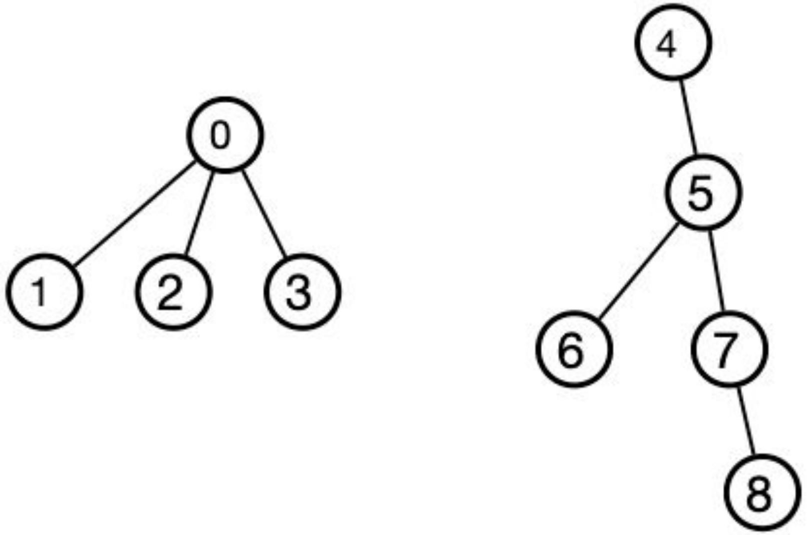
★ **Find**: Given a particular element of the set, it identifies the subset of the element

How do we know two elements are in same set (tree)

★ **Unite**: Joins two subsets into a single subset

How do we know we already joined two trees?

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

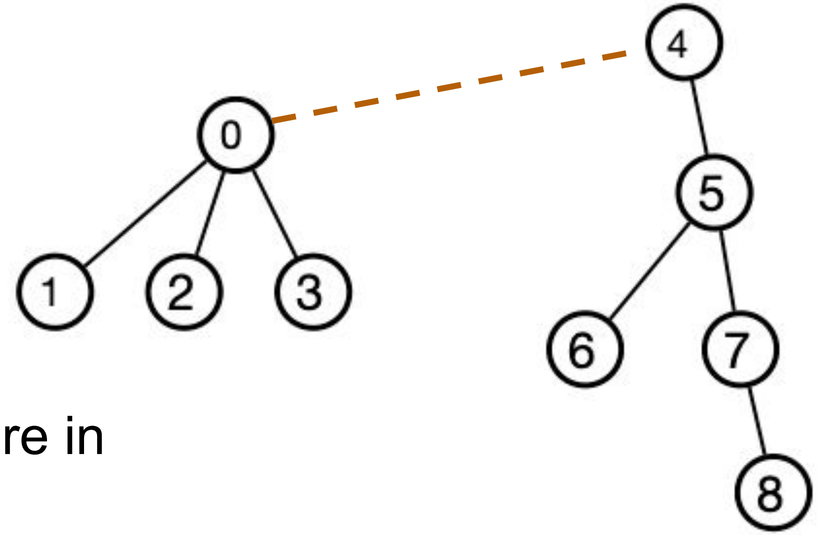


Basic Functions

Consider the 2 basic functions of DSU

```
int get(int x) {
    // return root of node x
}

void unite(int x, int y) {
    // unite the two trees that x and y are in
}
```

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



Practice: unite two disjoint sets that nodes 1 and 8 are in
What will parent array look like after union?
https://onlinegdb.com/EtFqdZ9OW

Code

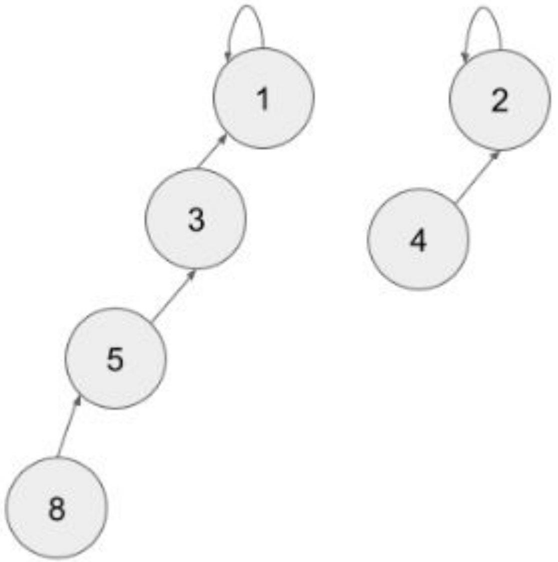
```
// p is for parent, we start p[x] = x
int get(int x) {
    if (x == p[x]) return x; // x is root
    return get(p[x]) // keep moving up
}

void unite(int x, int y) {
    x = get(x);
    y = get(y);
    p[x] = y;
}
```

Improvements

Question: What is the performance of the code?

What happens when the tree is very long?

How can we fix/improve this?



Improvements

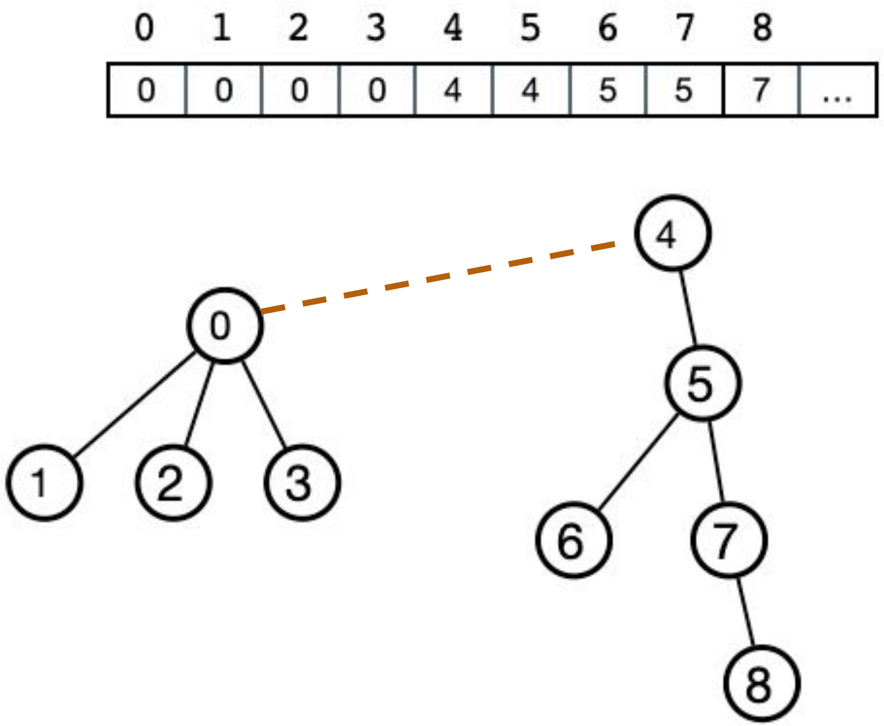
Question: In this example, should final array be

4 0 0 0 4 4 5 5 7

or

0 0 0 0 0 4 5 5 7

How do we make sure it is more balanced?

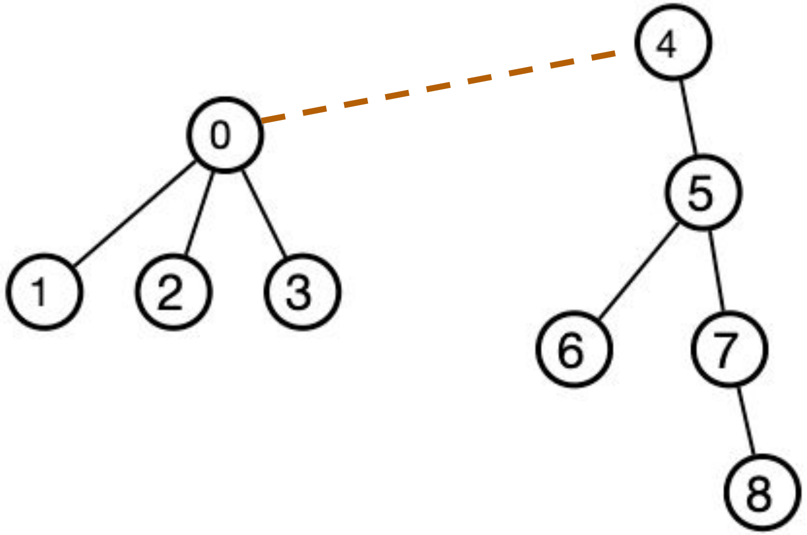


Union by Rank

First Improvement - Link lower rank tree to higher rank one (use layer depth as rank)

```
int get(int x) {
    if (x == p[x]) return x;
    return get(p[x])
}

void unite(int x, int y) {
    x = get(x);
    y = get(y);
    if (r[x] == r[y]) {
        p[x] = y;
        r[y]++; // why does this change?
    } else if (r[x] < r[y]) {
        p[x] = y;
    } else p[y] = x;
}
```

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

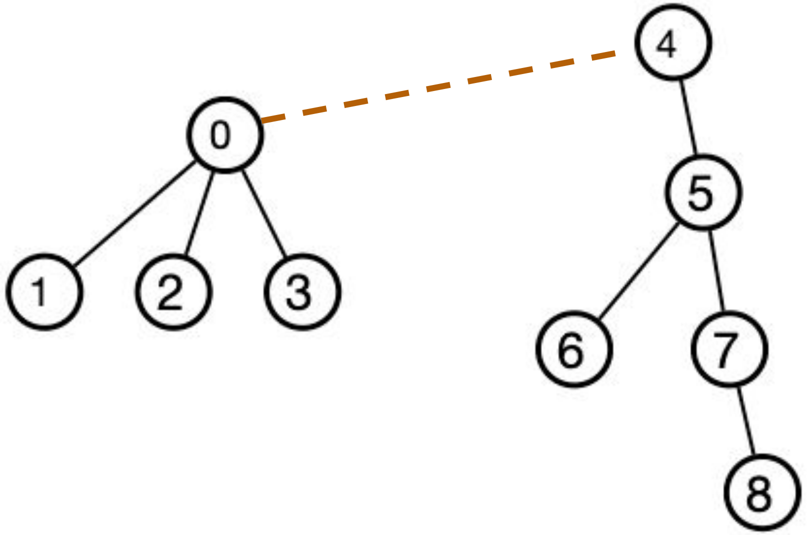


Tree Height

Tip - Keep track of the node size as you unite items together!

```
void unite(int x, int y) {
    x = get(x);
    y = get(y);
    if (r[x] == r[y]) {
        p[x] = y;
        r[y]++;
        size[y] += size[x];
    } else if (r[x] < r[y]) {
        p[x] = y;
        size[y] += size[x];
    } else {
        p[y] = x;
        size[x] += size[y];
    }
}
```

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



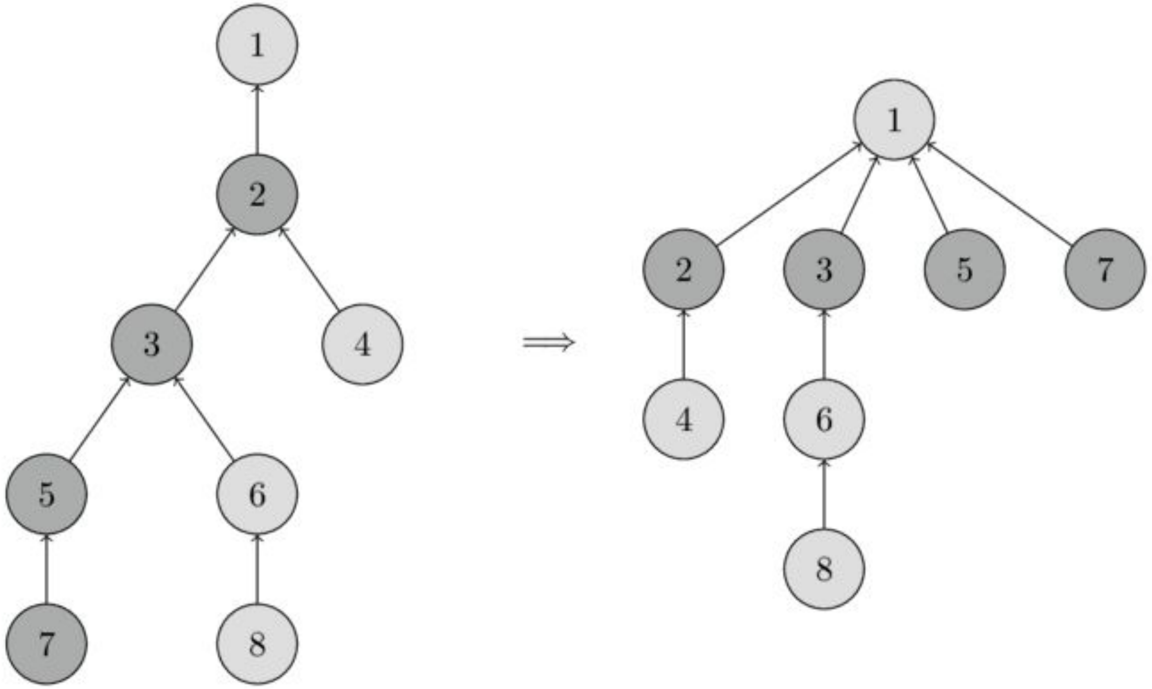
Path Compression

Second Improvement - link nodes directly to root to avoid unnecessary recursive calls.

This is called **Path Compression**.

What is the benefit?

How to implement it?



Path Compression

Second Improvement - link nodes directly to root to avoid unnecessary recursive calls

Naive

```
int get(int x) {
    if (x == p[x]) return x;
    return get(p[x])
}
```

Compressed

```
int get(int x) {
    if (x == p[x]) return x;
    return p[x] = get(p[x])
}
```

Review

- What were the two optimizations we made and why did we make them?
- How can we represent these sets of connected nodes in code?
- how many parent pointers does each node have?

Review - Key Factors for BFS

	Pioneer	Word Ladder	Pour Wine
State	int maze[501][501] struct PNT { int x, y;}	set<string> dict	int cup_volumes[3]; map<vector<int>, int> target_wine_volumes; vector<int> wine_volumes;
Transition	Up, down, left, right	isNeighbor()	int from_cups[6] = {0, 0, 1, 1, 2, 2}; int to_cups[6] = {1, 2, 0, 2, 0, 1}; Either from_cup is empty, or to_cup is full.
Visited tracking	maze[i][j] == -1 ?	map<string, int> step;	map<vector<int>, int> visited_wine_volumes; Do existence check
Count	maze[][]	map<string, int> step;	map<vector<int>, int> visited_wine_volumes; Check integer value.