Graphs and Union-Find

Welcome back to CS 2100!

Prof. Rasika Bhalerao

Let's use recursion to search for an element in a `Tree`:

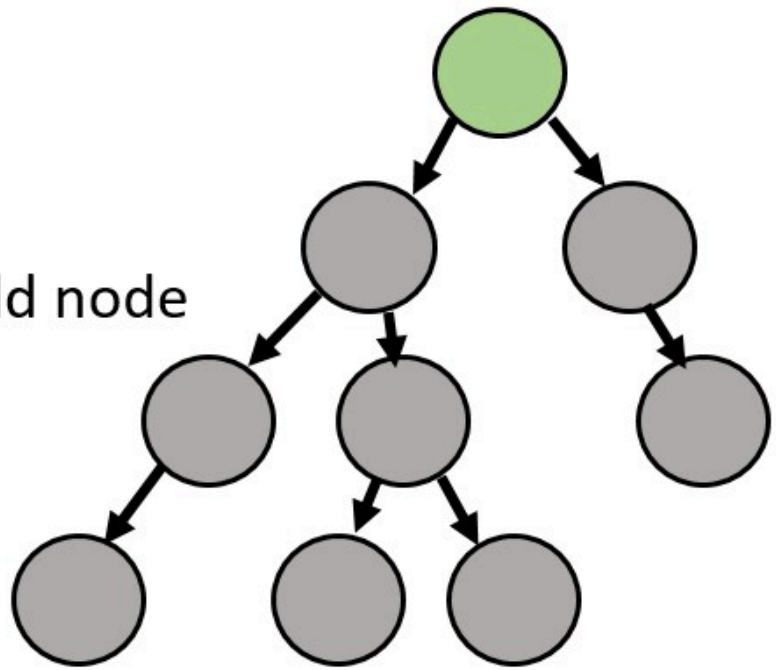
```
class Tree(Generic[T]):
    def __init__(self, root_data: Optional[T] = None) -> None:
        if root_data is None:
            self.root: Optional[Node[T]] = None
        else:
            self.root = Node[T](root_data)

    def __str__(self) -> str:
        return self.root.__str__()

    def __contains__(self, item: T) -> bool:
        return self.contains(item, self.root)

    def contains(self, item: T, node: Optional[Node[T]]) -> bool:
        if node is None:
            return False
        elif node.data == item:
            return True
        else:
            return self.contains(item, node.left) or self.contains(item, node.right)

tree: Tree[str] = Tree[str]('Entry way')

assert tree.root is not None
tree.root.left = Node[str]('Living room')

tree.root.left.right = Node[str]('Kitchen')

print('Kitchen' in tree)  # True
print('Bathroom' in tree)  # False
```

Recursive backtracking

Recursive backtracking is a strategy to search every node of a tree.

Strategy: given a node, for each of its children:

1. Choose a child to explore
2. Recursively perform this strategy for that child node
3. Un-choose that child

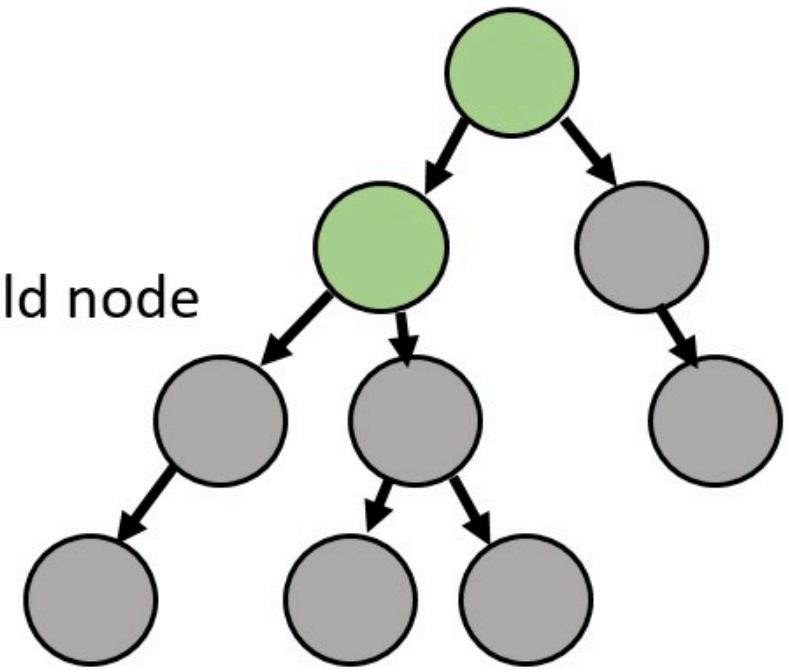


Recursive backtracking

Recursive backtracking is a strategy to search every node of a tree.

Strategy: given a node, for each of its children:

1. Choose a child to explore
2. Recursively perform this strategy for that child node
3. Un-choose that child

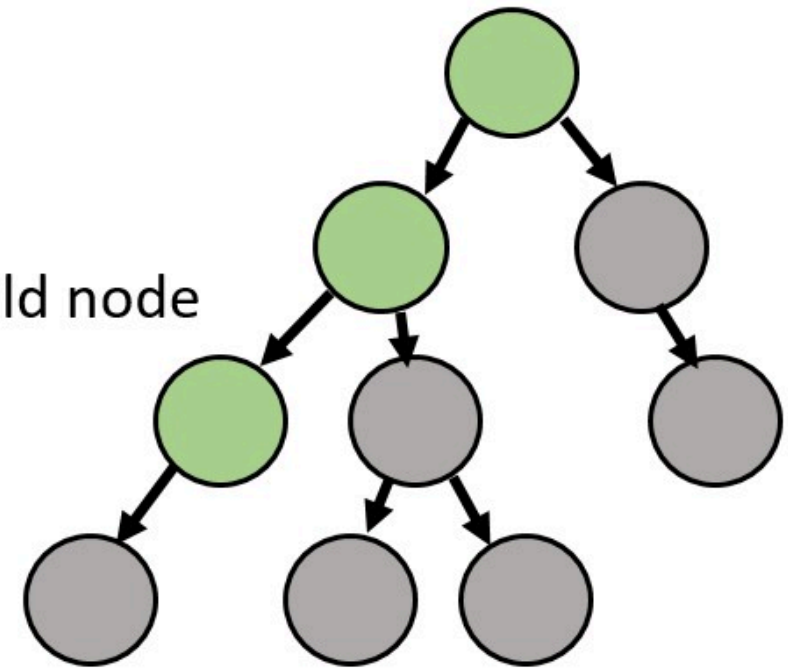


Recursive backtracking

Recursive backtracking is a strategy to search every node of a tree.

Strategy: given a node, for each of its children:

1. Choose a child to explore
2. Recursively perform this strategy for that child node
3. Un-choose that child

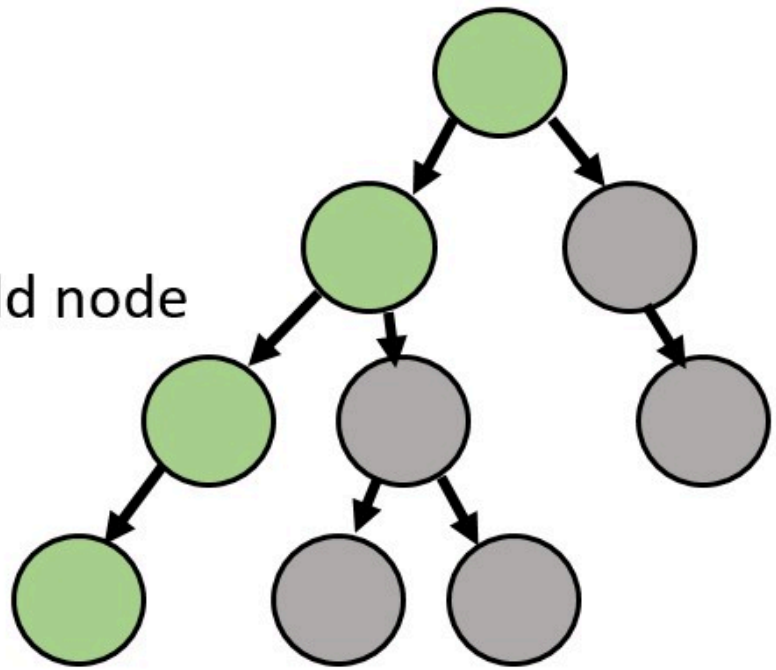


Recursive backtracking

Recursive backtracking is a strategy to search every node of a tree.

Strategy: given a node, for each of its children:

1. Choose a child to explore
2. Recursively perform this strategy for that child node
3. Un-choose that child

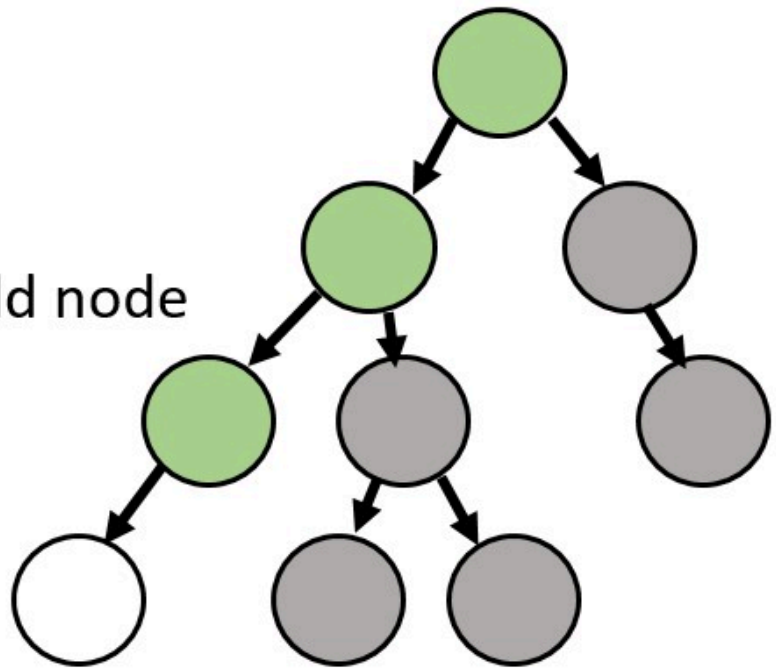


Recursive backtracking

Recursive backtracking is a strategy to search every node of a tree.

Strategy: given a node, for each of its children:

1. Choose a child to explore
2. Recursively perform this strategy for that child node
3. Un-choose that child

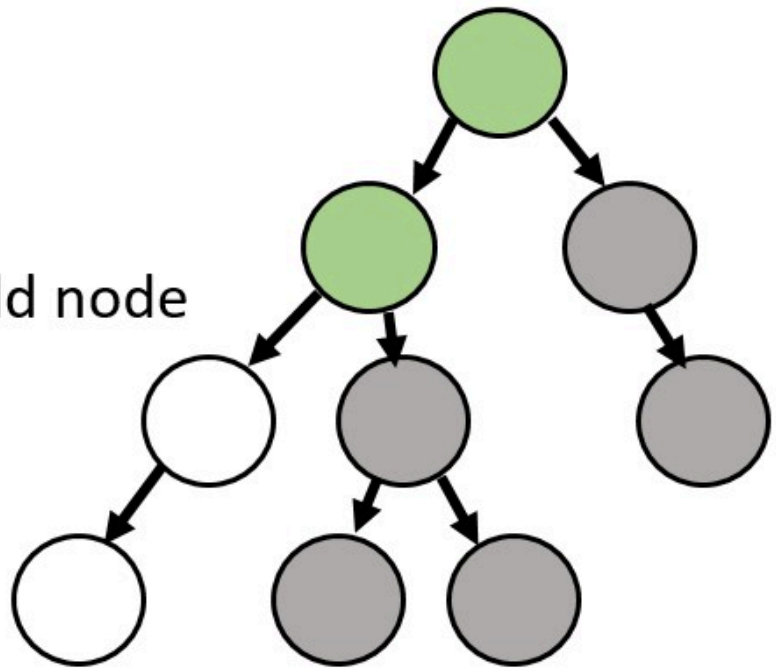


Recursive backtracking

Recursive backtracking is a strategy to search every node of a tree.

Strategy: given a node, for each of its children:

1. Choose a child to explore
2. Recursively perform this strategy for that child node
3. Un-choose that child

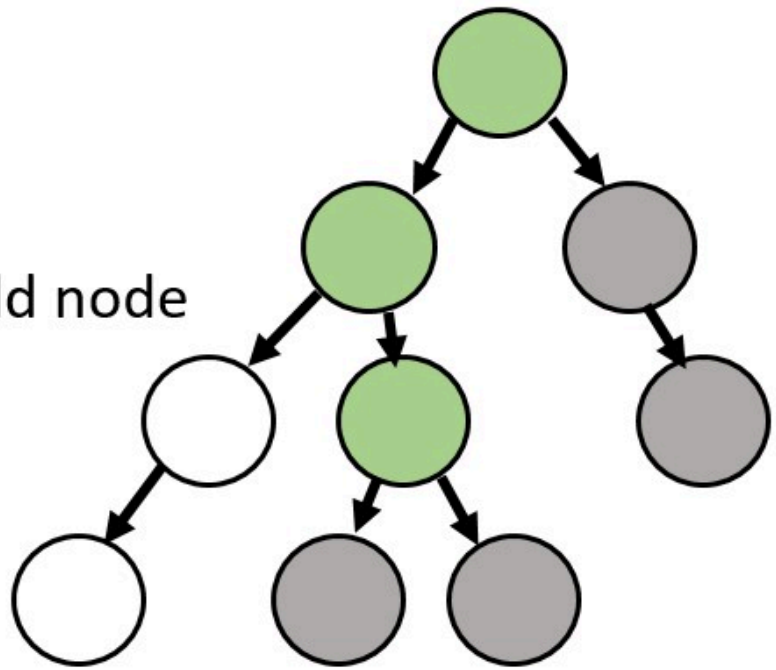


Recursive backtracking

Recursive backtracking is a strategy to search every node of a tree.

Strategy: given a node, for each of its children:

1. Choose a child to explore
2. Recursively perform this strategy for that child node
3. Un-choose that child

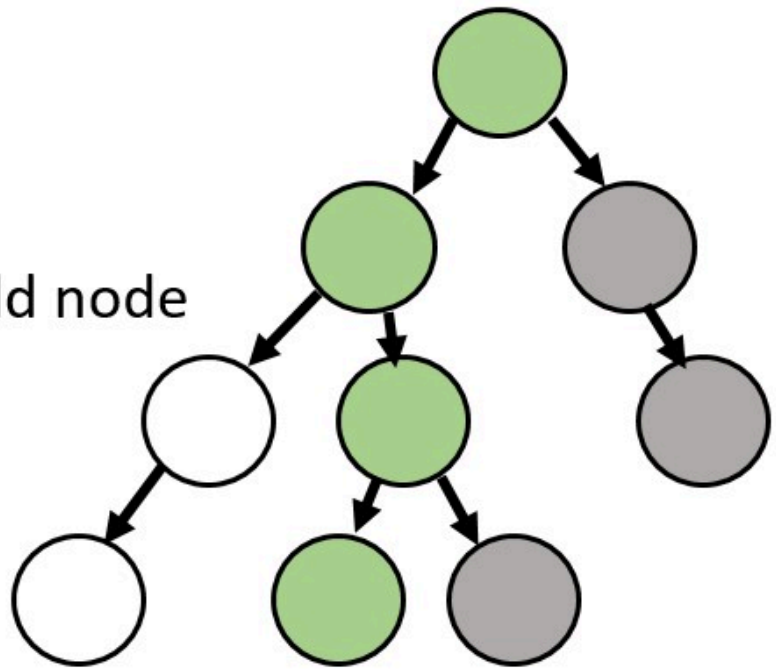


Recursive backtracking

Recursive backtracking is a strategy to search every node of a tree.

Strategy: given a node, for each of its children:

1. Choose a child to explore
2. Recursively perform this strategy for that child node
3. Un-choose that child

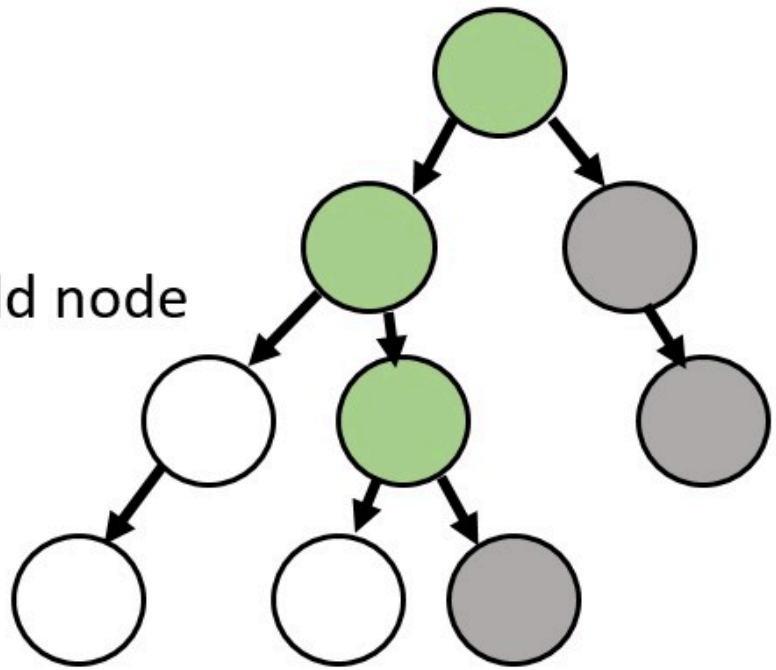


Recursive backtracking

Recursive backtracking is a strategy to search every node of a tree.

Strategy: given a node, for each of its children:

1. Choose a child to explore
2. Recursively perform this strategy for that child node
3. Un-choose that child

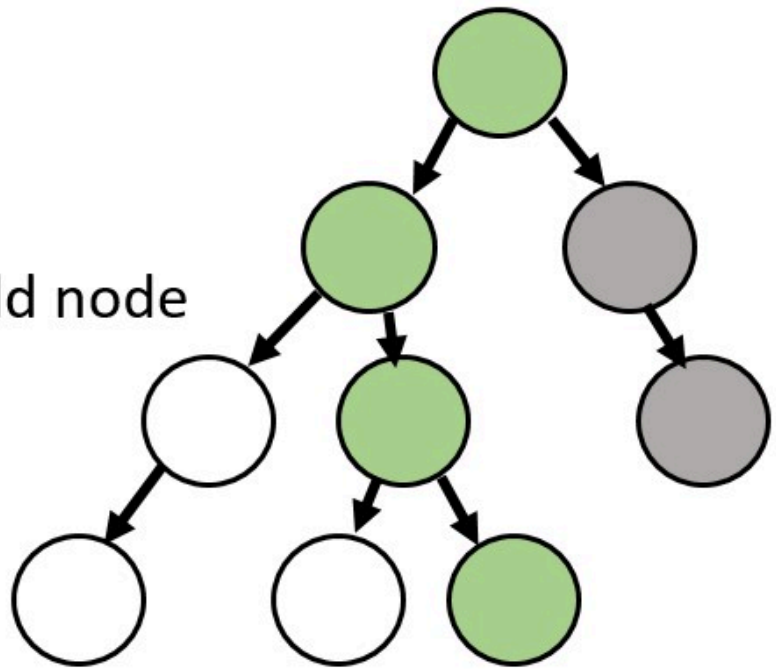


Recursive backtracking

Recursive backtracking is a strategy to search every node of a tree.

Strategy: given a node, for each of its children:

1. Choose a child to explore
2. Recursively perform this strategy for that child node
3. Un-choose that child

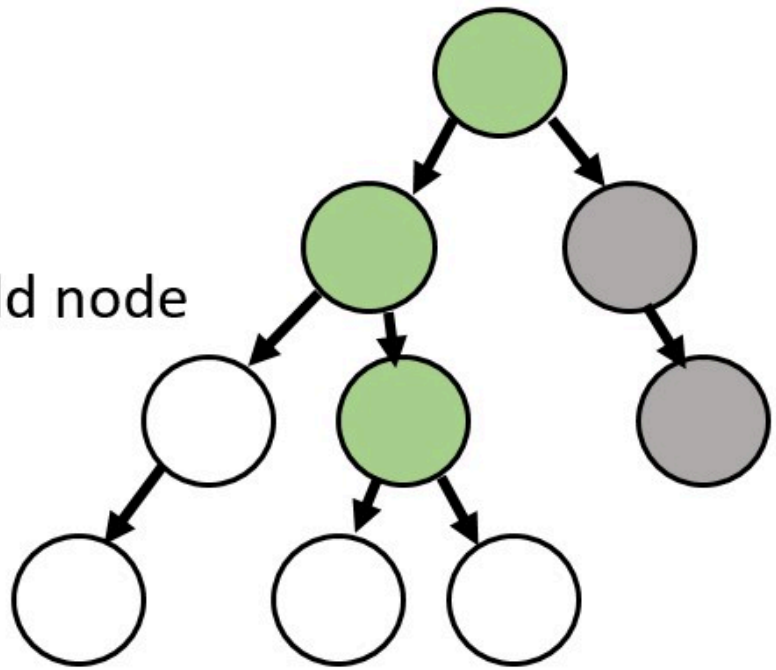


Recursive backtracking

Recursive backtracking is a strategy to search every node of a tree.

Strategy: given a node, for each of its children:

1. Choose a child to explore
2. Recursively perform this strategy for that child node
3. Un-choose that child

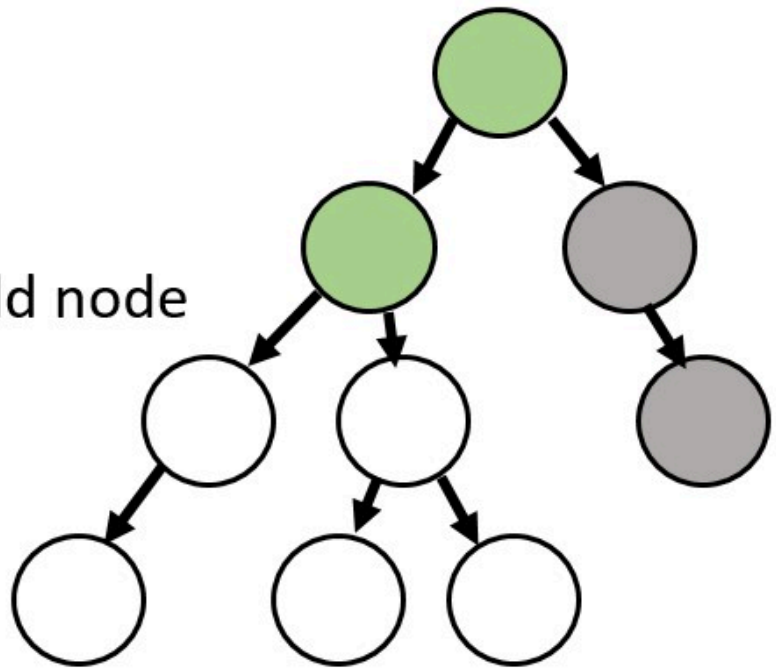


Recursive backtracking

Recursive backtracking is a strategy to search every node of a tree.

Strategy: given a node, for each of its children:

1. Choose a child to explore
2. Recursively perform this strategy for that child node
3. Un-choose that child

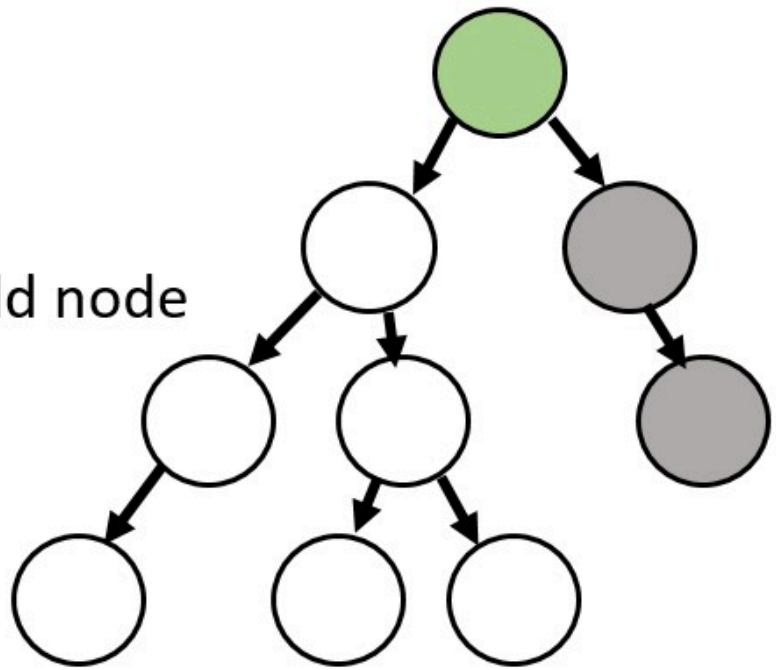


Recursive backtracking

Recursive backtracking is a strategy to search every node of a tree.

Strategy: given a node, for each of its children:

1. Choose a child to explore
2. Recursively perform this strategy for that child node
3. Un-choose that child

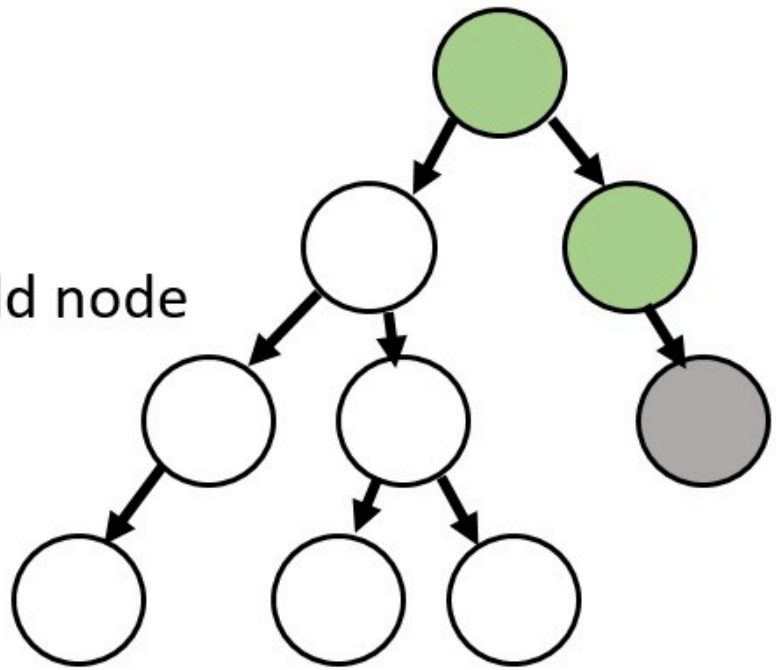


Recursive backtracking

Recursive backtracking is a strategy to search every node of a tree.

Strategy: given a node, for each of its children:

1. Choose a child to explore
2. Recursively perform this strategy for that child node
3. Un-choose that child

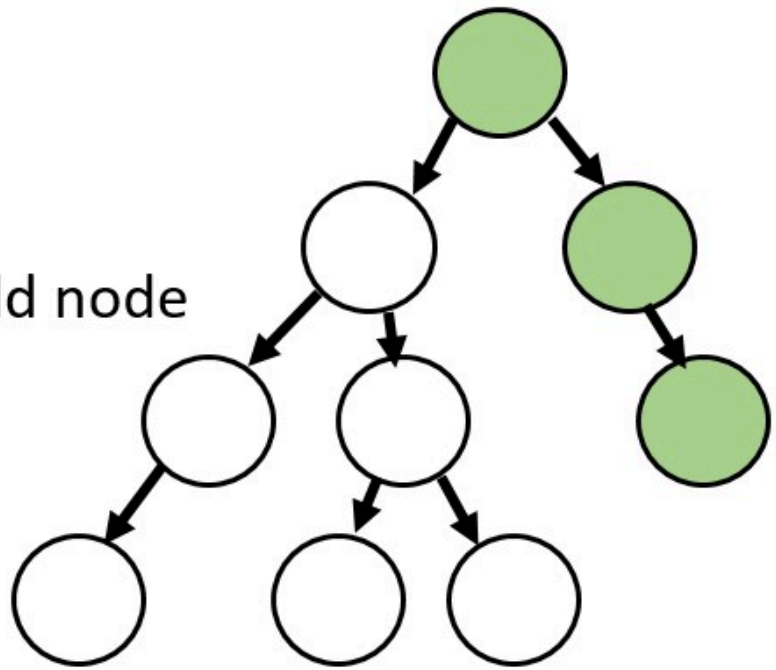


Recursive backtracking

Recursive backtracking is a strategy to search every node of a tree.

Strategy: given a node, for each of its children:

1. Choose a child to explore
2. Recursively perform this strategy for that child node
3. Un-choose that child

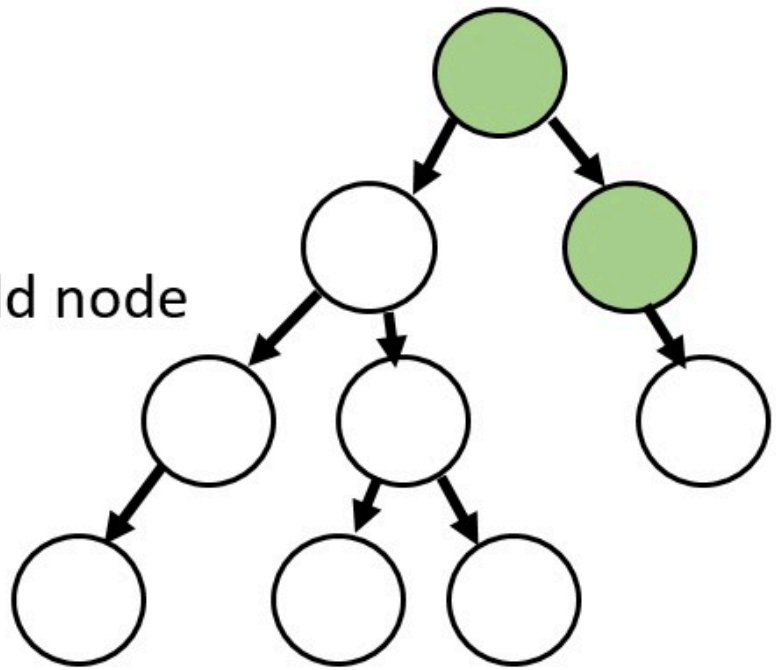


Recursive backtracking

Recursive backtracking is a strategy to search every node of a tree.

Strategy: given a node, for each of its children:

1. Choose a child to explore
2. Recursively perform this strategy for that child node
3. Un-choose that child

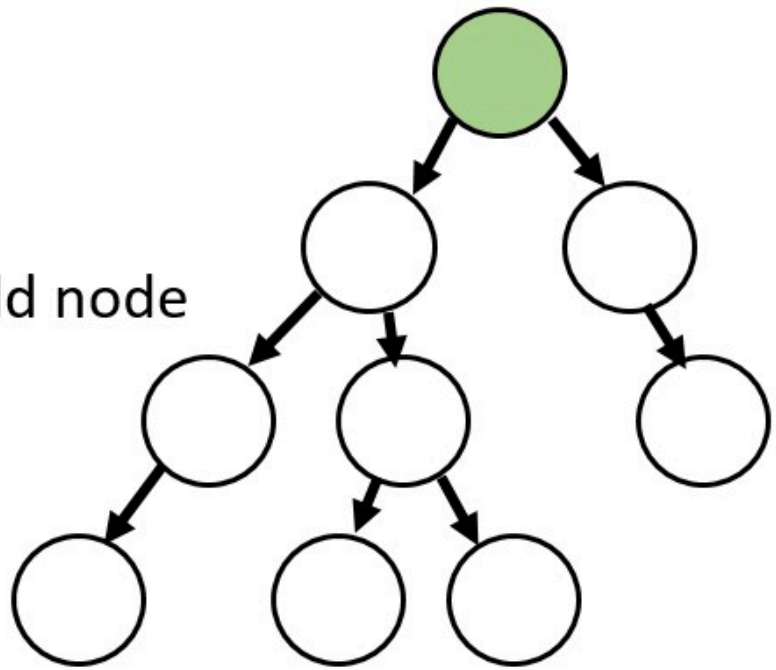


Recursive backtracking

Recursive backtracking is a strategy to search every node of a tree.

Strategy: given a node, for each of its children:

1. Choose a child to explore
2. Recursively perform this strategy for that child node
3. Un-choose that child

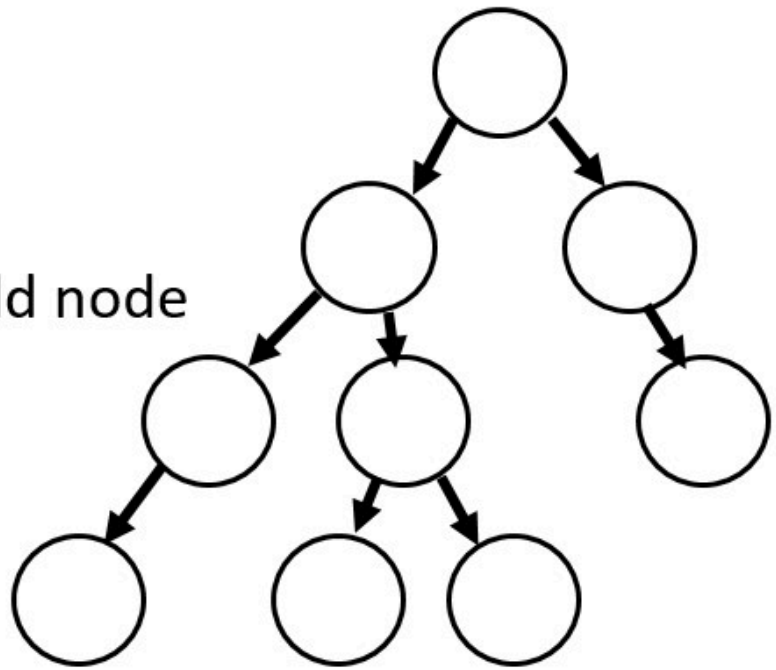


Recursive backtracking

Recursive backtracking is a strategy to search every node of a tree.

Strategy: given a node, for each of its children:

1. Choose a child to explore
2. Recursively perform this strategy for that child node
3. Un-choose that child



That recursive backtracking algorithm is called Depth-First Search, since it explores "deep" in one area before moving on to other unexplored "shallow" places (closer to the root).

Poll: This is pseudocode for a Depth-First Search on a graph that is not a tree (because it has cycles). What's a good base case?

```
DFS(node):
    Base case:
        ???
    Recursive case:
        For each child:
            Add child to explored nodes
            DFS(child)
```

1. If the node is in the set of explored nodes, do nothing
2. If the node is a leaf, add it to the set of explored nodes
3. If the node is `None`, do nothing
4. If the node is `None`, add it to the set of explored nodes

Graphs which might not be trees

- No root or leaves
- Instead of having parent-child relationships:
  - Directed graph: edges point *from* one node *to* another
  - Undirected graph: edges do not have a specific direction; all edges go both ways

Graph term	Definition	Social network example
Node	Same as tree's node	A person
Directed edge	Edge points from one node to another	It's possible Blob is Glob's friend but not vice versa
Undirected edge	Edge must go both ways	If Blob is Glob's friend, then Glob must be Blob's friend

```python
class Person:
    def __init__(self, person_id: str):
        self.id = person_id
        self.friends: set[Person] = set()

    def __str__(self) -> str:
        return self.id

    def __repr__(self) -> str:
        return self.__str__()

class SocialGraph:
    def __init__(self, location: str) -> None:
        self.people: set[Person] = set()
        self.location = location

    def __str__(self) -> str:
        return f'People in {self.location}: {self.people}'
```

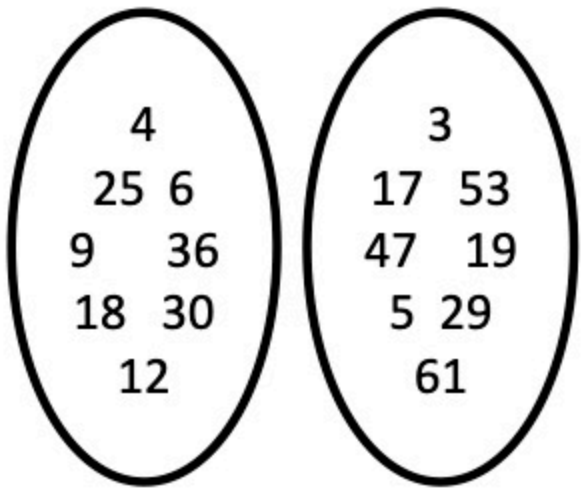
```python
students = SocialGraph('Oakland')

me = Person('Rasika')
students.people.add(me)

mini = Person('Mini')
students.people.add(mini)

me.friends.add(mini)
mini.friends.add(me)

print(students)  # People in Oakland: {Mini, Rasika}

famous_person = Person('Famous')
students.people.add(famous_person)

me.friends.add(famous_person)
mini.friends.add(famous_person)

print(me.friends)  # {Mini, Famous}
print(mini.friends)  # {Rasika, Famous}
print(famous_person.friends)  # set()
```

Poll: Is our social media graph above directed or undirected?

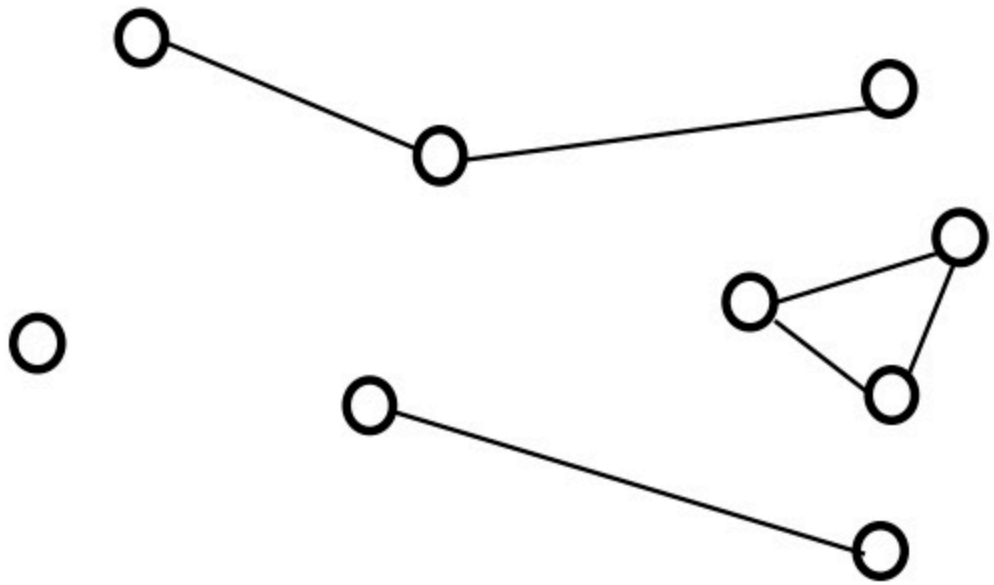
1. Directed
2. Undirected

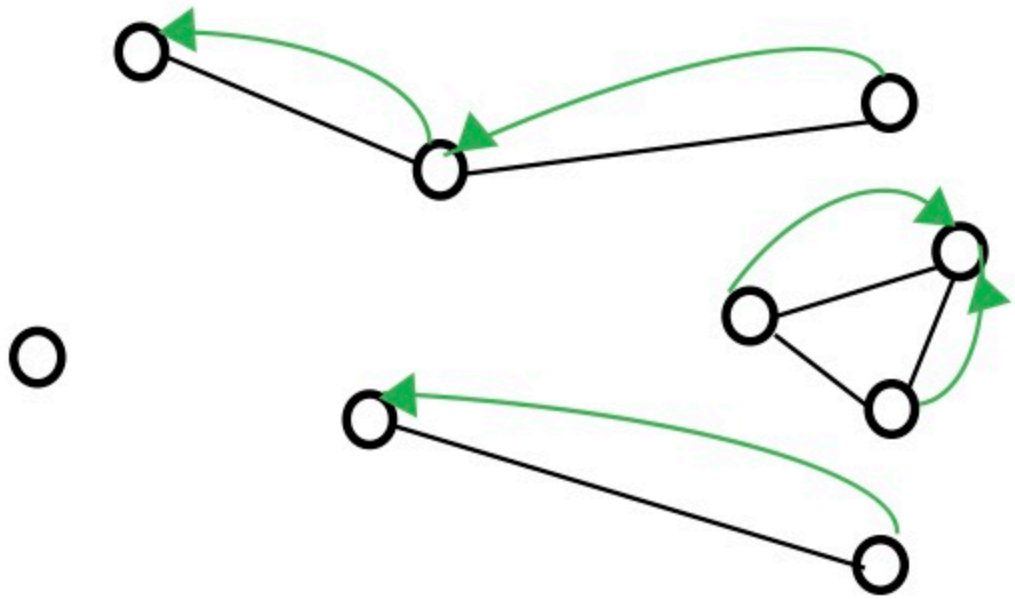
The Union-Find algorithm finds disjoint sets.

Disjoint sets are sets that have no elements in common.



https://www.worldatlas.com/geography/continents-by-number-of-countries.html

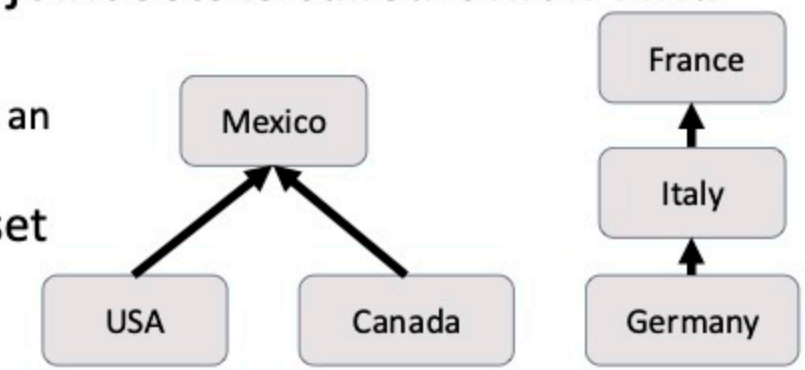




Representing disjoint sets

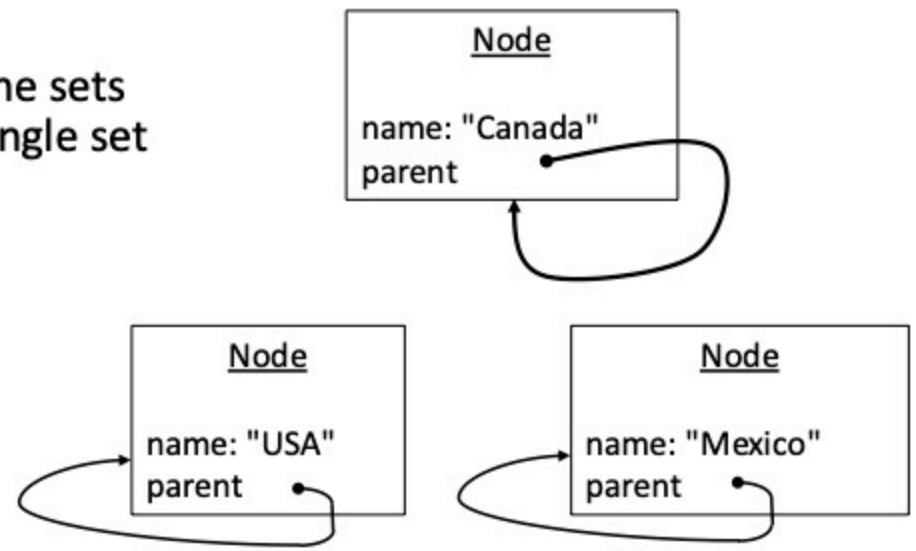
- The goal of Union-Find is to result in a representation of disjoint sets:
  - Each item is a node in a graph
  - Each disjoint set is a tree within a *forest*
  - Each disjoint set has one element which is the *representative element*
    - That item is the root of the tree.

- The algorithm for finding / managing disjoint sets is called Union-Find
  - union: combine two sets
    - Do this when we find that those two sets have an element in common
  - find: find the representative element of a set
    - Given an element, find the root of the tree that it belongs to



Union-Find overview

- Each node keeps track of:
  - Its data (the element)
  - Its parent (another node)
    - `node.parent == node` if node is a root
    - `node.parent == node2` if node2 is its parent
- Operations
  - `union(node1, node2)` combines the sets containing node1 and node2 into a single set
  - `find(node)` returns node's root

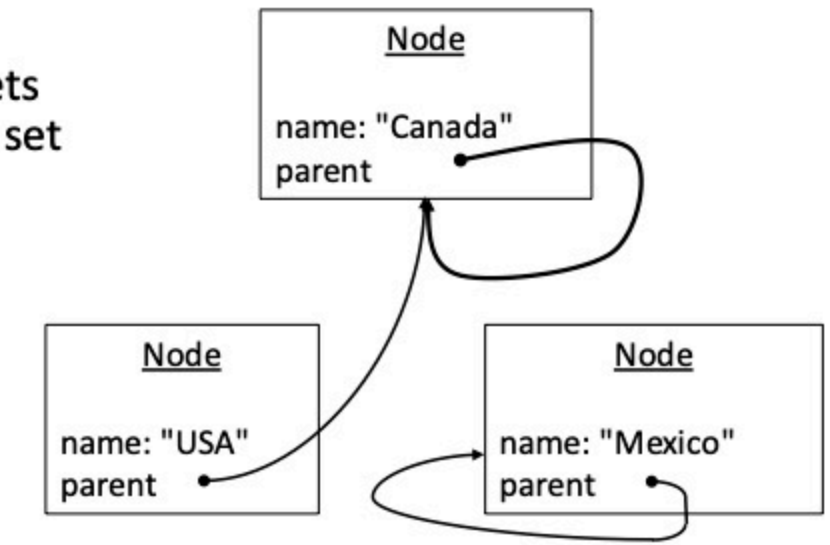


Union-Find overview

- Each node keeps track of:
  - Its data (the element)
  - Its parent (another node)
    - `node.parent == node` if node is a root
    - `node.parent == node2` if node2 is its parent
- Operations
  - `union(node1, node2)` combines the sets containing node1 and node2 into a single set

    `union(USA, Canada)`

  - `find(node)` returns node's root

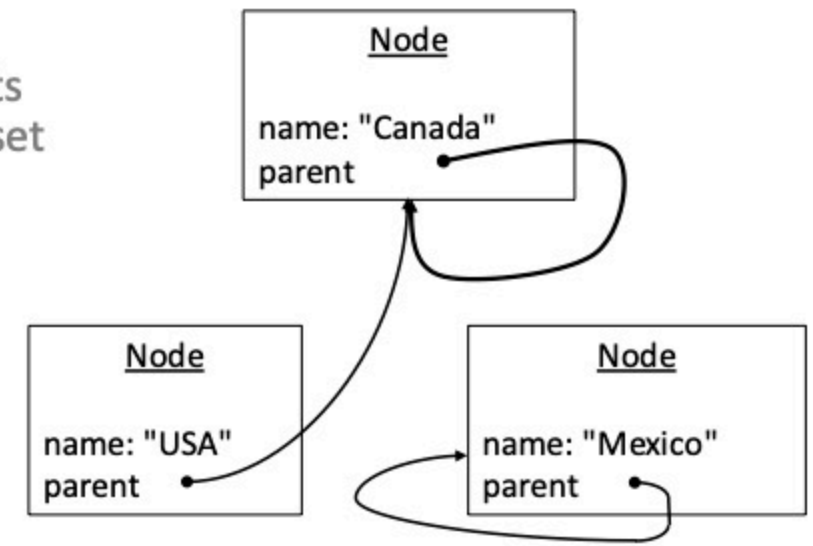


Poll: How might we implement `find(node)`?

- Each node keeps track of:
  - Its data (the element)
  - Its parent (another node)
    - `node.parent == node` if node is a root
    - `node.parent == node2` if node2 is its parent
- Operations
  - `union(node1, node2)` combines the sets containing node1 and node2 into a single set

    `union(USA, Canada)`

  - **`find(node)` returns node's root**

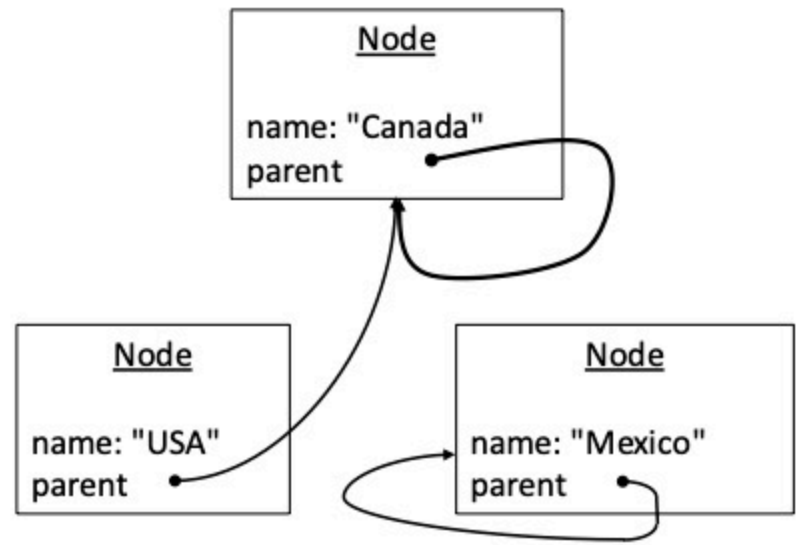


Implementing `find(node)`

- Each node keeps track of:
  - Its data (the element)
  - Its parent (another node)
    - `node.parent == node` if node is a root
    - `node.parent == node2` if node2 is its parent

Given a node, we want to find its root

```
find (node):
    while (node.parent != node):
        node = node.parent
    return node
```

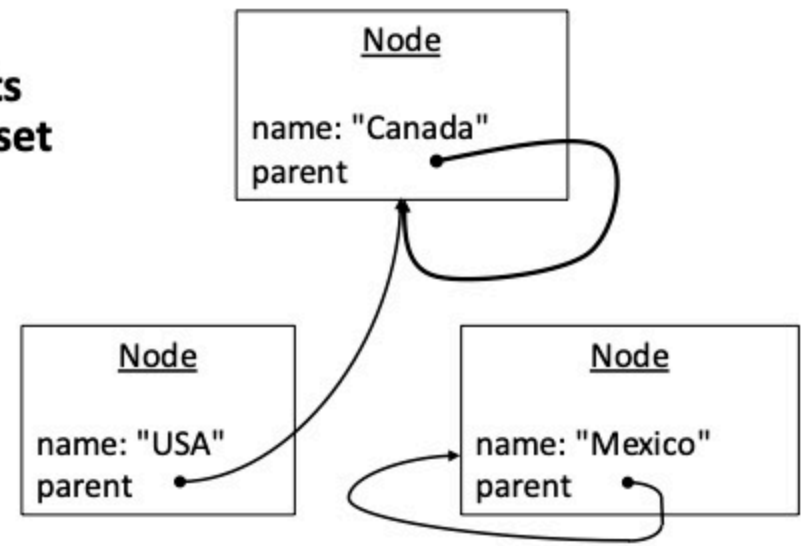


Poll: How might we implement `union()`?

- Each node keeps track of:
  - Its data (the element)
  - Its parent (another node)
    - `node.parent == node` if node is a root
    - `node.parent == node2` if node2 is its parent
- Operations
  - **`union(node1, node2)` combines the sets containing node1 and node2 into a single set**

    `union(USA, Canada)`

  - `find(node)` returns node's root

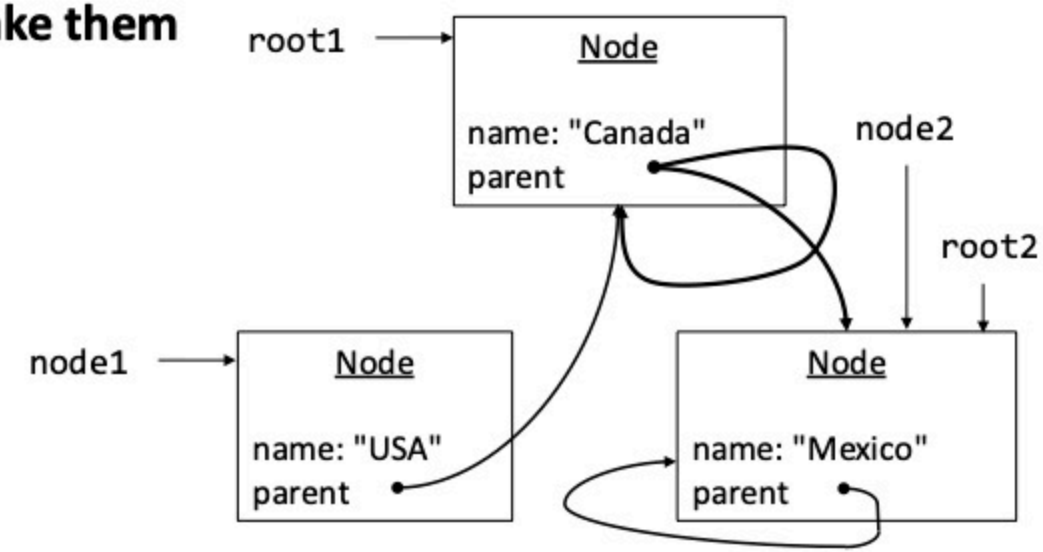


Implementing `union(node1, node2)`

- Each node keeps track of:
  - Its data (the element)
  - Its parent (another node)
    - `node.parent == node` if node is a root
    - `node.parent == node2` if node2 is its parent

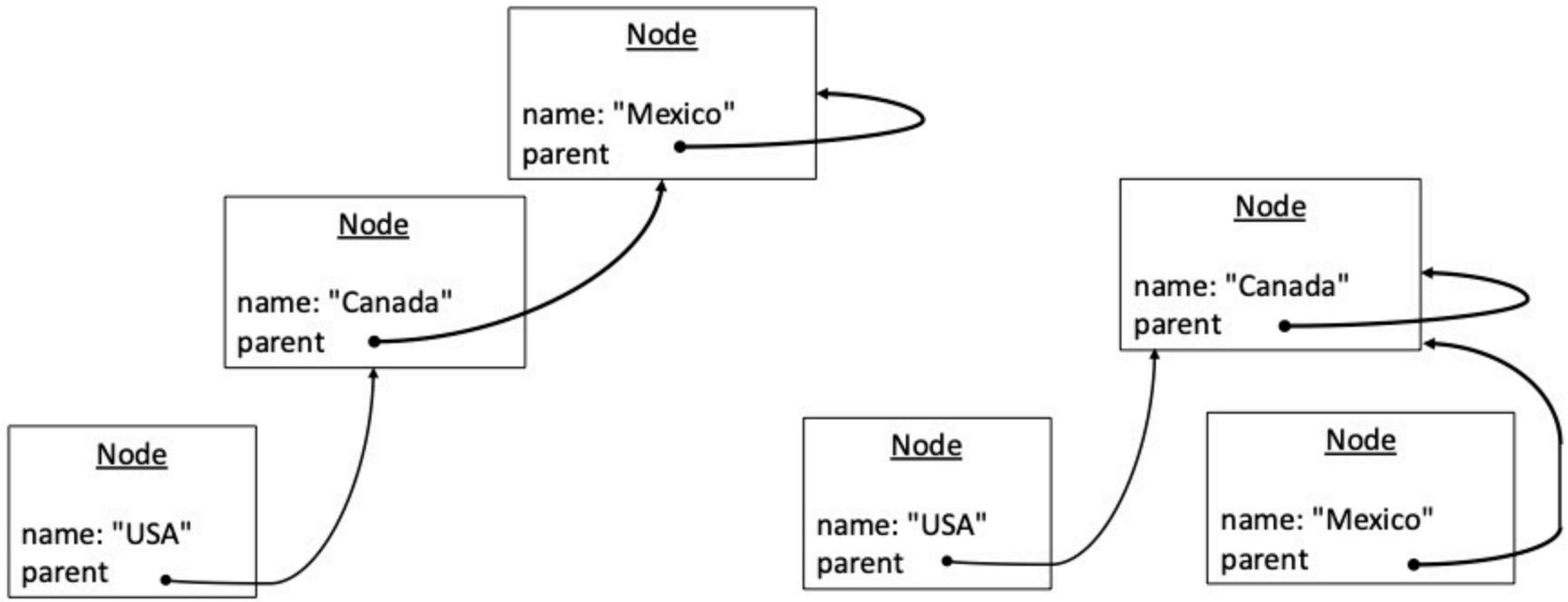
Given two nodes, we want to make them have the same root

```
union (node1, node2):
    root1 = find(node1)
    root2 = find(node2)
    if (root1 != root2):
        root1.parent = root2
```



Problem with that union() implementation

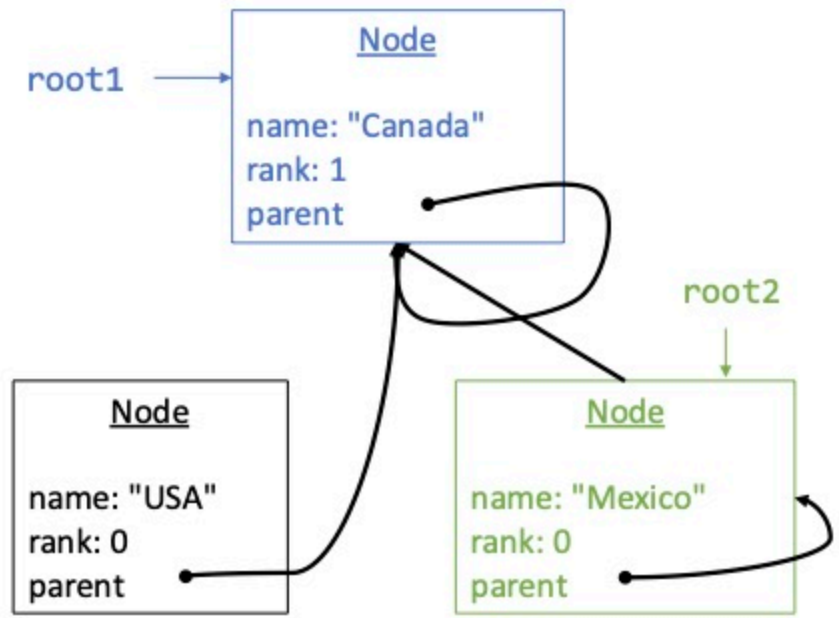
Algorithm can lead to long paths (trees of great height)



A better implementation of `union(node1, node2)`

Rank: how "high up" that node is in the tree
Aim is to keep ranks low

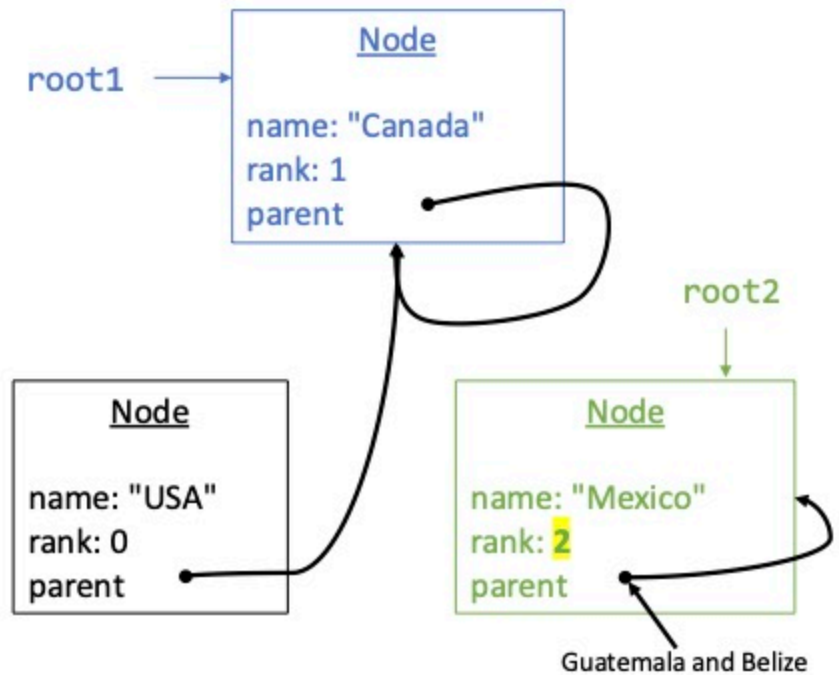
```
union(node1, node2):
  root1 = find(node1)
  root2 = find(node2)
  if root1 != root2:
    if root1.rank > root2.rank:
      root2.parent = root1
    else:
      root1.parent = root2
      if root1.rank == root2.rank:
        root2.rank++

union(USA, Mexico)
```



A better implementation of `union(node1, node2)`

Rank: how "high up" that node is in the tree
Aim is to keep ranks low

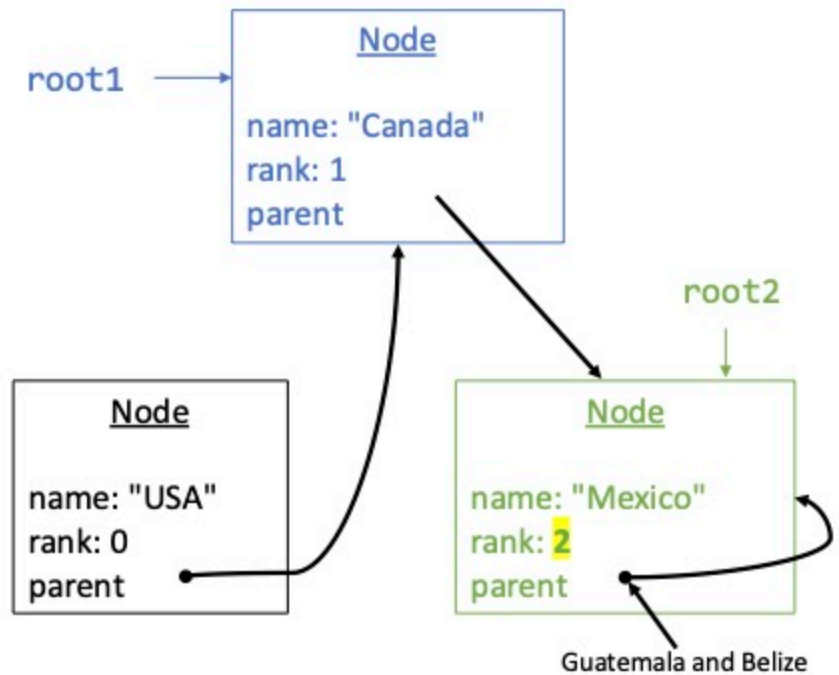
```
union(node1, node2):
  root1 = find(node1)
  root2 = find(node2)
  if root1 != root2:
    if root1.rank > root2.rank:
      root2.parent = root1
    else:
      root1.parent = root2
      if root1.rank == root2.rank:
        root2.rank++

union(USA, Mexico)
```



A better implementation of `union(node1, node2)`

```
union(node1, node2):
  root1 = find(node1)
  root2 = find(node2)
  if root1 != root2:
    if root1.rank > root2.rank:
      root2.parent = root1
    else:
      root1.parent = root2
      if root1.rank == root2.rank:
        root2.rank++

union(USA, Mexico)
```

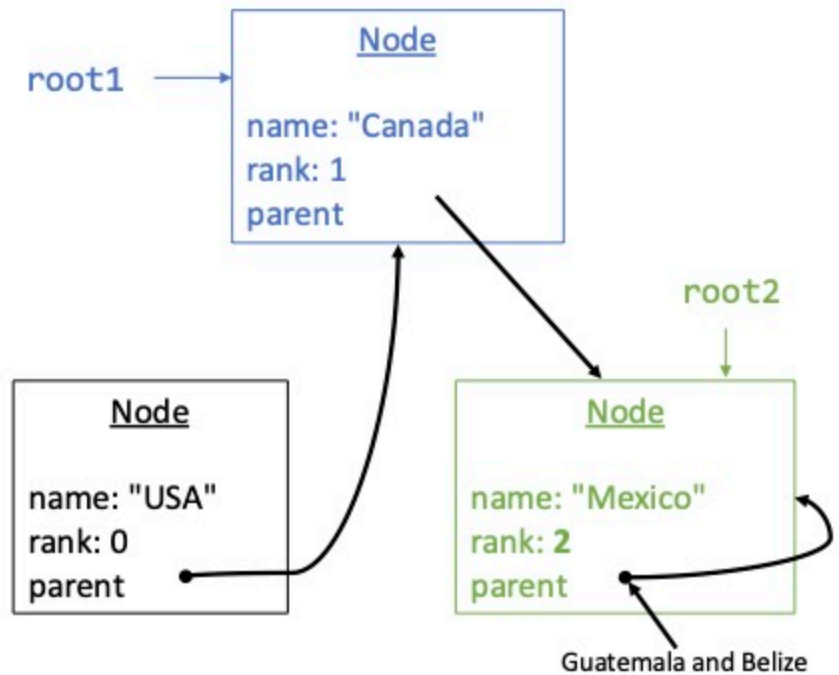
Rank: how "high up" that node is in the tree
Aim is to keep ranks low



Poll: Why do we check if `root1.rank == root2.rank` before incrementing `root2.rank`?

```
union(node1, node2):
  root1 = find(node1)
  root2 = find(node2)
  if root1 != root2:
    if root1.rank > root2.rank:
      root2.parent = root1
    else:
      root1.parent = root2
      if root1.rank == root2.rank:
        root2.rank++

union(USA, Mexico)
```

Rank: how "high up" that node is in the tree
Aim is to keep ranks low



Poll: What does the Union-Find algorithm do?

A. It groups nodes into sets such that none of the nodes in each set are connected to each other, and all of the edges are between two sets
B. It groups nodes into sets such that the nodes in each set are connected to each other, and there are no edges between two sets
C. It groups edges into sets such that none of the sets of edges have any nodes in common
D. It works on undirected graphs (all edges go both ways)
E. It works on directed graphs (all edges are one-way)

Union-Find Summary

- The Union-Find algorithm finds disjoint sets
  - It groups nodes into sets such that, for each set:
    - The nodes in that set are connected to each other
    - None of the nodes in that set have an edge with a node outside that set

- Each node keeps track of:
  - Its data (the element)
  - Its parent (another node)
    - `node.parent == node` if node is a root
    - `node.parent == node2` if node2 is its parent
- Operations
  - `union(node1, node2)` combines the sets containing node1 and node2 into a single set
  - `find(node)` returns node's root

```
find (node):
    while (node.parent != node):
        node = node.parent
    return node

union(node1, node2):
  root1 = find(node1)
  root2 = find(node2)
  if root1 != root2:
    if root1.rank > root2.rank:
      root2.parent = root1
    else:
      root1.parent = root2
      if root1.rank == root2.rank:
        root2.rank++
```

Poll:

1. What is your main takeaway from today?

2. What would you like to revisit next time?