Trees

Welcome back to CS 2100!

Prof. Rasika Bhalerao

Poll: What's wrong with this recursive function?

```
def find_max(nums: list[int], index: int = 0) -> int:
    """Returns the maximum of the numbers in nums, starting from the given index"""
    if index == len(nums) - 1:
        return nums[index]
    else:
        max_of_rest: int = find_max(nums, index)
        if nums[index] < max_of_rest:
            return max_of_rest
        else:
            return nums[index]
```

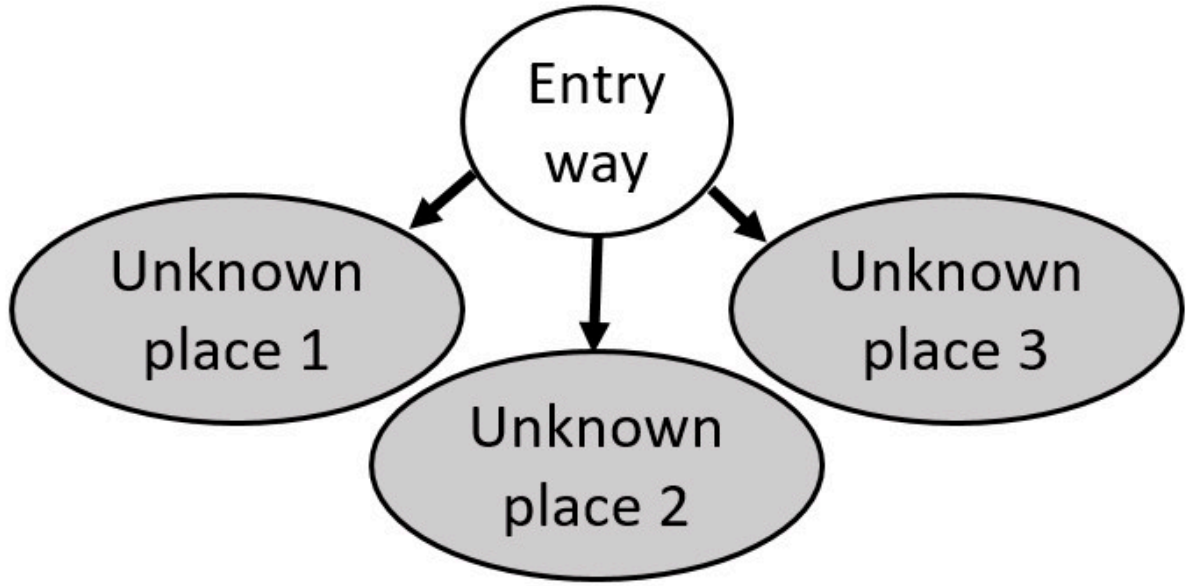
1. It's missing a base case
2. It's missing a recursive case
3. The recursive case doesn't progress towards the base case
4. It has an "off-by-one bug"

Poll: What's wrong with this recursive function?

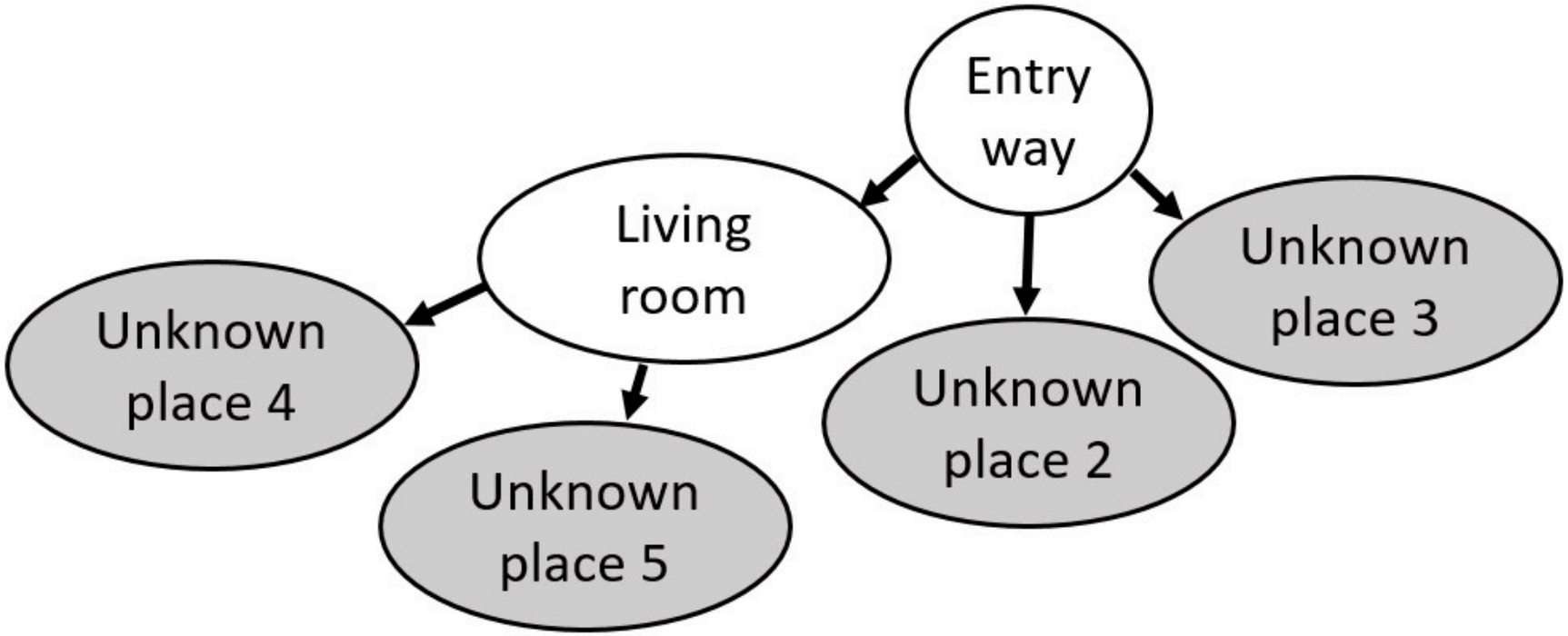
```
def countdown(n: int) -> None:
    """Prints numbers from n down to 0, one on each line"""
    if n > 0:
        print(n)
        countdown(n - 1)
```

1. It's missing a base case
2. It's missing a recursive case
3. The recursive case doesn't progress towards the base case
4. It has an "off-by-one bug"

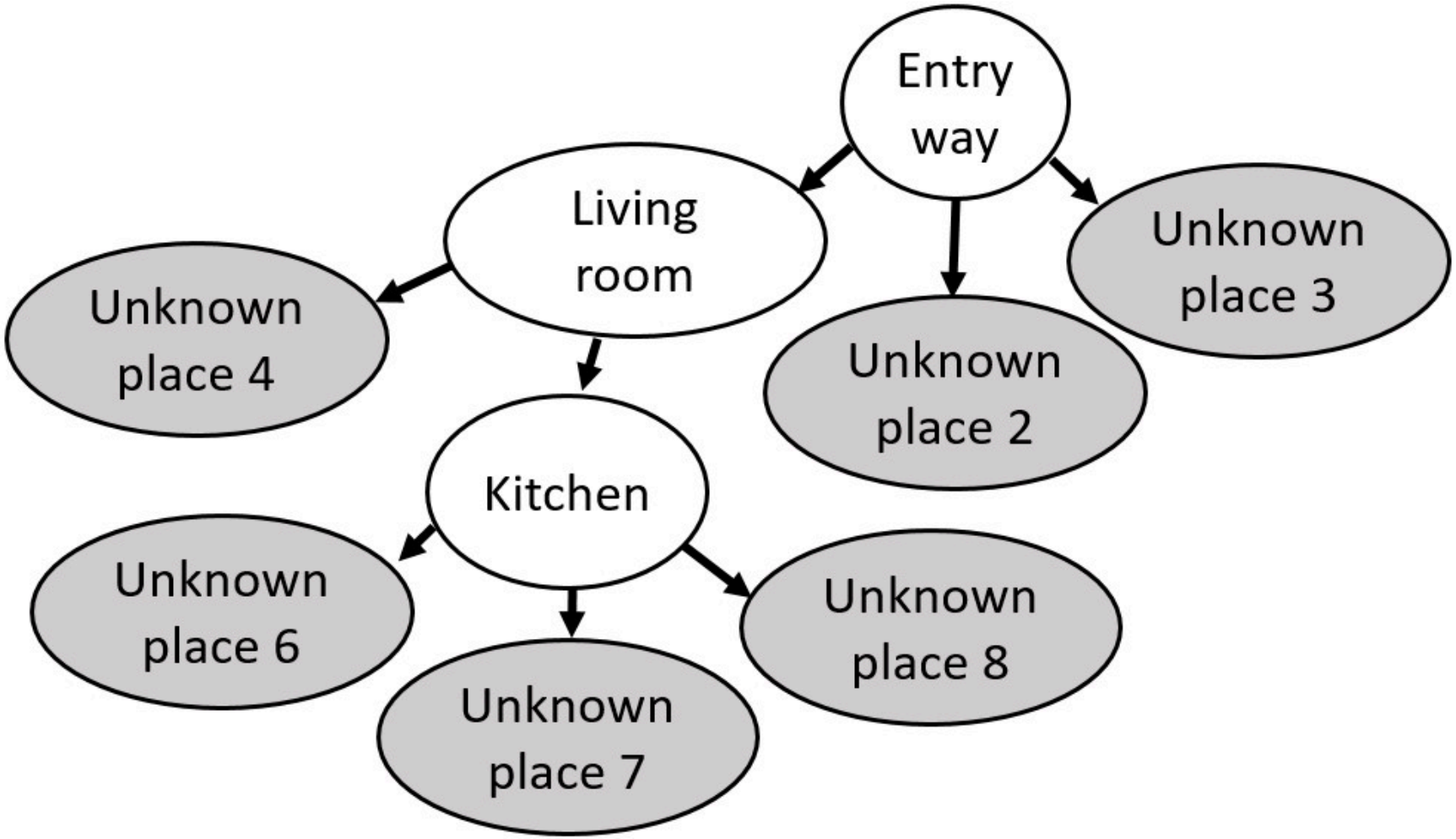
Let's say you're exploring a place...



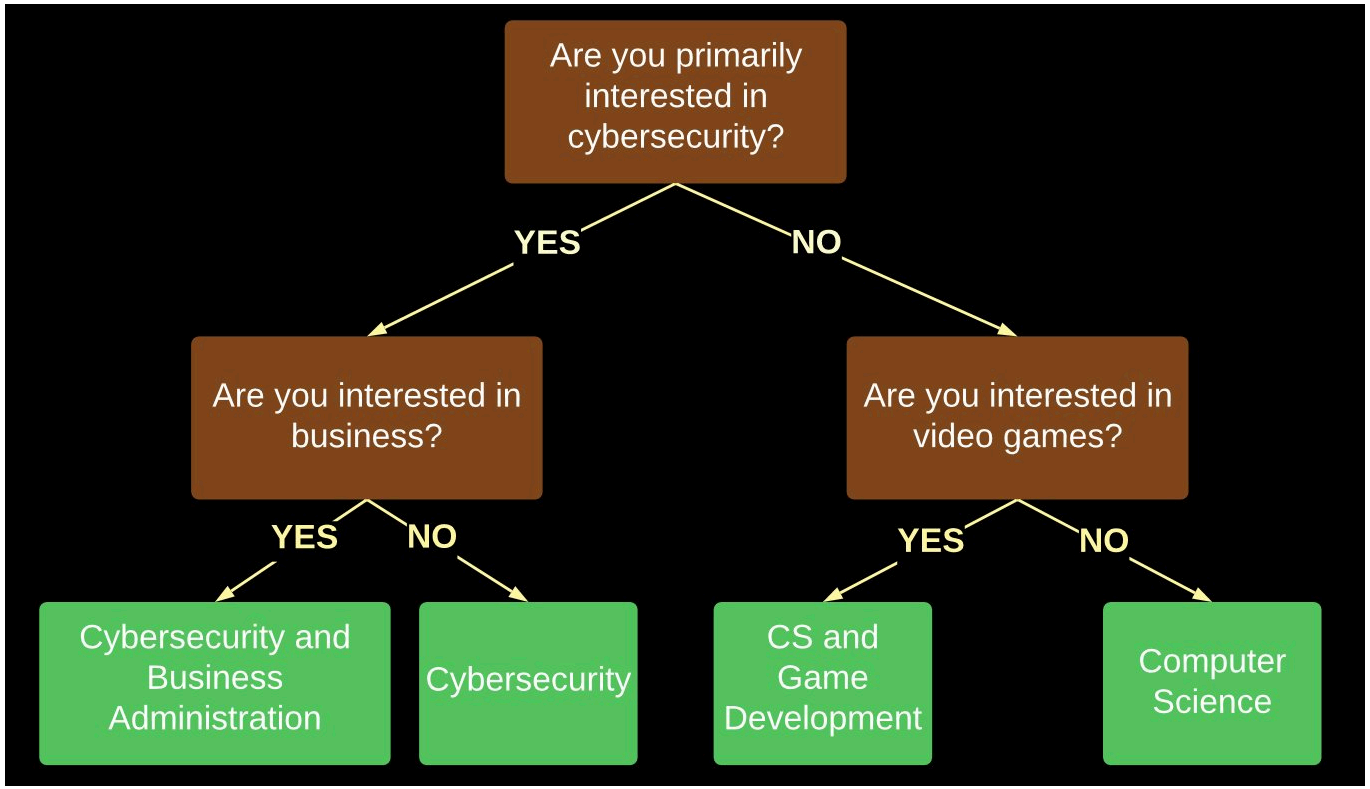
Let's say you're exploring a place...



Let's say you're exploring a place...



You may have seen trees like this decision tree:



Some tree terminology / rules

- Each *node* may have data
- There is a *root* node (the start)
- Each node points to any number of *child* nodes
- Cycles are not permitted
- There are any number of *leaf* nodes (node with no children)

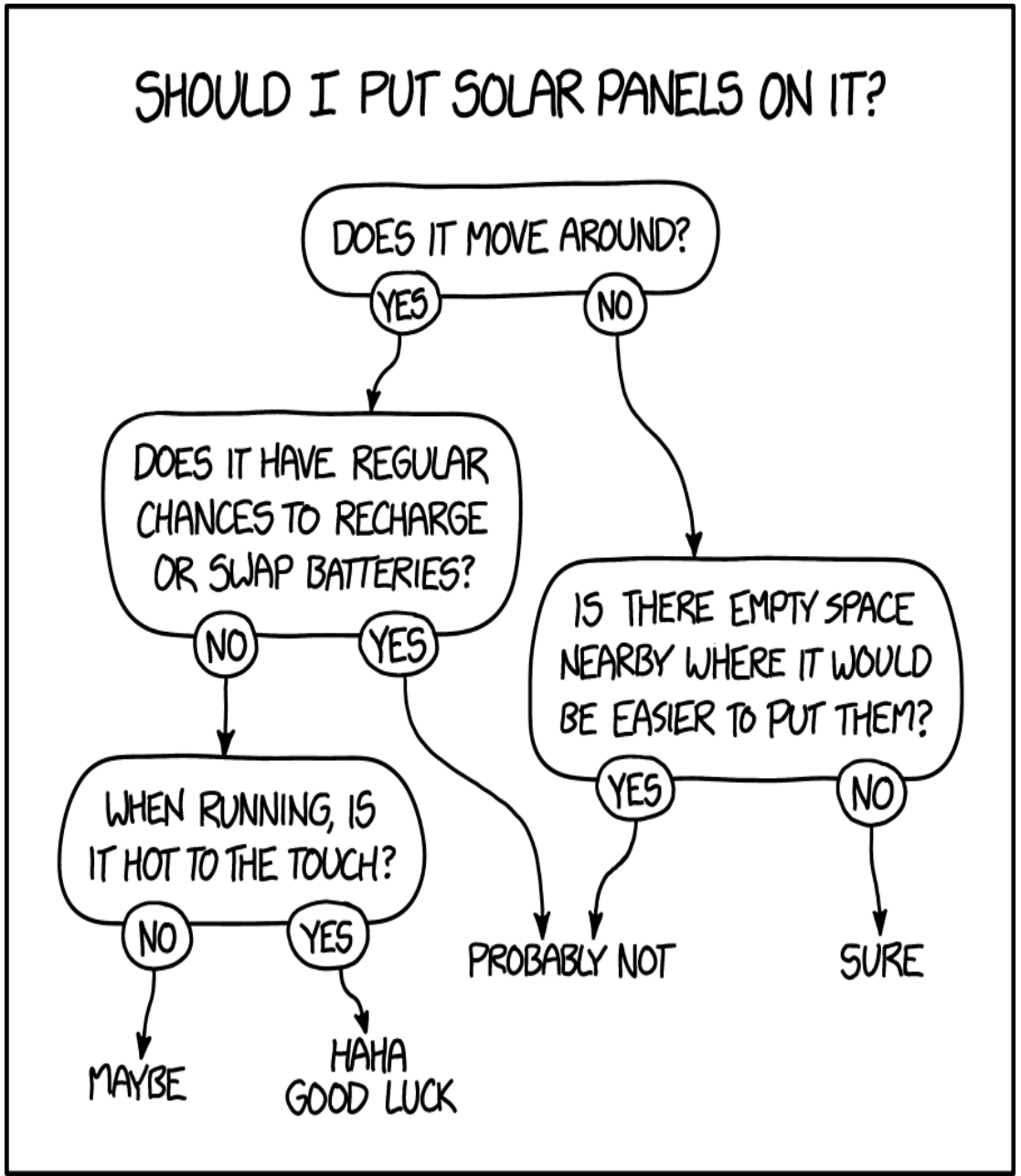
Source: Reddit

Note: In computer science, we draw our trees with the root at the top and the leaves at the bottom.

Poll: Is this a tree?

Source: https://www.canva.com/graphs/decision-trees/

1. Yes
2. No



Source: https://xkcd.com/1924/

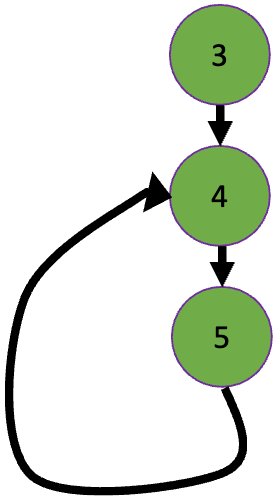
Poll: Is this a tree?

1. Yes
2. No

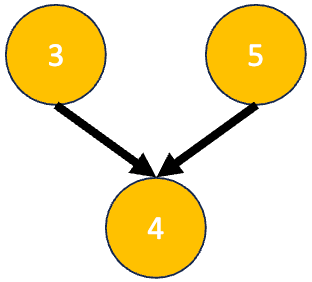
Poll: Which of these are trees?

A

B



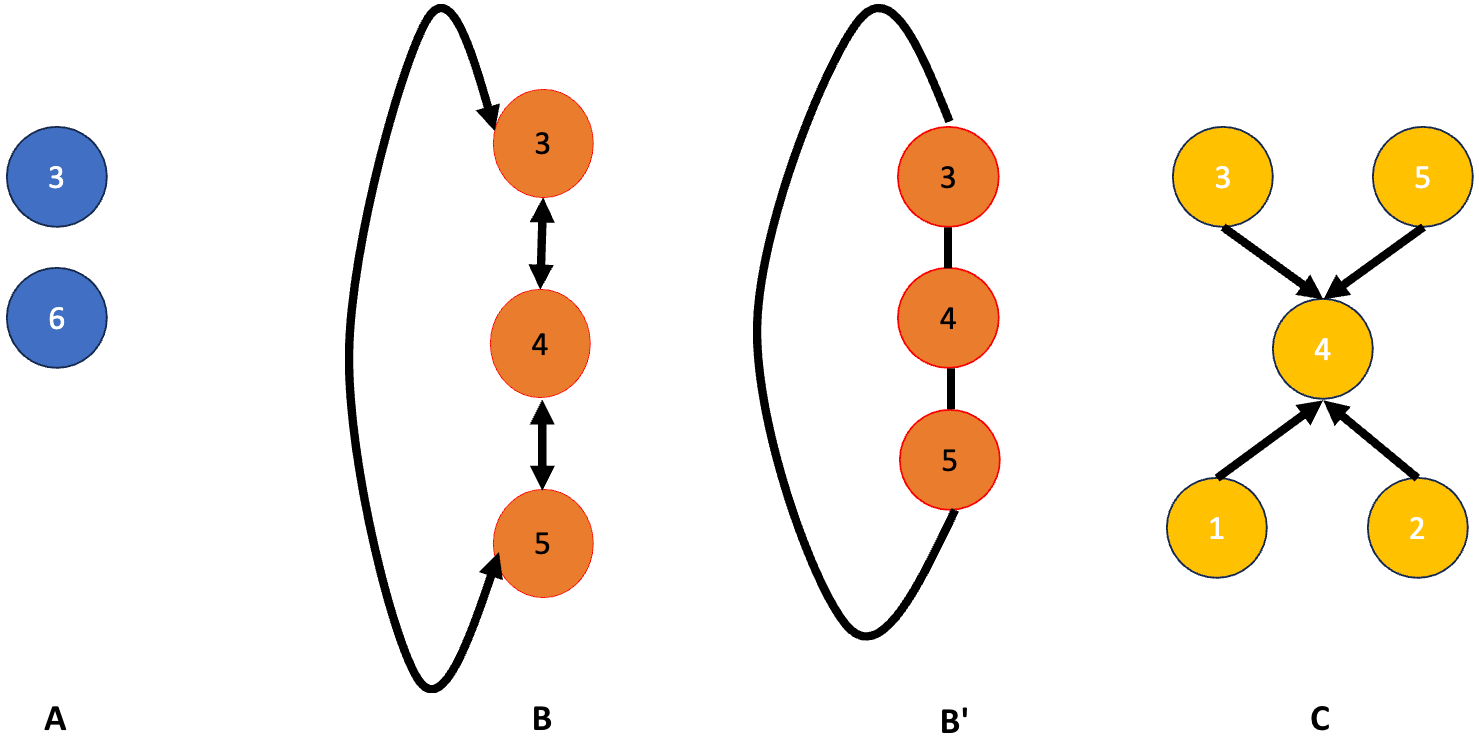
C



D

1. A
2. B
3. C
4. D

Poll: Which of these are trees? (B and B' are equivalent.)



1. A
2. B and B'
3. C

```
T = TypeVar('T')

class Node(Generic[T]):
    def __init__(self, data: T):
        self.data = data
        self.left: Optional[Node[T]] = None
        self.right: Optional[Node[T]] = None

    def __eq__(self, other: object) -> bool:
        if not isinstance(other, Node):
            raise NotImplementedError
        return self.data.__eq__(other.data)

    def __str__(self) -> str:
        value: str = f'{self.data}'
        if self.left is not None:
            value += f' {self.left}'
        else:
            value += '  *'
        if self.right is not None:
            value += f' {self.right}'
        else:
            value += '  *'
        return f'({value})'
```

```
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__()

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(tree)
```

```
(Entry way (Living room  * (Kitchen  *  *))  *)
```

```
(Entry way (Living room  * (Kitchen  *  *))  *)
```

Split that printed output into multiple lines:

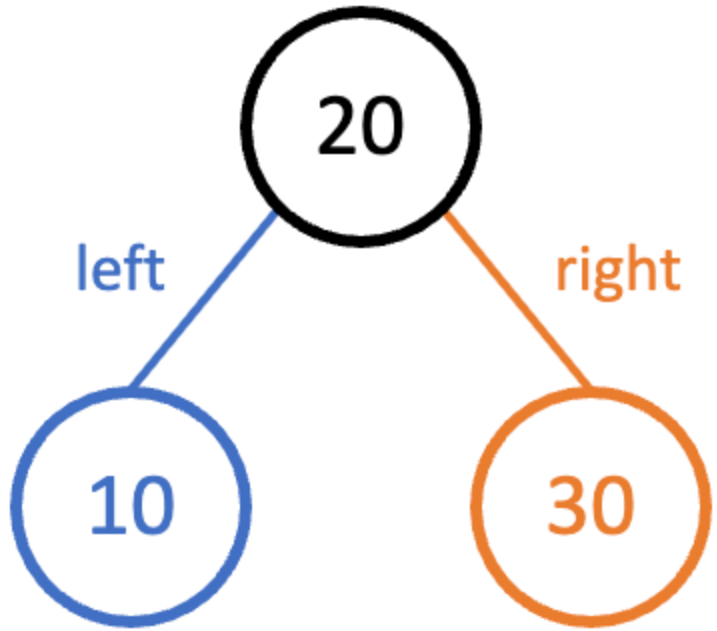
```
(Entry way
           (Living room
                       *
                       (Kitchen  *  *))
           *
)
```

Binary Search Trees

Binary Tree: A tree in which each node has at most 2 children

The first and second child of a node are called the left and right child, respectively.

Binary Search Tree: A binary tree in which each node's data is greater than everything in its left subtree and less than everything in its right subtree



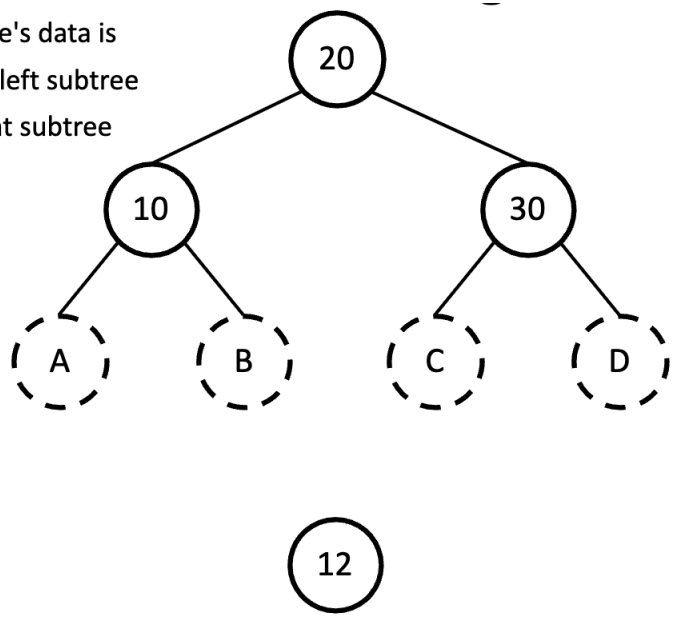
A binary tree in which each node's data is

- greater than everything in its left subtree
- less than everything in its right subtree

Poll: This is a Binary Search Tree. Where should the 12 go?

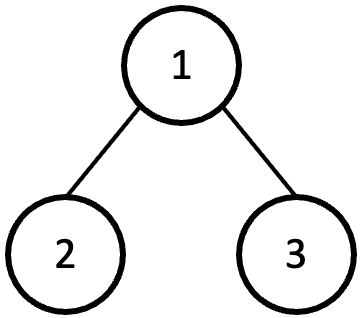
A binary tree in which each node's data is

- greater than everything in its left subtree
- less than everything in its right subtree

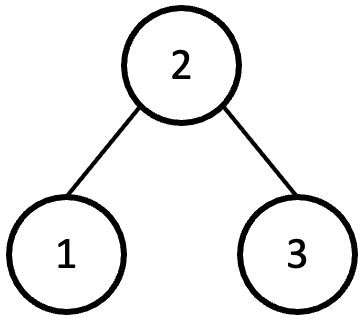


1. A
2. B
3. C
4. D

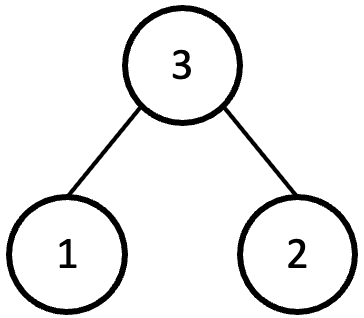
Poll: Which of these are Binary Search Trees?



A



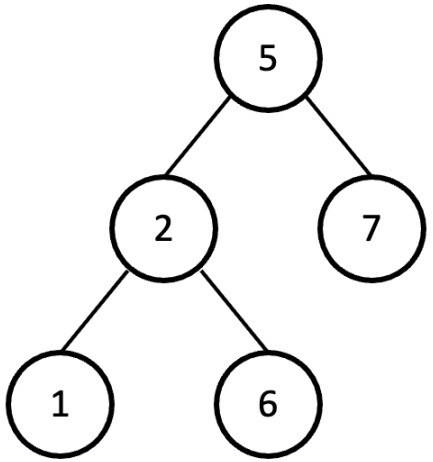
B



C

1. A
2. B
3. C

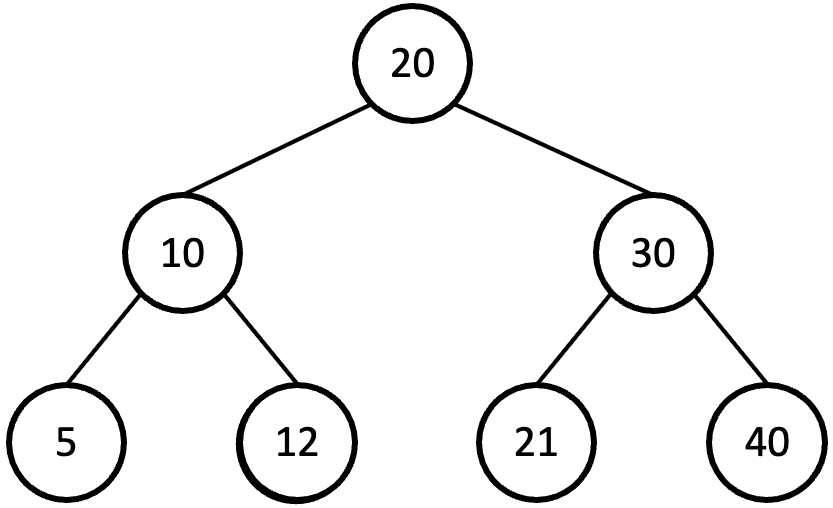
Poll: Is this a BST?



1. Yes
2. No

What's a Binary Search Tree good for?

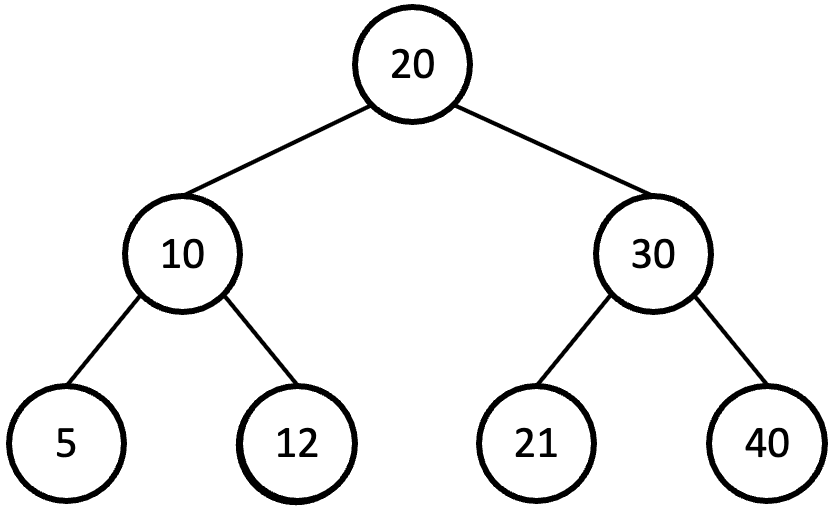
Let's say I want to check for whether the tree contains 12. How do we do that?



What about searching for 11 (with the same BST)?

What's a Binary Search Tree good for?

Let's say I want to check for whether the tree contains 12. How do we do that?



What about searching for 11 (with the same BST)?

Hey, this is more efficient than searching a list!

It's not as efficient as hashing, but it does make a pretty good set.

And it's more useful if you care about the order of things.

`sortedcontainers.SortedSet` stores elements using a Binary Search Tree

(`pip install sortedcontainers`)

```
ss = SortedSet([3, 1, 4, 1, 5])
print(ss)  # SortedSet([1, 3, 4, 5])
ss.add(2)
print(ss)  # SortedSet([1, 2, 3, 4, 5])
```

Corresponding `SortedDict` :

```
sm = SortedDict({2: [1, 2, 3], 1: [0, 0, 0]})
print(sm)  # SortedDict({1: [0, 0, 0], 2: [1, 2, 3]})
```

`set`	`sortedcontainers.SortedSet`
Stored as a hash table (list of lists) Index of each element is calculated using `__hash__()`	Stored as a Binary Search Tree Elements must implement Comparable protocol
Constant time to look up / add / remove	Logarithmic time to look up / add / remove
Use when care more about speed than order	Use when care more about order than speed

Poll: Which are true?

1. For a `SortedDict`, it is constant time to check whether it contains a key
2. For a `SortedDict`, it is constant time to check whether it contains a value
3. `set`s can store things which don't implement the Comparable protocol
4. `set`s can store things which aren't hashable
5. When we iterate over a `set`, the elements will be increasing in size
6. When we iterate over a `SortedSet` the elements will be increasing in size

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

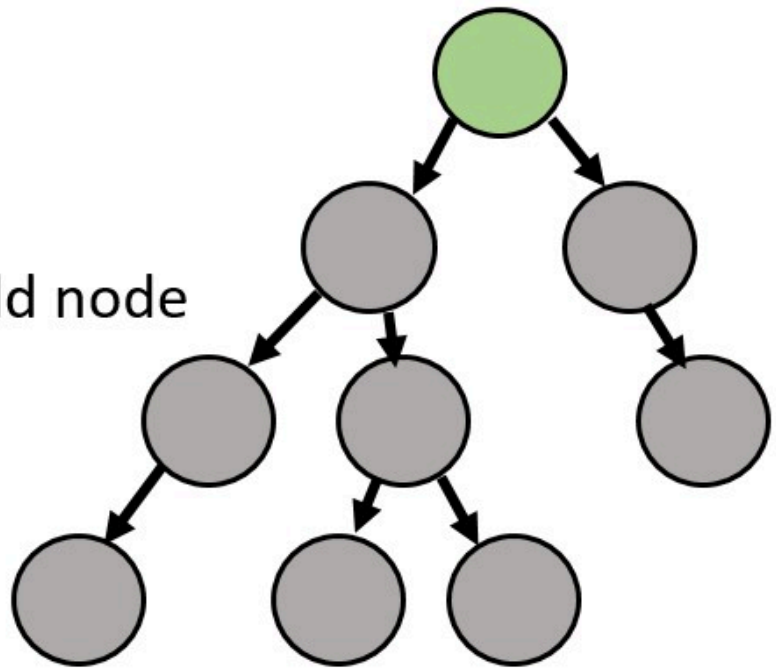
```
class Tree(Generic[T]):
    def __init__(self, root_data: Optional[T] = None) -> None:
        if root_data is None:
            self.root: Optional[Node[T]] = None
        else:
            self.root = Node[T](root_data)

    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

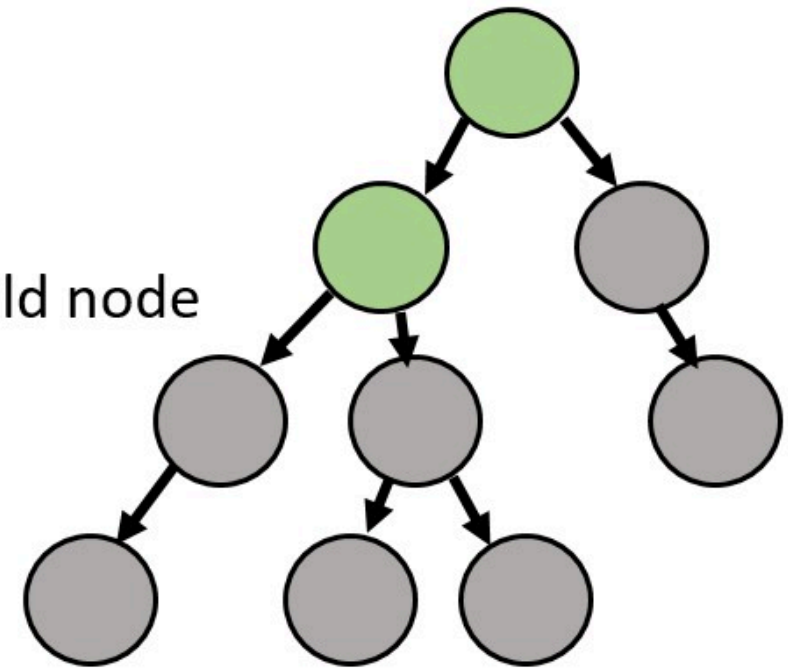


Recursive backtracking

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

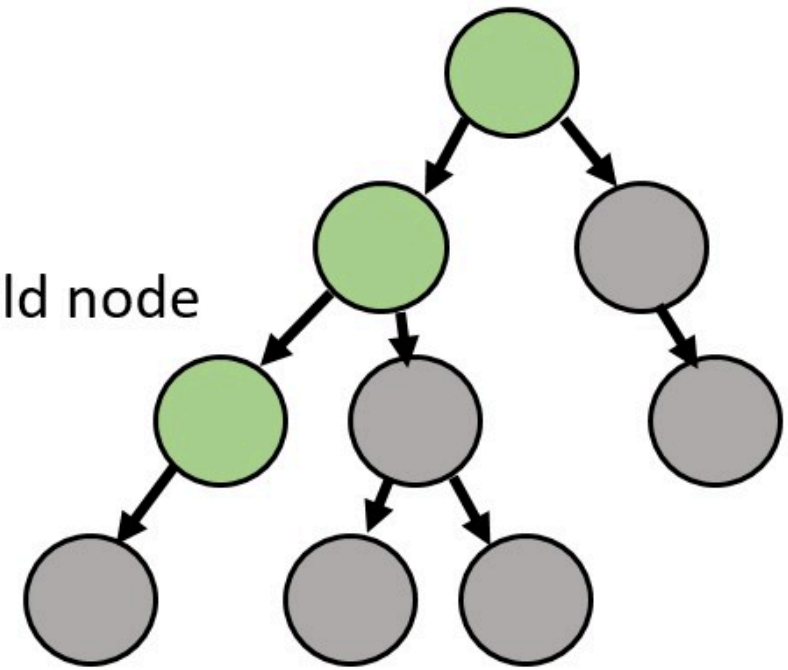


Recursive backtracking

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

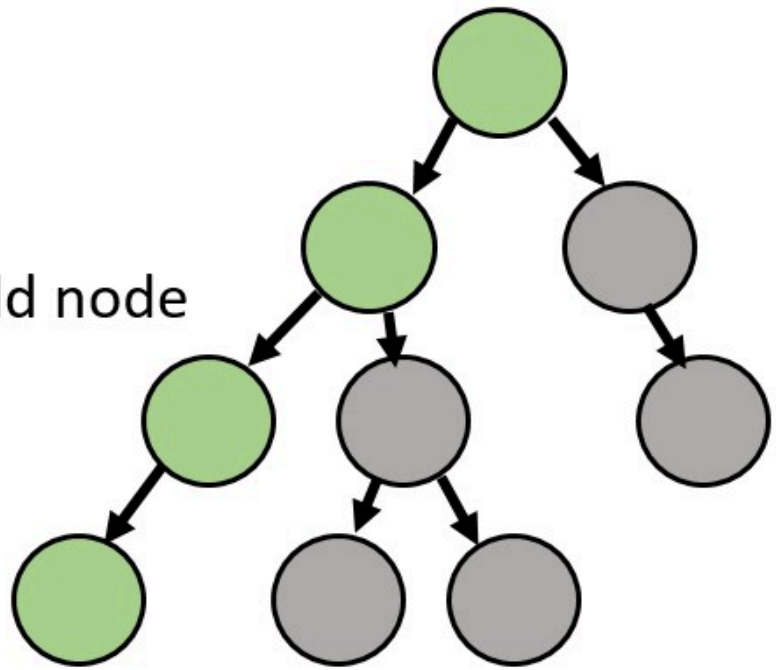


Recursive backtracking

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

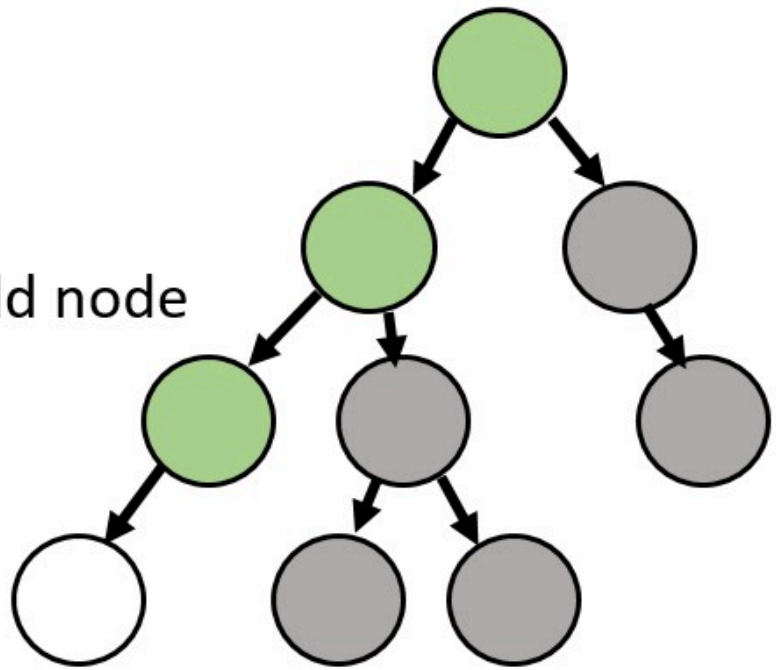


Recursive backtracking

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

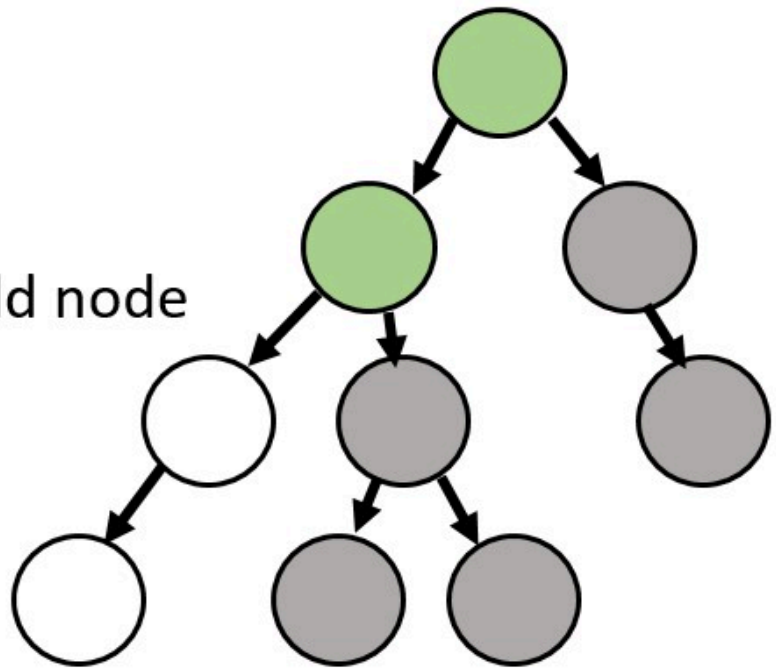


Recursive backtracking

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

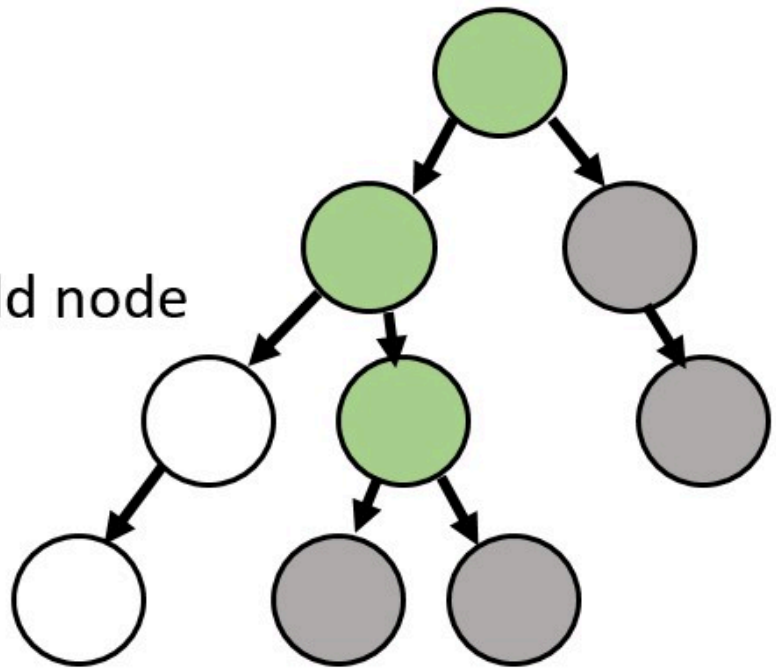


Recursive backtracking

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

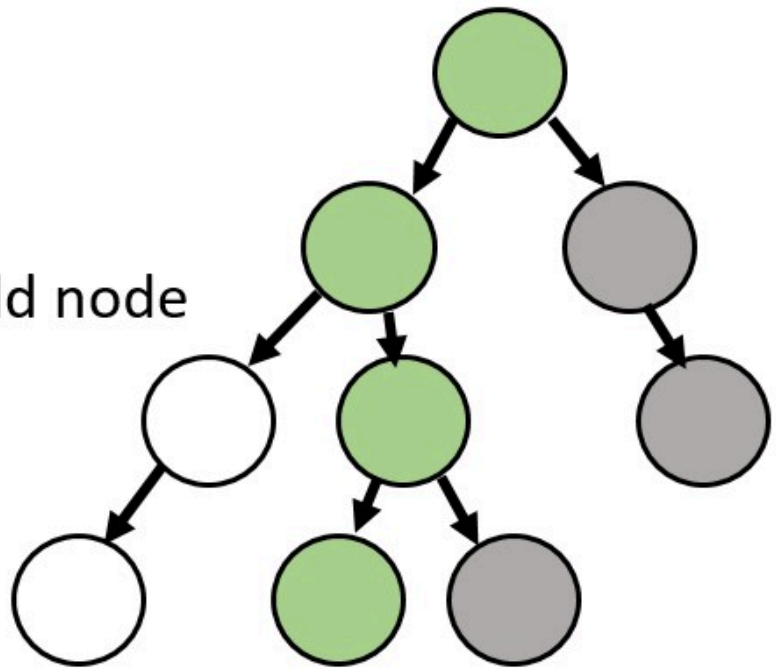


Recursive backtracking

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

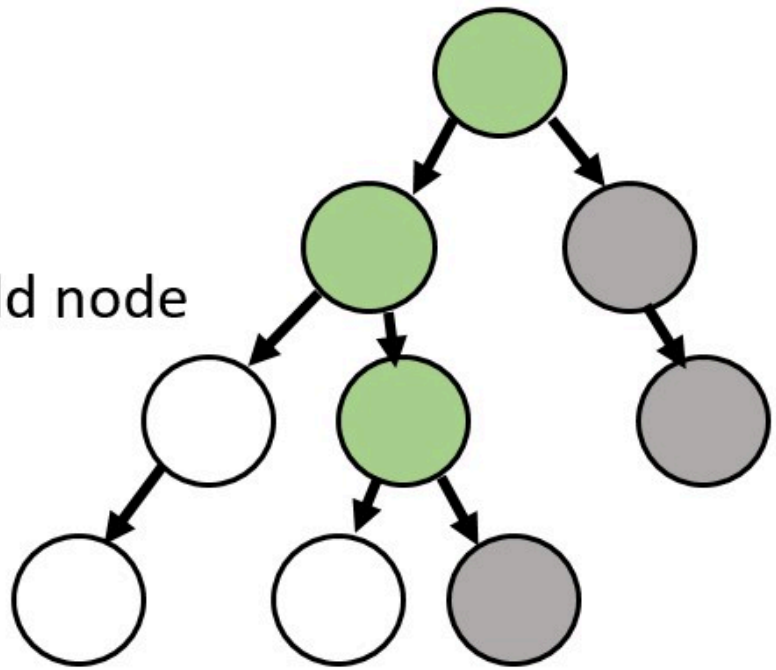


Recursive backtracking

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

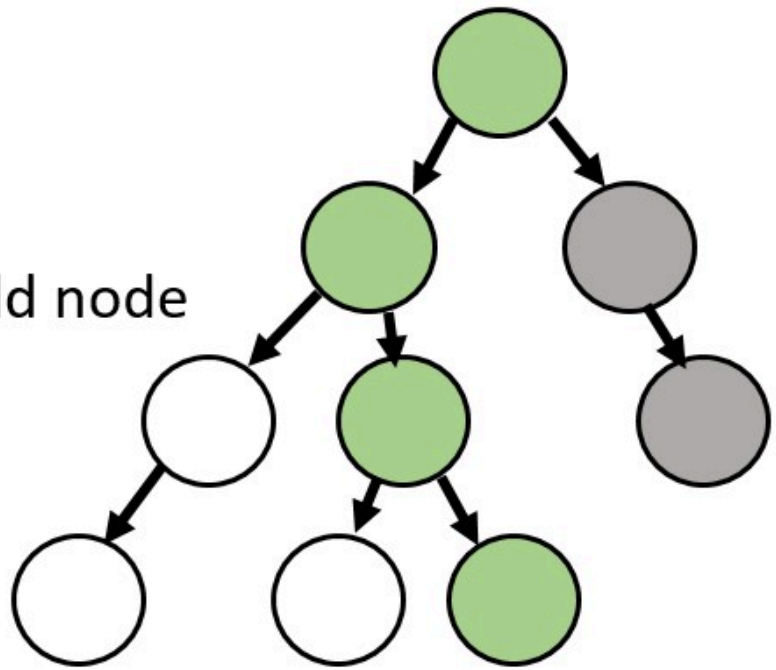


Recursive backtracking

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

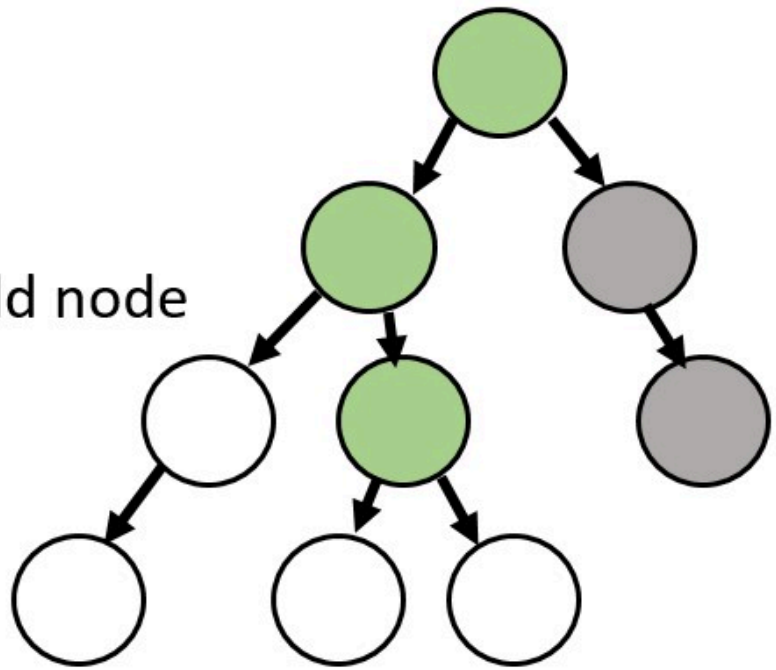


Recursive backtracking

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

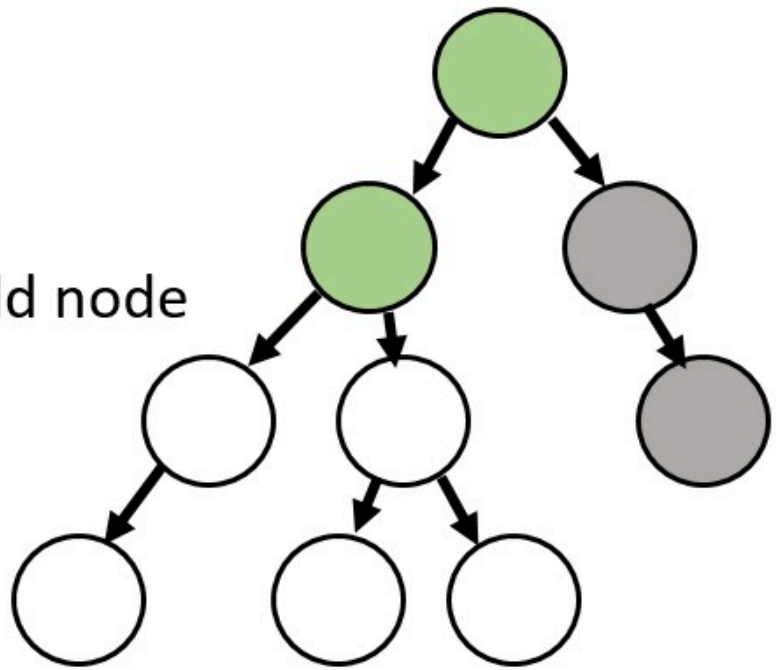


Recursive backtracking

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

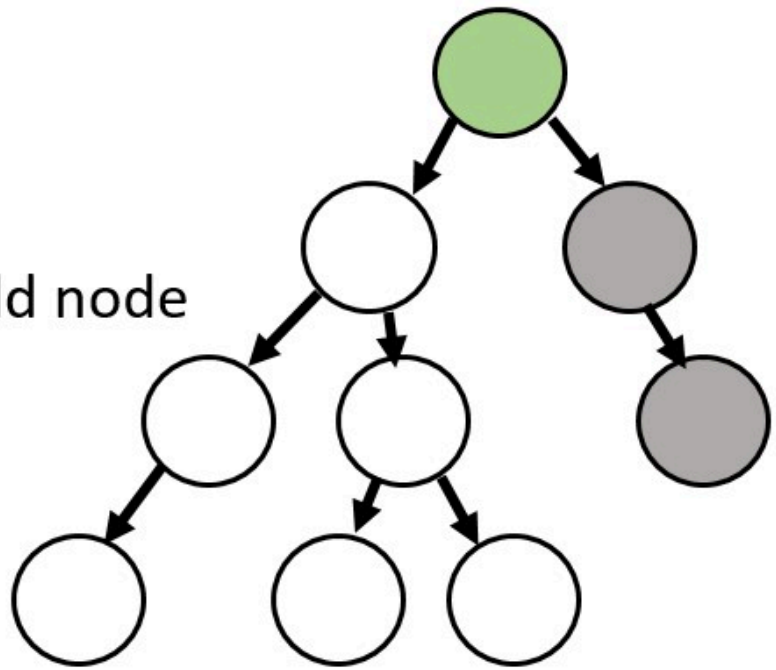


Recursive backtracking

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

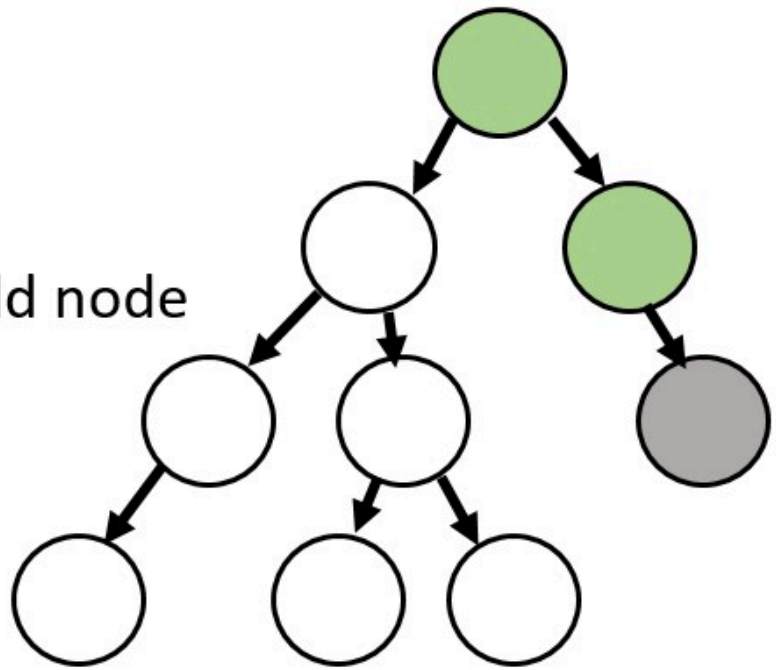


Recursive backtracking

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

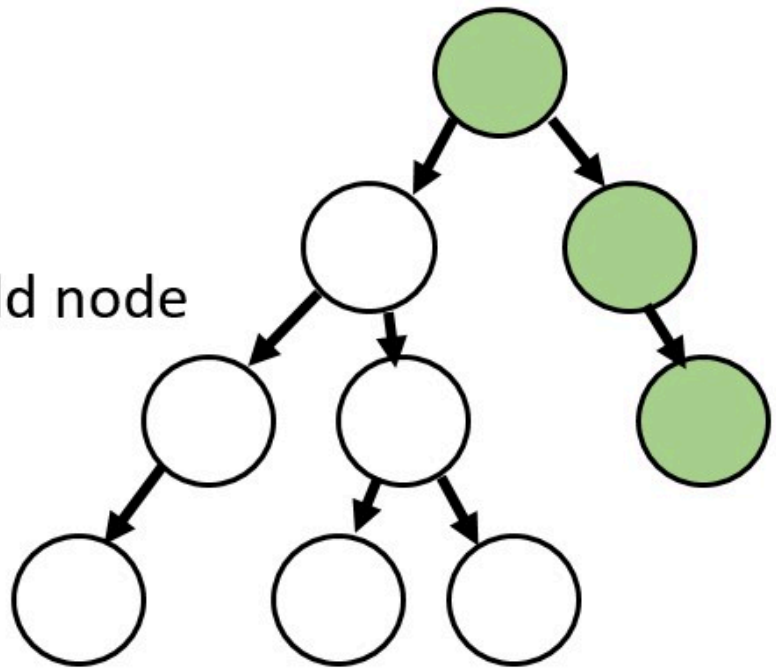


Recursive backtracking

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

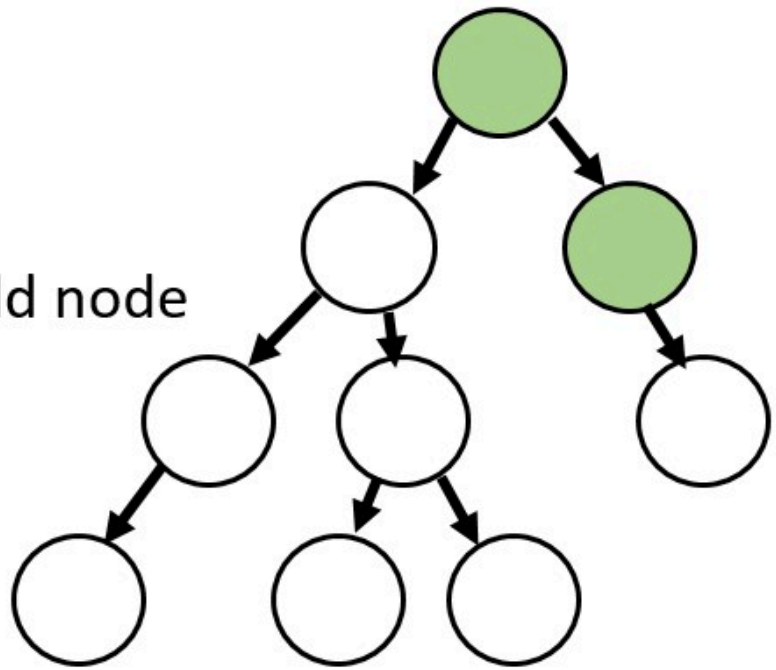


Recursive backtracking

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

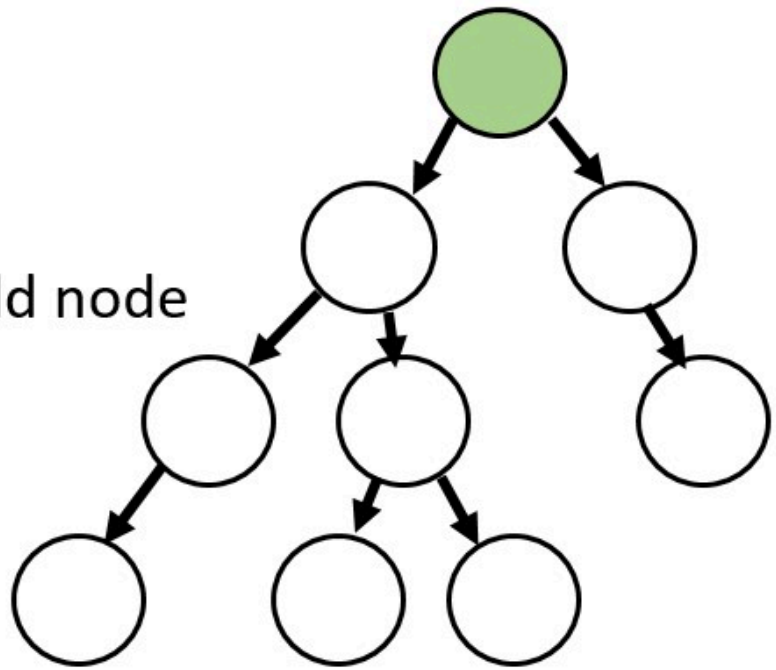


Recursive backtracking

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

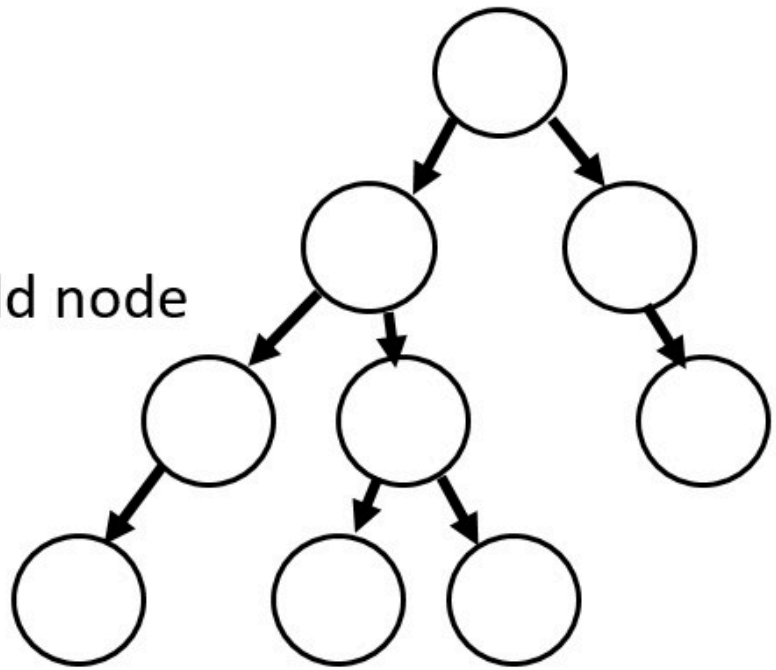


Recursive backtracking

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

Poll:

1. What is your main takeaway from today?

2. What would you like to revisit next time?