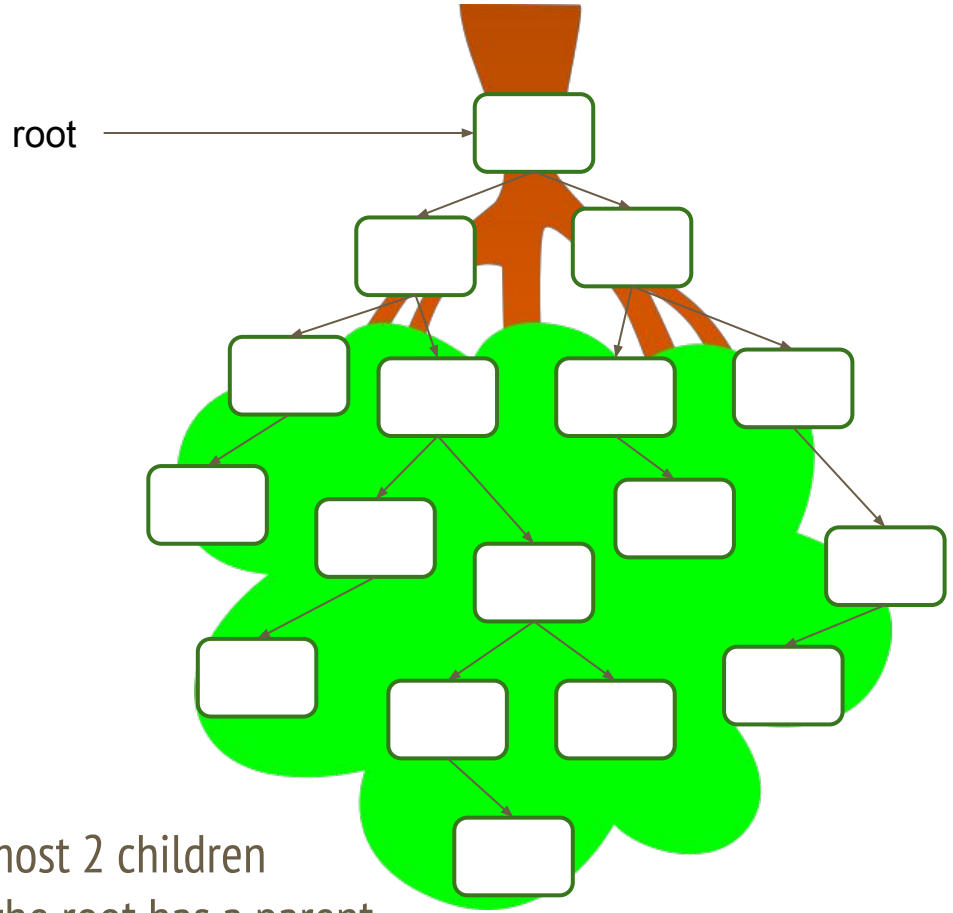
Binary Search Trees

CSCI 60, Fall 2025

Sara Krehbiel, 12/3/25-12/5/25



BINARY TREE:

Each node has at most 2 children

Each node except the root has a parent

TERMS:

Graph theory

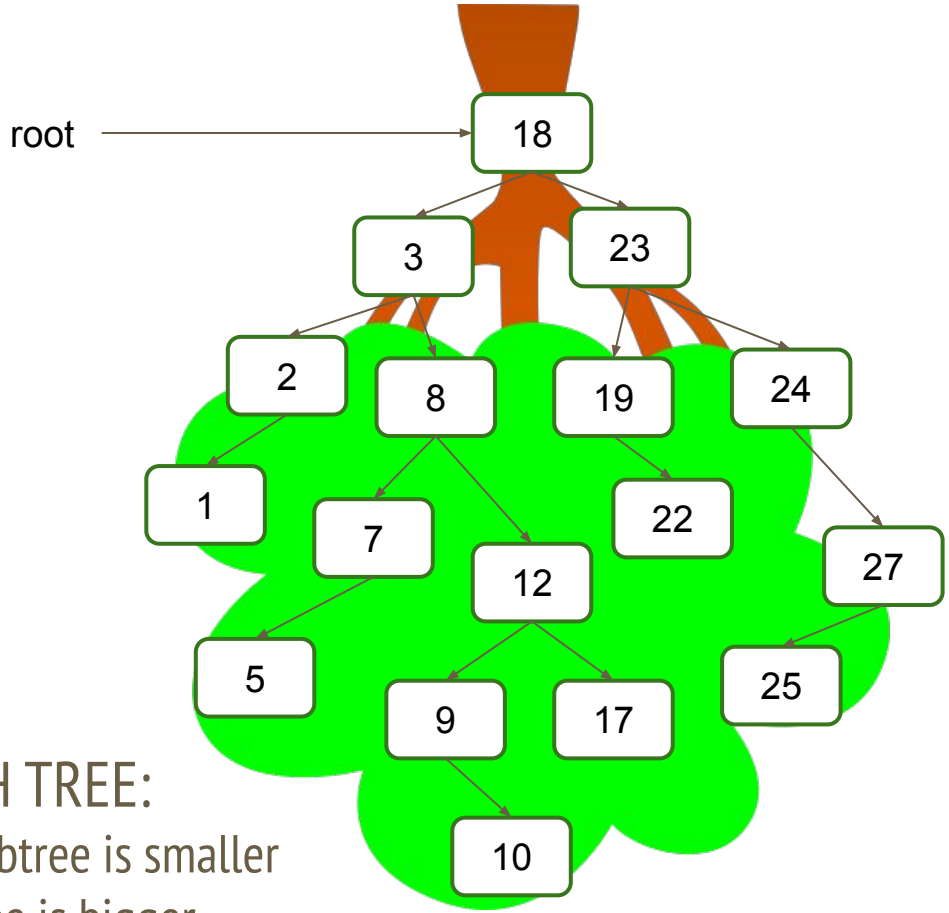
- Node / vertex
- Edge
- Size
- Depth

Botany

- Root
- Leaf

Genealogy

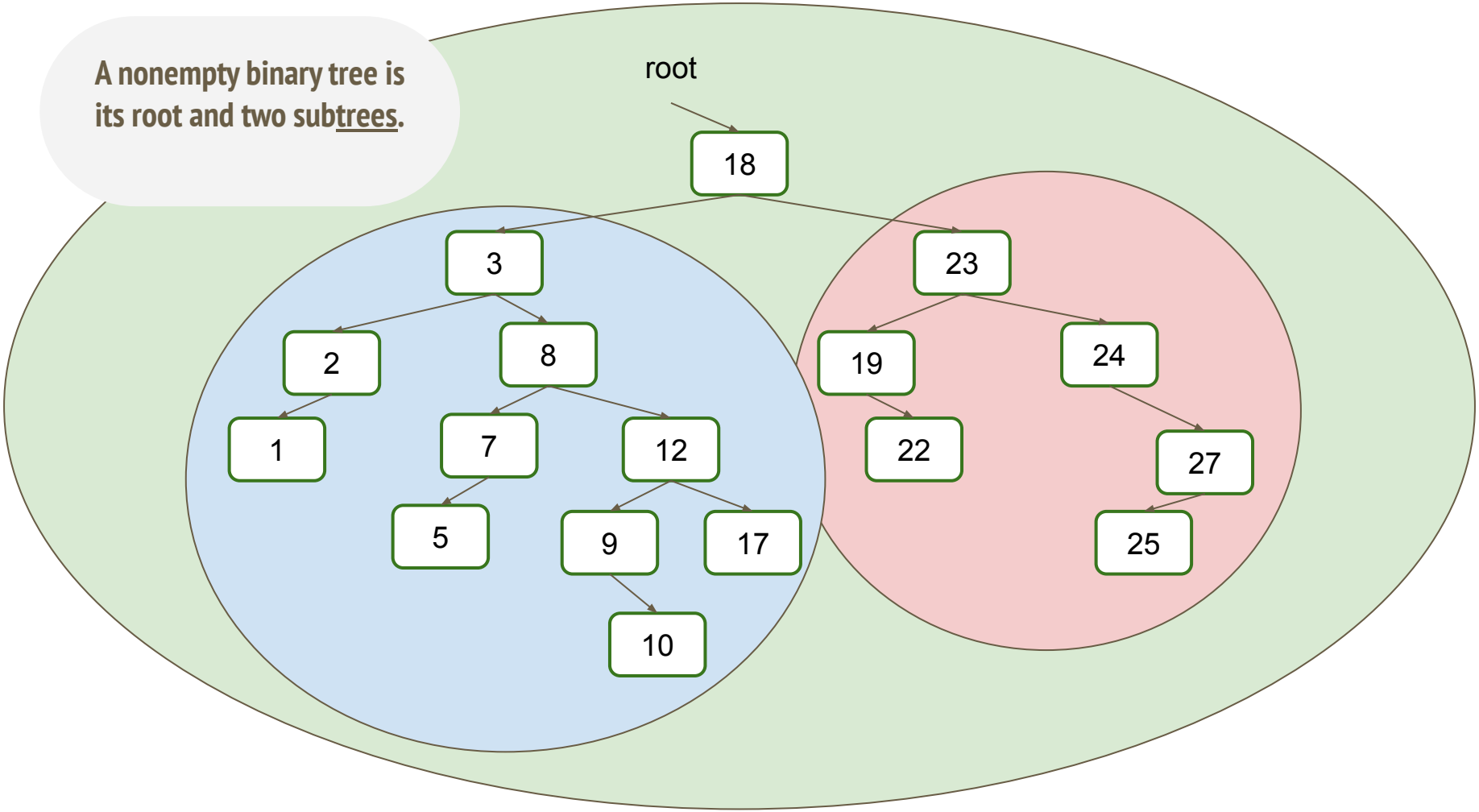
- Parent
- Child
- Ancestor

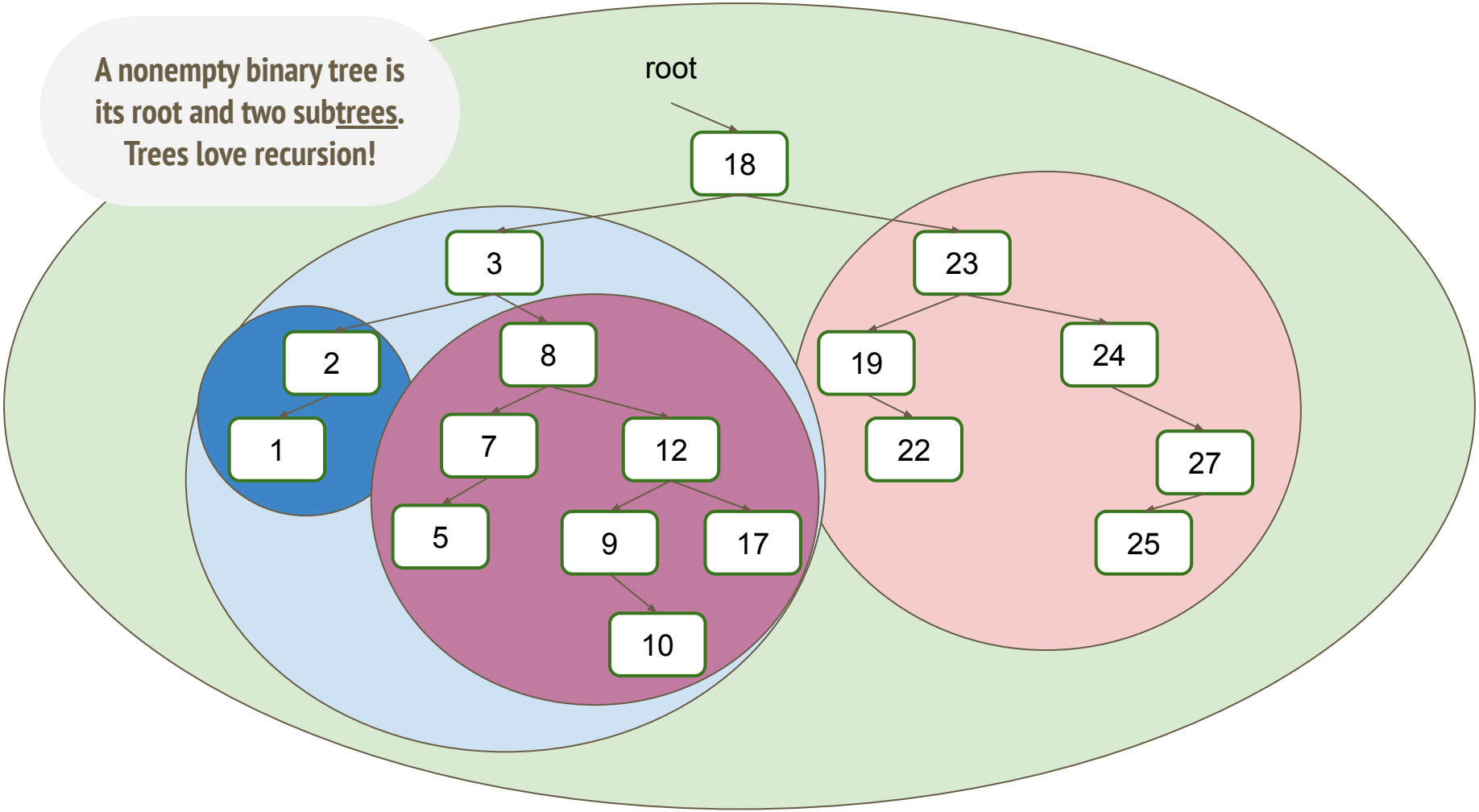


BINARY SEARCH TREE:

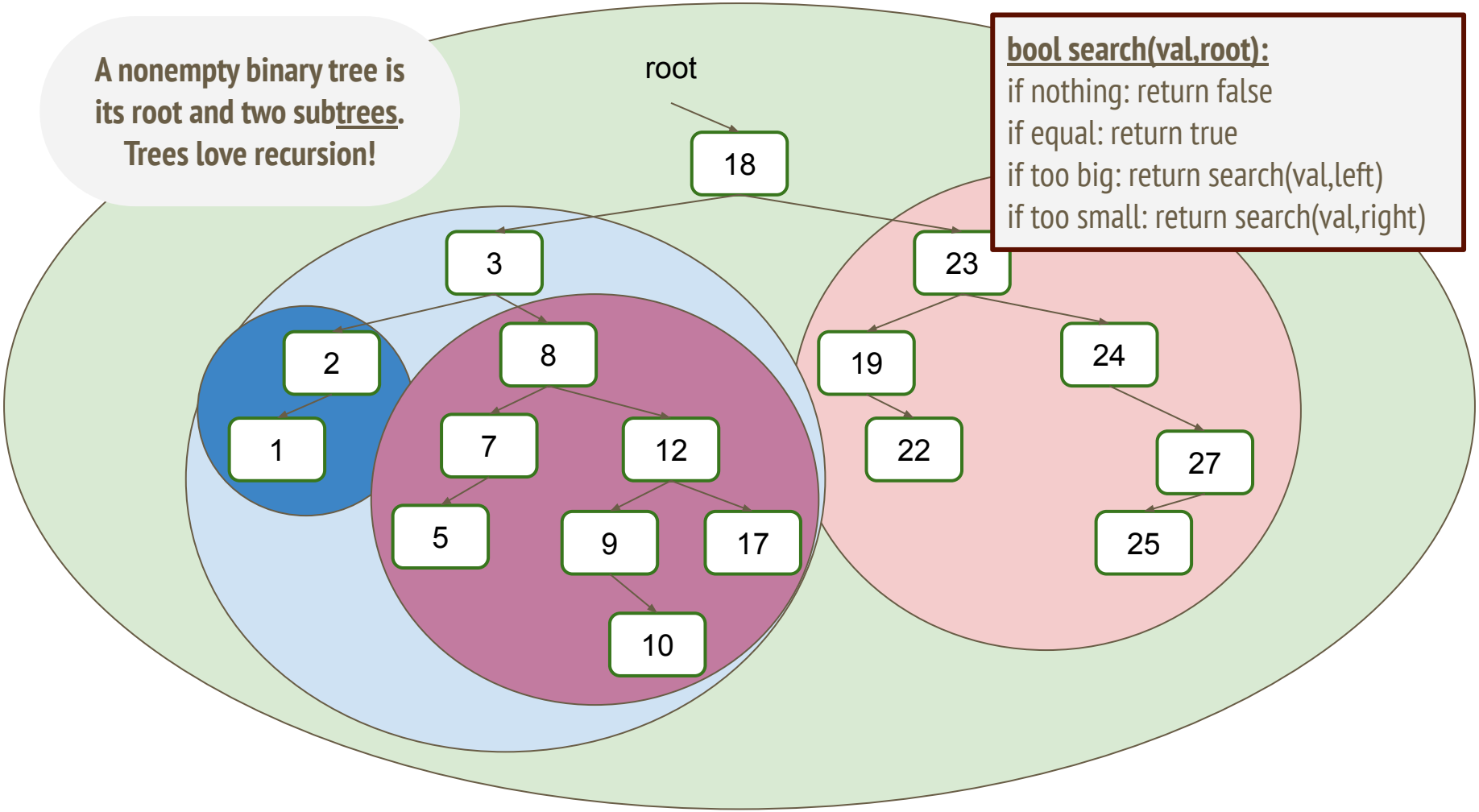
Each node's left subtree is smaller and its right subtree is bigger.

A nonempty binary tree is its root and two subtrees.

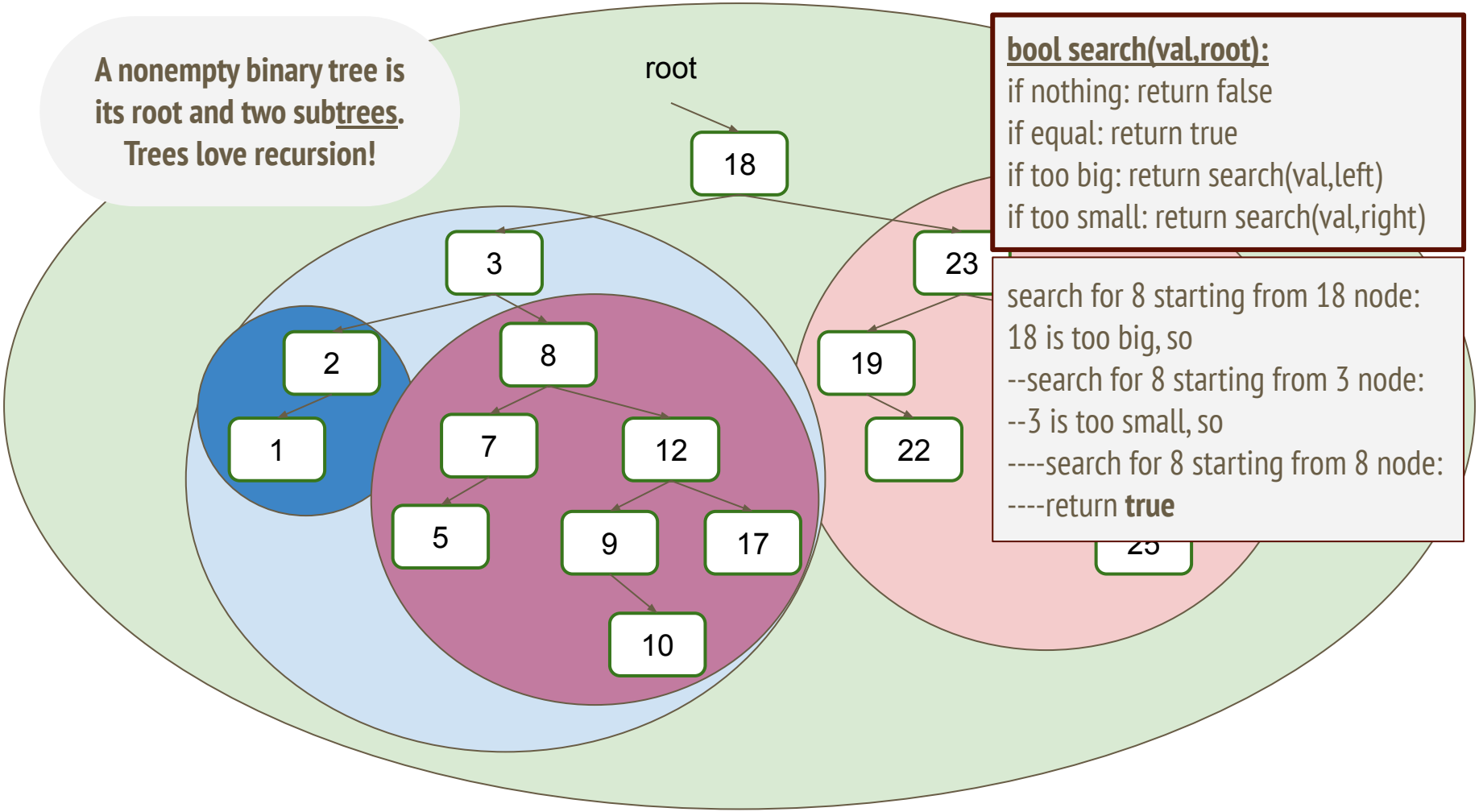



A nonempty binary tree is its root and two subtrees. Trees love recursion!
root
18
3
23
2
8
19
24
1
7
12
22
27
5
9
17
25
10

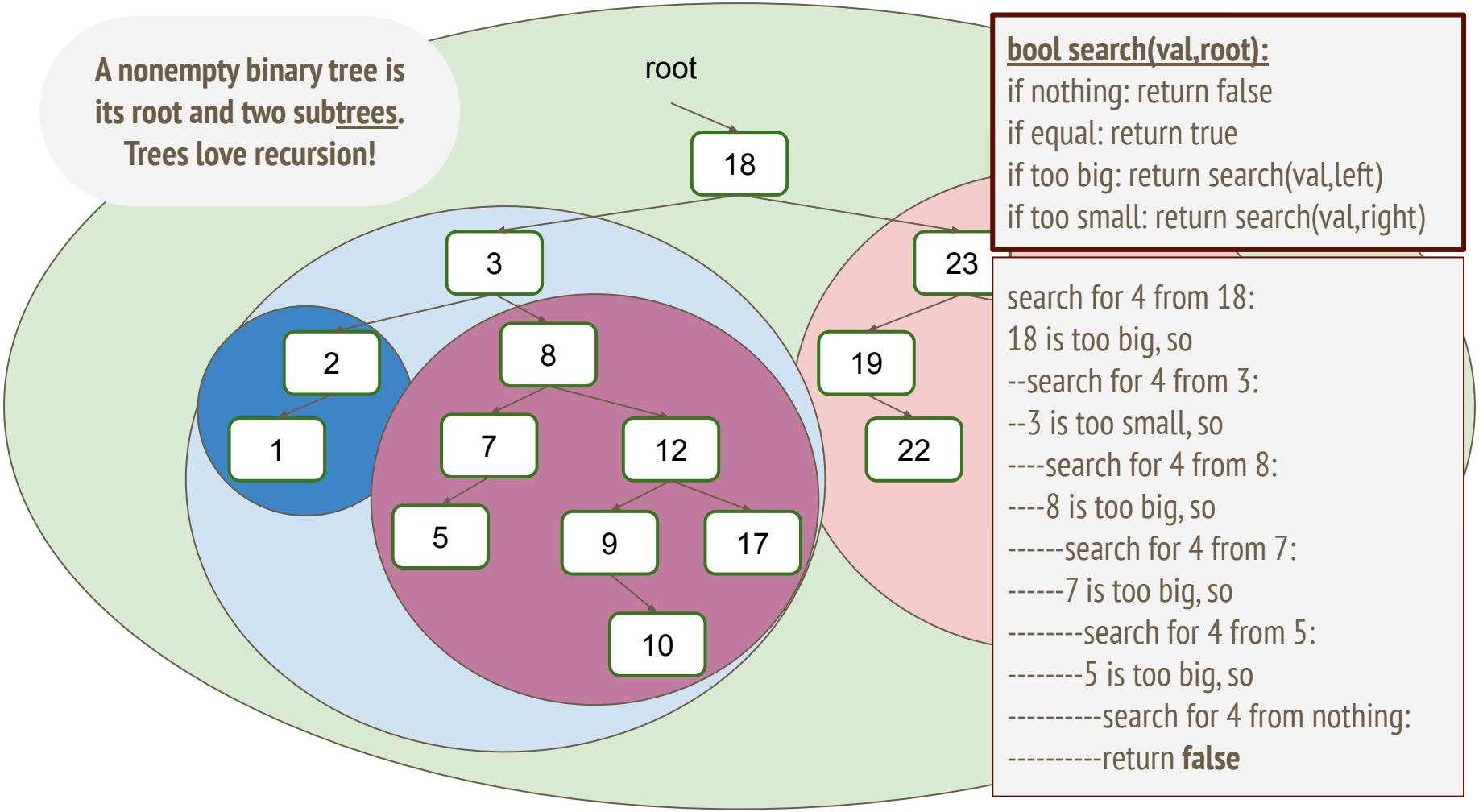
A nonempty binary tree is its root and two subtrees. Trees love recursion!



bool search(val,root):
if nothing: return false
if equal: return true
if too big: return search(val,left)
if too small: return search(val,right)

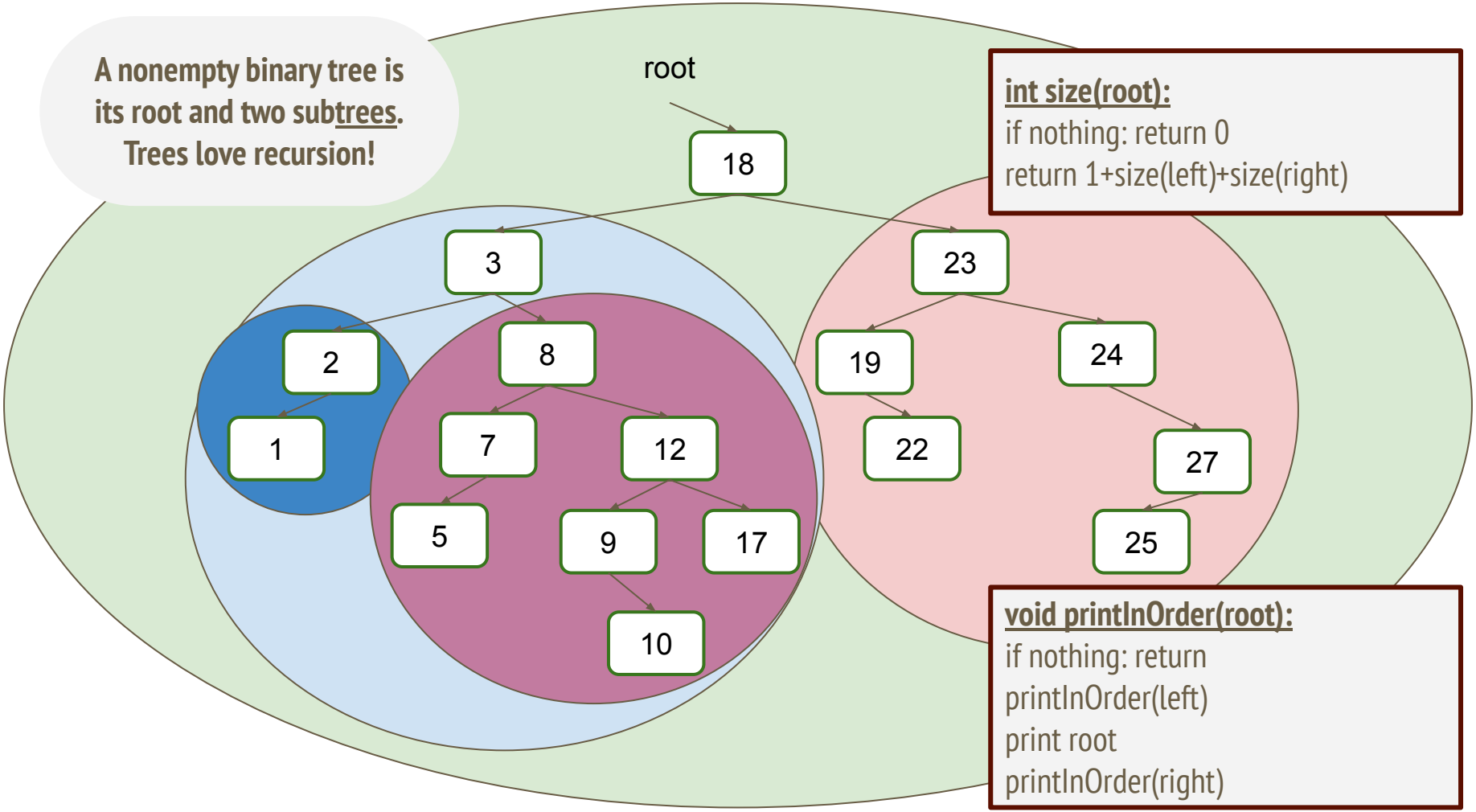

A nonempty binary tree is its root and two subtrees. Trees love recursion!
root
18
3
23
2
8
19
1
7
12
22
5
9
17
10
bool search(val,root):
if nothing: return false
if equal: return true
if too big: return search(val,left)
if too small: return search(val,right)
search for 8 starting from 18 node:
18 is too big, so
--search for 8 starting from 3 node:
--3 is too small, so
----search for 8 starting from 8 node:
----return true

A nonempty binary tree is its root and two subtrees. Trees love recursion!



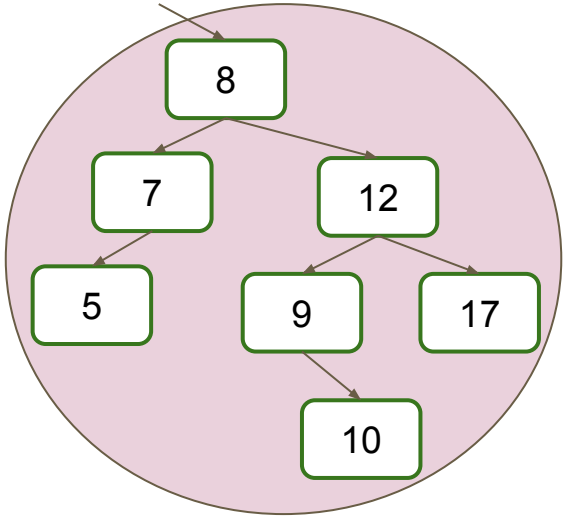
bool search(val,root):
if nothing: return false
if equal: return true
if too big: return search(val,left)
if too small: return search(val,right)

search for 4 from 18:
18 is too big, so
--search for 4 from 3:
--3 is too small, so
----search for 4 from 8:
----8 is too big, so
------search for 4 from 7:
------7 is too big, so
--------search for 4 from 5:
--------5 is too big, so
----------search for 4 from nothing:
----------return **false**

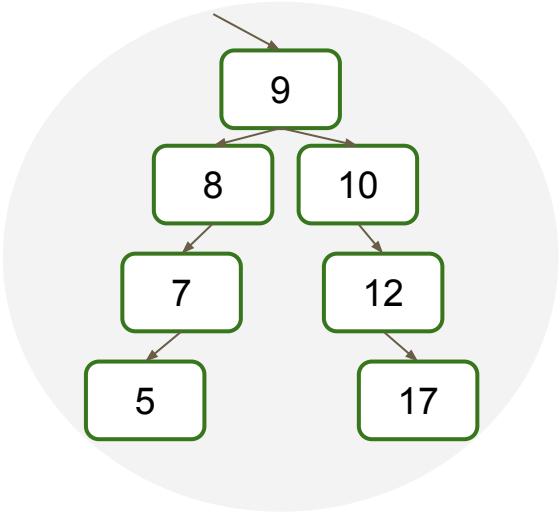

A nonempty binary tree is its root and two subtrees. Trees love recursion!
root
18
3
23
2
8
19
24
1
7
12
22
27
5
9
17
25
10
int size(root):
if nothing: return 0
return 1+size(left)+size(right)
void printInOrder(root):
if nothing: return
printInOrder(left)
print root
printInOrder(right)

Insertion order matters:
BSTs may have the same elements with different structures

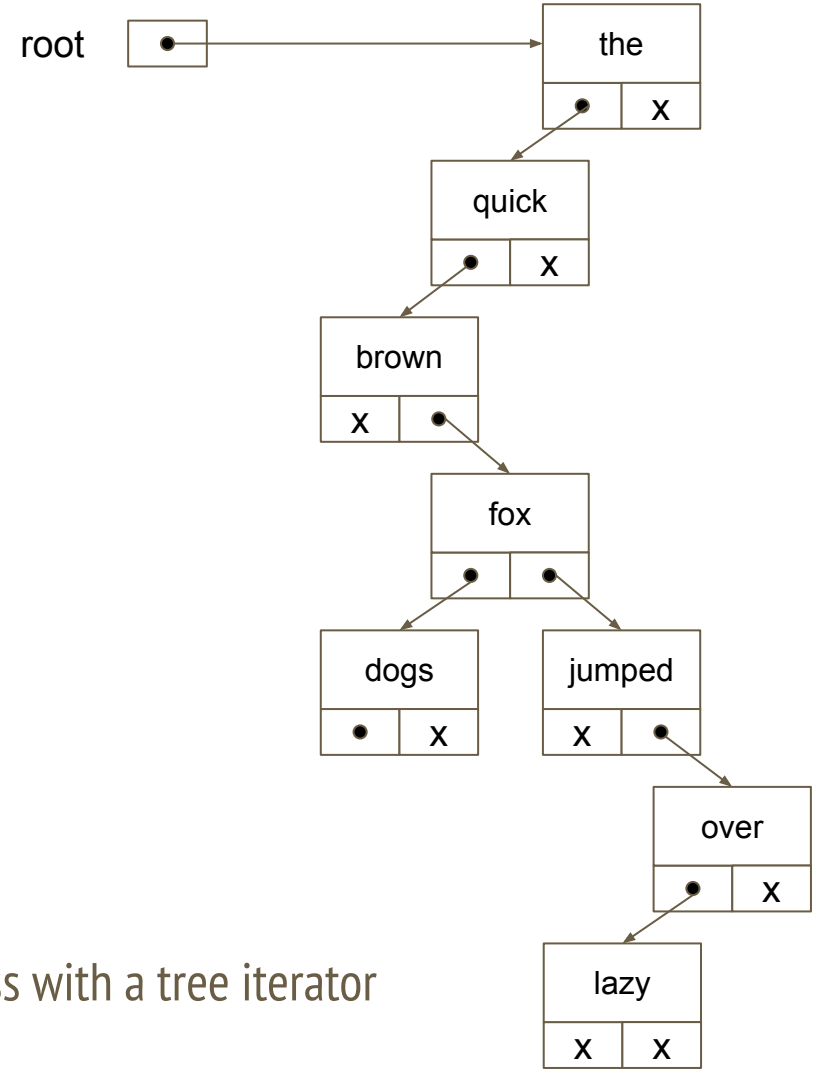
?? insert(val,root):
if nothing: new node(val)
else if too big: insert(val,left)
else if too small: insert(val,right)
else:



8, 12, 9, 7, 5, 17, 10



9, 8, 7, 5, 10, 12, 17



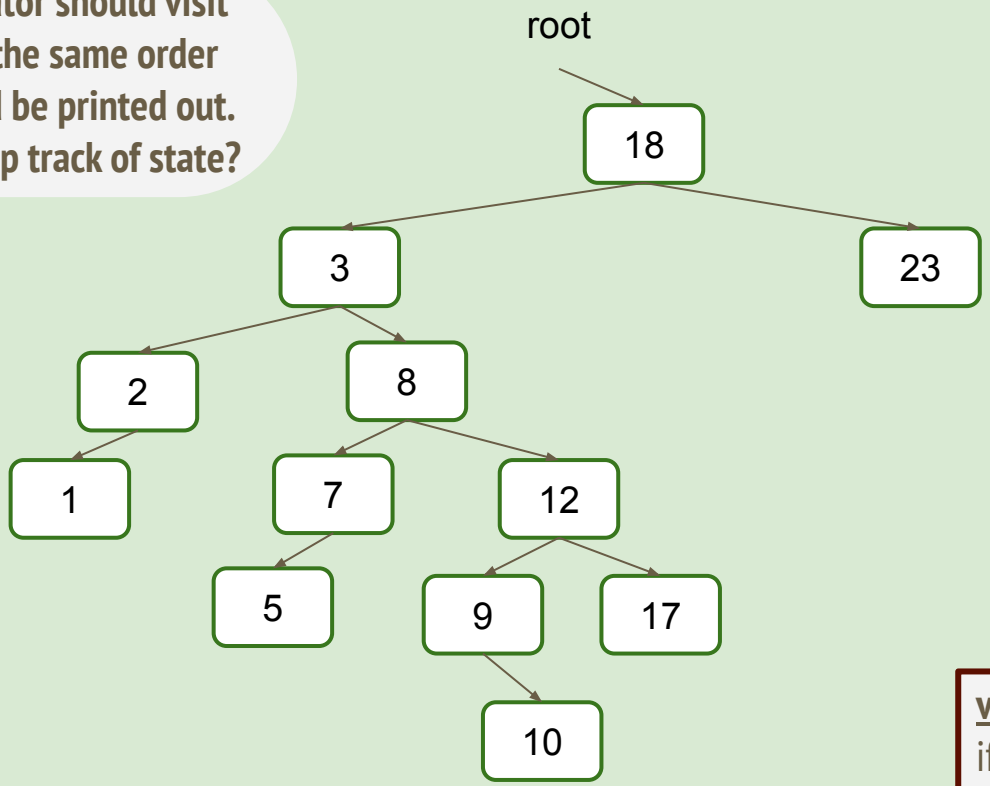

```
BST<string> words;
words.insert("the");
words.insert("quick");
words.insert("brown");
words.insert("fox");
words.insert("jumped");
words.insert("over");
words.insert("the");
words.insert("lazy");
words.insert("dogs");

for (auto word : words) {
  cout << word << ", ";
}
```

```
brown, dogs, fox, jumped,
lazy, over, quick, the,
```

GOAL:

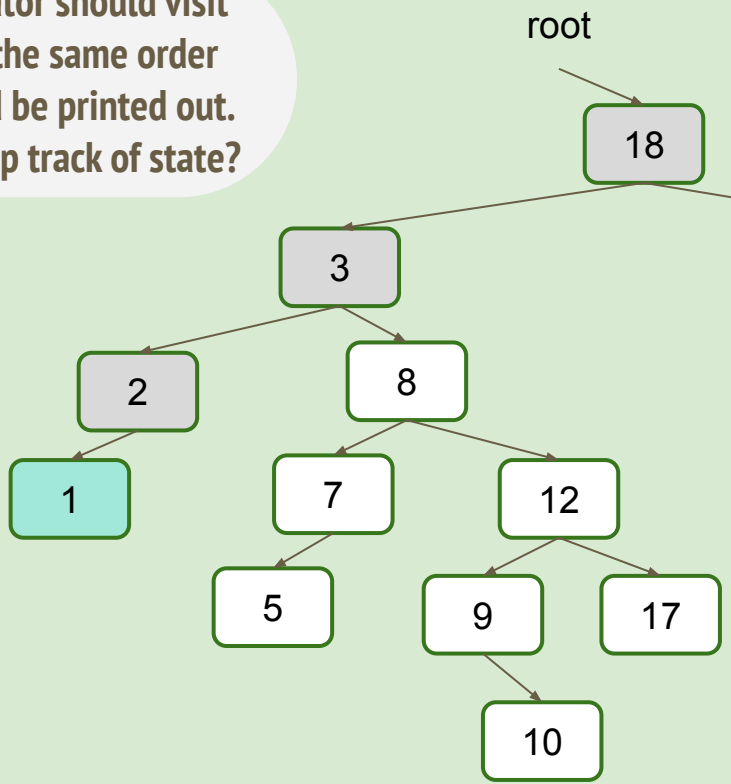
A BST class with a tree iterator

Goal: Iterator should visit nodes in the same order they would be printed out. How to keep track of state?



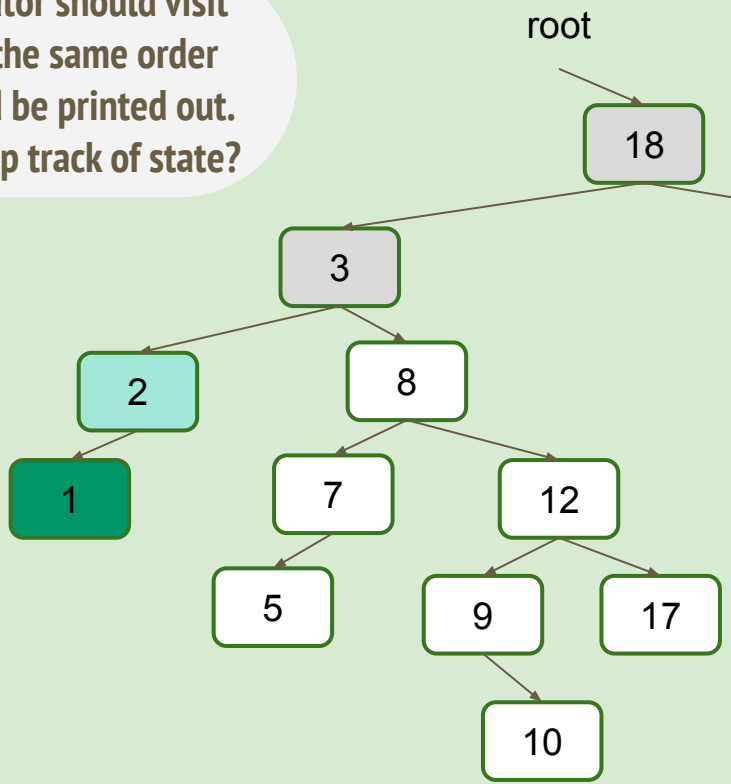
void printInOrder(root):
if nothing: return
printInOrder(left)
print root
printInOrder(right)

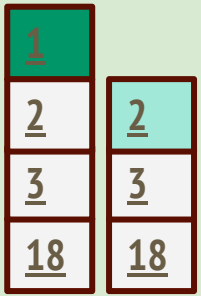
Goal: Iterator should visit nodes in the same order they would be printed out. How to keep track of state?



void printInOrder(root):
if nothing: return
printInOrder(left)
print root
printInOrder(right)

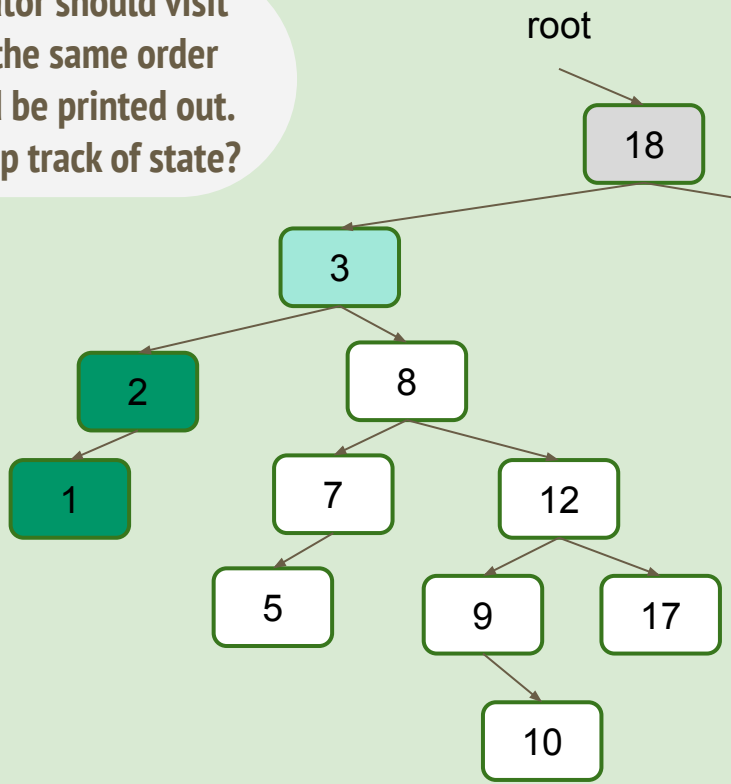
Goal: Iterator should visit nodes in the same order they would be printed out. How to keep track of state?

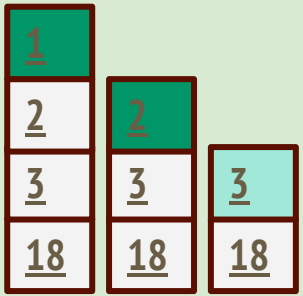




void printInOrder(root):
if nothing: return
printInOrder(left)
print root
printInOrder(right)

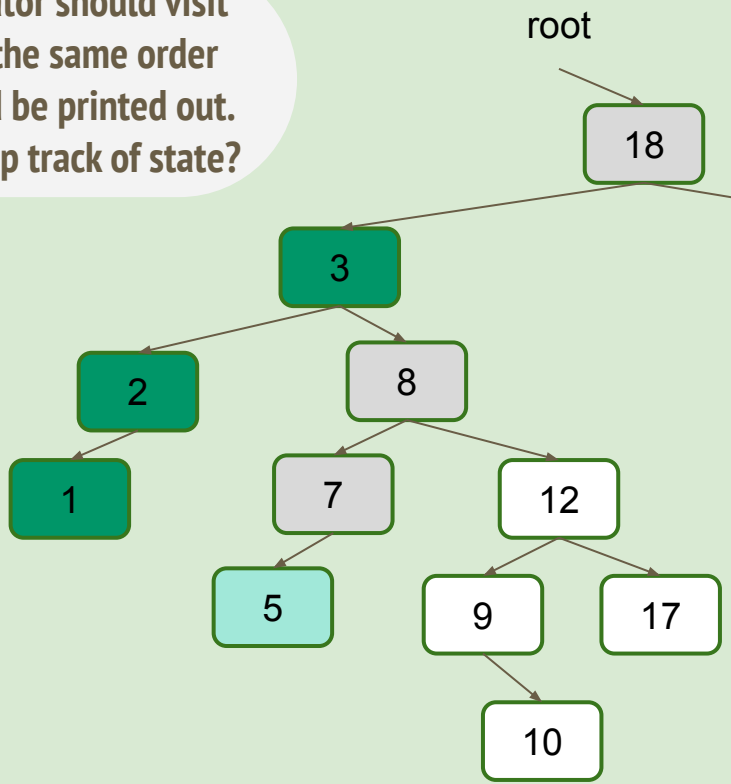
Goal: Iterator should visit nodes in the same order they would be printed out. How to keep track of state?

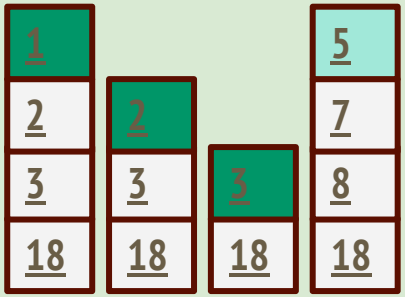




void printInOrder(root):
if nothing: return
printInOrder(left)
print root
printInOrder(right)

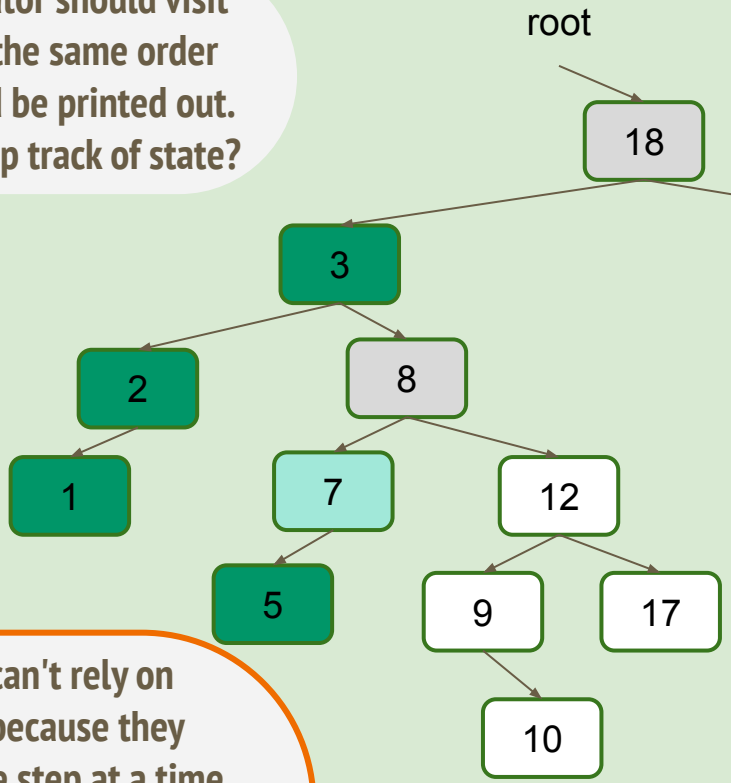
Goal: Iterator should visit nodes in the same order they would be printed out. How to keep track of state?

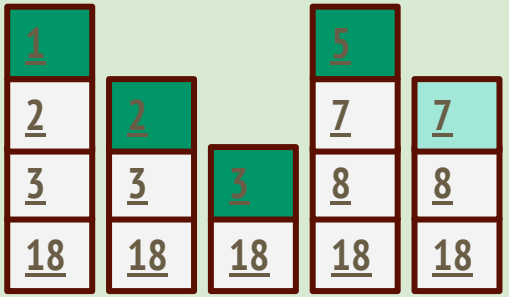




void printInOrder(root):
if nothing: return
printInOrder(left)
print root
printInOrder(right)

Goal: Iterator should visit nodes in the same order they would be printed out. How to keep track of state?





void printInOrder(root):
if nothing: return
printInOrder(left)
print root
printInOrder(right)

Iterators can't rely on recursion because they increment one step at a time, but they can use a new STL data structure: a *stack*! Let's check out the code...