Recursion & Binary Search Trees

Gabe Johnson

Applied Data Structures & Algorithms

University of Colorado-Boulder

This is the second sequence in the data structures and algorithms class! The first sequence was all about pointers and how memory works, and we explored this by implementing a linked list data structure.

In this second sequence, we'll introduce the concept of recursion and explore it by implementing a data structure called a binary search tree.

Episode 1

Recursion

In this first episode we’re introducing recursion.

Recursion, n: *see Recursion.*

Recursion is another tough concept, but fortunately it lends itself to some humor. I've had programming books with this in the glossary.

Recursion is all around. Any time you see a structure where that same structure is repeated (with variation) within itself, that's likely a kind of recursion too.

Seriously, what's recursion?

```
func add_it_up(num) {
   if num > 0 {
     return num + add_it_up(num - 1)
   } else {
     return 0
   }
}
```

Recursive functions call themselves.

A recursive function is one that calls itself. Different invocations of the function have their own local variable scope, so they are effectively separate. As an example, consider this pseudocode that sums up all the numbers that are less than and including some input number:

Seriously, what's recursion?

```
func add_it_up(num) {
   if num > 0 {
     return num + add_it_up(num - 1)
   } else {
     return 0
   }
}
```

Calling add_it_up(6) gives 21: 6 + 5 + 4 + 3 + 2 + 1 + 0

A recursive function is one that calls itself. Different invocations of the function have their own local variable scope, so they are effectively separate. As an example, consider this pseudocode that sums up all the numbers that are less than and including some input number

<next>

Calling `add_it_up` gives 21:

6 + 5 + 4 + 3 + 2 + 1 + 0

Where is recursion used?

Computer Science, Software Engineering, Mathematics

Used when a problem can be broken down into self-similar parts. Divide and conquer!

Used in data structures, programming language parsing, data mining & machine learning, ***ad nauseam…***

Recursion is a ubiquitous concept in computer science, programming, and math. And even though this is another brain-melty concept, once you get the hang of it, it will be second nature, and it is a super duper powerful tool to have at your disposal. Super duper.

Recursion is typically used to address problems that can be either predictably subdivided, or if a solution can be transformed into a simpler case for which the solution is known.

It is used in lots of situations: data structures, parsing programming languages, data mining and machine learning, not to mention all the mathematical uses.

Heuristic for recursion

1. Are we in a done state with the final answer?	**Return.**
2. Are we in a state where we can't continue?	**Return.**
3. Are we in a state where we can break the problem down further?	**Break it down & recurse further.**

The main thing is that we can detect when we should stop recursing so we don't do it forever.

When you're writing a recursive function, the trick is to determine which of the following three states you're in:

1. We're in a state where we can complete a task and return. So: do the work and return.
2. We're in a state where we know we can't complete the task based on the information we have. So: return, either nothing or with some kind of message that we got stuck.
3. We're not yet in a state where we can return, but it is possible the answer is 'downstream', where we can use information we have to recurse deeper. So: break the problem apart and recurse, possibly using the results from the recursive call to do work and return.

add_it_up example

```
add_it_up(6)
= 6 + add_it_up(5)                    // return 6 plus add_it_up(5)
    = 5 + add_it_up(4)                // return 5 plus add_it_up(4)
        = 4 + add_it_up(3)            // return 4 plus add_it_up(3)
            = 3 + add_it_up(2)        // return 3 plus add_it_up(2)
                = 2 + add_it_up(1)    // return 2 plus add_it_up(1)
                    = 1 + add_it_up(0)// return 1 plus add_it_up(0)
                        = 0           // directly return zero
```

In other words, when we're in a recursive function, ask if we're done, or if not, can we subdivide our problem into one where the solution is known or at least might be achievable?

With the `add_it_up` example, the first time the function is called it has an input of 6. We can't directly answer, but we do know the answer is six plus whatever the result of `add_it_up(5)` is. So we recursively call `add_it_up` with 5.

Then we're in `add_it_up` with the input of 5, and sort of like last time we know the answer is five plus whatever the result of `add_it_up(4)` is.

This continues until eventually we invoke `add_it_up` with zero. In this case we can stop recursing and and can return a value. And we happen to return zero.

add_it_up example

```
add_it_up(6)
= 6 + add_it_up(5)                    // return 6 + 15 = 21
    = 5 + add_it_up(4)                // return 5 + 10 = 15
        = 4 + add_it_up(3)            // return 4 + 6 = 10
            = 3 + add_it_up(2)        // return 3 + 3 = 6
                = 2 + add_it_up(1)    // return 2 + 1 = 3
                    = 1 + add_it_up(0) // return 1 + 0 = 1
                        = 0           // 0
```

Now all the other calls to `add_it_up` that were waiting for a response has something to work with, in the reverse order they were invoked. `add_it_up(1)` was waiting to add one to whatever `add_it_up(0)` returned, which ended up zero. So `add_it_up(1)` returns one, and `add_it_up(2)` now has _it's_ answer. This process continues until the original call to `add_it_up(6)` returns = 21.

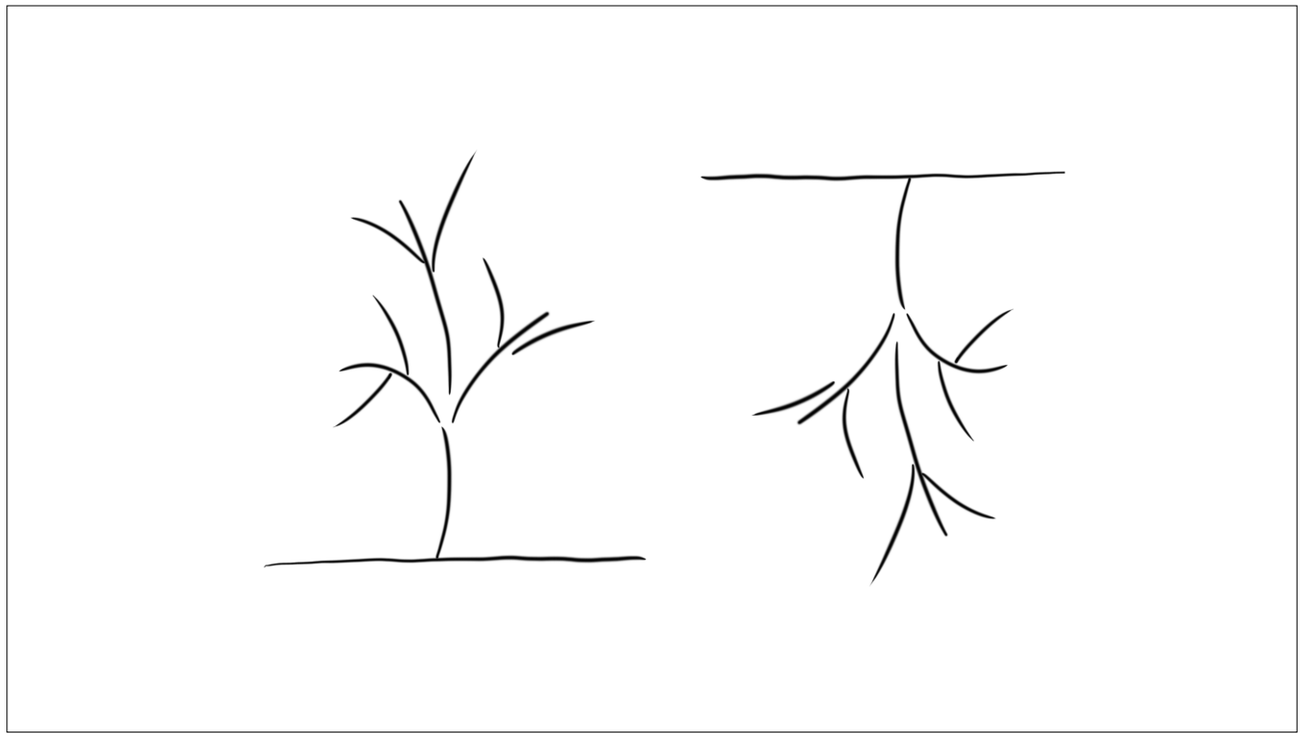
Coming up we'll use recursion in our next data structure, binary search trees.

If you'd like more information on recursion, please see episode 1.

Episode 2

Trees

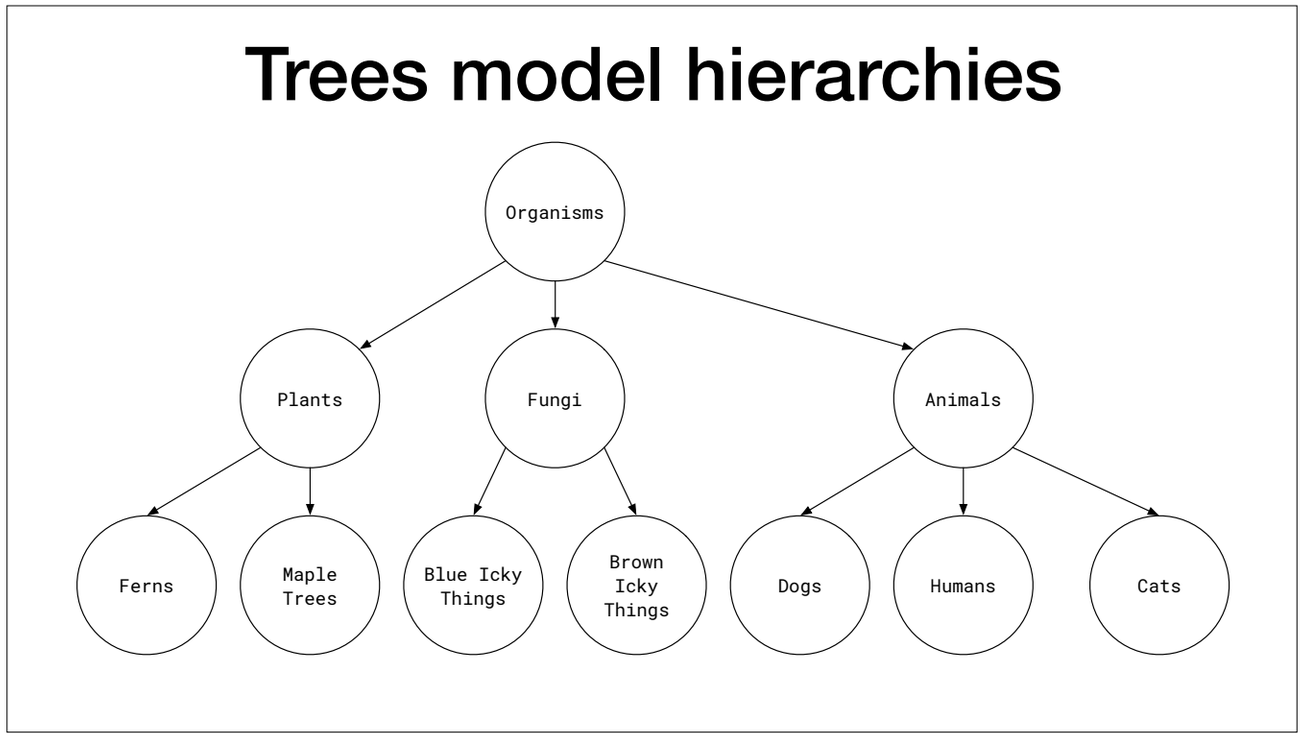
This episode is all about an abstraction called Trees.



Computer scientists love trees. But instead of the kind that grows up from the ground, computer scientists draw their trees starting from the top, and grows downward.

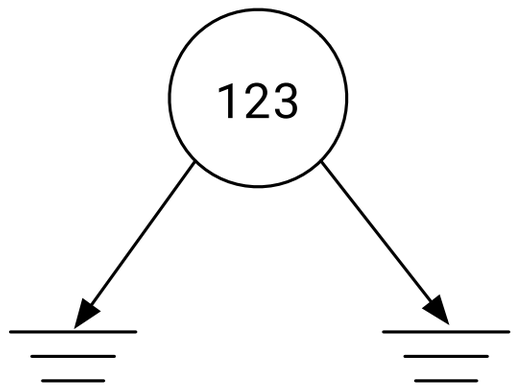
A tree is used to model hierarchies. Trees are built out of nodes and links, sort of like linked lists. Except with trees, nodes can have zero or more child nodes.

Tree structures lend themselves to recursive operations because of their self-similarity, which we're going to talk about in the episode after this one.



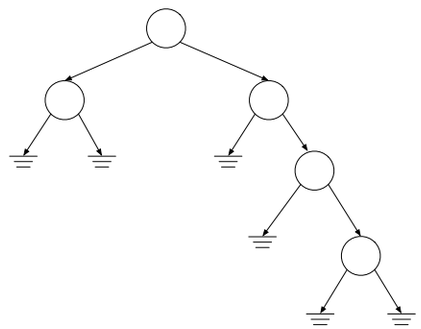
Here's a tree that models some organisms. This particular tree shows a hierarchy of supergroup / subgroup. For example, humans are animals (some more so than others), and animals are organisms. And according to this tree, humans aren't plants, or fungus.

Drawing Trees

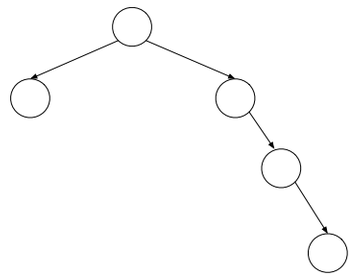


You can draw a tree node like this. The circle has a value in it, just like our linked list node did. But instead of one link to the next node, a tree can have many more links than that. This particular node has two links, both of which are null in this drawing. And since there are two, we call it a binary node, and trees built out of binary nodes are binary trees. If our node had four or eight children, they'd make quad or oct trees.

Drawing Trees



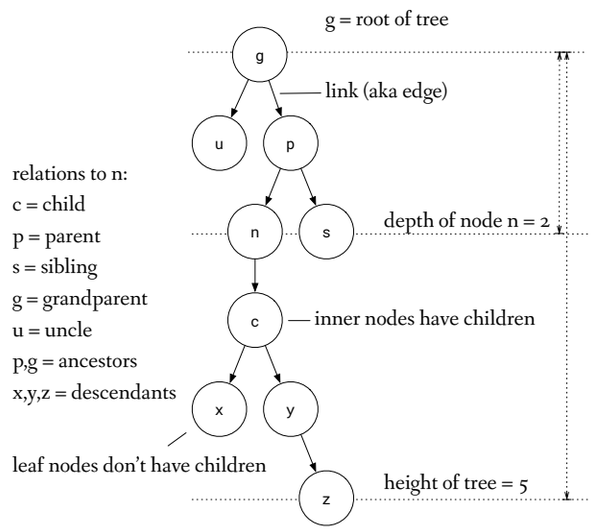
Typically the null links are omitted



This is the same tree without null child links.

By convention, we don't draw the null links, partly because you'd run out of room or they'd clutter things up. Just keep in mind, those null links are still there.

Tree Terminology



Just having two links instead of just one lets us have some interesting structures and relationships between nodes.

An empty tree has no nodes. If the tree does have at least one node, the one at the top is the root node. And a tree can only have one root node. A connection between two nodes can be called a link, or also an edge.

There are family tree names for different relationships. A node with children is called a parent, and all a parent node's children are called siblings. Nodes at the bottom, those with no children, are called leaves. If you follow the parent/child relationships up or down, you can get a node's ancestors, and and node's decedents. A route from one node to another, either up or down, is called a path.

Any node is also the root of a sub-tree, even if the node doesn't have any children.

Unlike a human family tree, a child node can only have one parent.

The overall length of a path from root to a node is the node's depth, and the height of a tree is the longest path from root to any leaf.

Uses for Trees

Maintain order (e.g. numeric, alphabetic, importance, age, etc.)

Find/store data quickly

Model hierarchies (super/sub set, “is a” relationships)

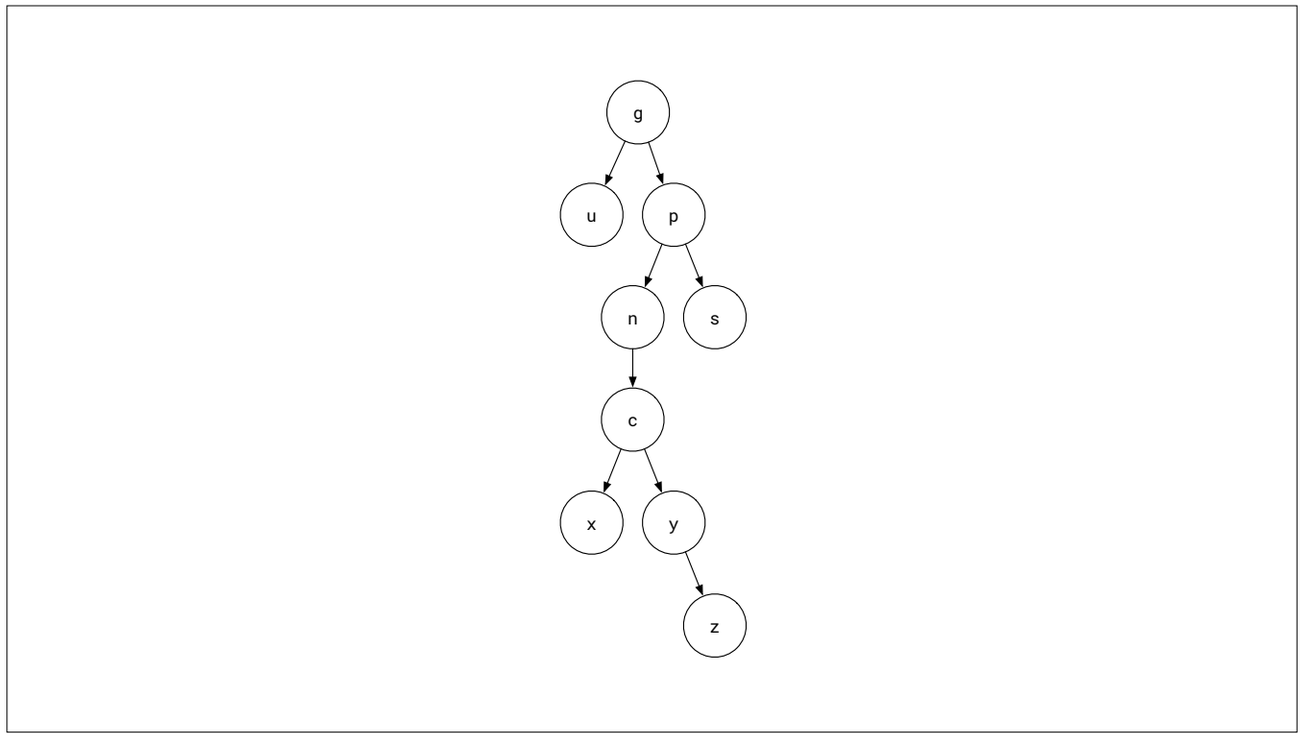
OK, so what can we do with trees? One use is to maintain some sort of order to the data associated with each node. As long as we can depend on that order being intact, we can find items in the tree faster. Throughout this class we're going to see a few kinds of trees---binary search trees to start with, and then a special kind of binary tree called a red/black tree, and later on we'll have a homework assignment on a super fancy kind of tree called a B-tree. All of those trees are designed to make it very fast to store and fetch information.

Beyond sorting, trees can be used to model the sort of parent/child, or superset/subset relationships that we saw with the organism example earlier. Trees can also model activities and the order that tasks can or must take place, which can be used to determine a plan of action to complete the work more quickly.

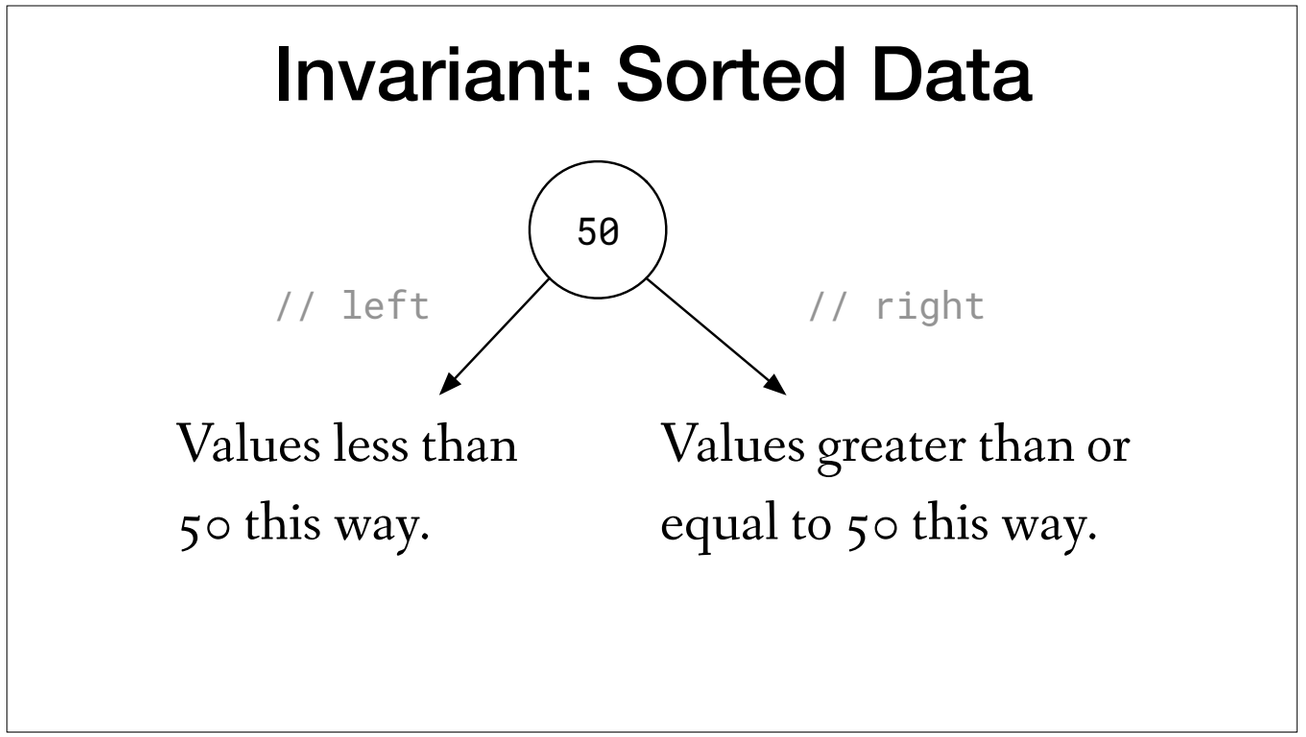
Episode 3

Binary Search Trees: Intro

A binary search tree is a special kind of tree that lets us quickly add and search for data.



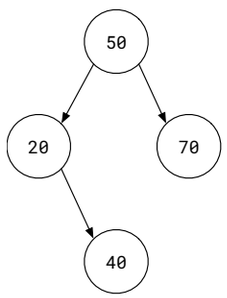
We've seen the general look and feel of binary trees in the last episode with this diagram. Now we're going to talk about what makes binary search trees interesting.



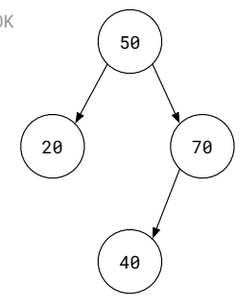
A binary tree can let us add, remove, and find data very quickly. And I'm probably going to leave the 'search' part out of the name from here on. Like the linked list we built earlier, our binary tree will just hold integers.

We use the binary tree node's two child links to keep data sorted. We call them left and right. Let's say we have a tree with a single node storing the number 50. An invariant of the binary tree is that values to the left are stricly less than a node's value, and those to the right are greater than or equal to a node's value. So looking at our 50 node, we can safely say that if we're, say, adding a node with 20, that it should be placed down the left path. If we're querying for a node with the value 70, we only need to look to the right.

Don’t Break the Invariant!




```
// bst ordering invariant OK
```




```
// bst ordering invariant wrong.
// 40 is on the wrong side of 50.
```

It is critical to obey the ordering invariant because the operations all depend on it. The tree at left is OK, but the one on the right has broken the invariant. The 40 should not be found down the right side of 50.

Why sorting matters

Say I have a billion data records, completely out of order.

I'm looking for X in that data set.

How many records must I look at to determine if it is present?

(answer: a billion, worst case scenario)

Say I have a billion data records stored in a nice binary tree.

How many records must I look at to determine if it is present?

(answer: maybe thirty, give or take)

I should say something about why sorted data is useful. Say I have a billion data records, each of which is some awful twenty digit number, but they're not in any particular order. What strategy would you use to determine if a particular number is there?

<next build>

You'd have to look at everything! But if they're in order, then you can use clever techniques--algorithms--for sifting through your data.

<next build>

If our data was in a binary tree, we'd be able to trim out half the remaining data every time we followed a link. Well, on average at least. We'll get into details on why that's on average later one. But anyway, We go from a billion to 500 million, to 250 million, to 125 million, and so on.

<next build>

On average you'd only have to look at 30 nodes to determine if your target number was there. 30 is a lot better than a billion!

Data Structure

```
struct bt_node {
  int data;
  bt_node* left;
  bt_node* right;
};
```

We're working with a binary tree node structure, which has this definition.

The name is `bt_node`, rather than just `node` to distinguish it from a linked list node. But we could have called it just plain old `node`, or whatever. It has an integer data field, and two links, left and right, both of which are pointers to other bt_nodes.

Binary Search Tree Operations

init_node	**create a new node**
insert and insert_data	**add data to the tree**
remove	**remove data from the tree**
contains	**does tree contain specific value?**
get_node	**return a node with a specific value, or NULL**
size	**report the number of nodes in the tree**
to_array	**report the values of the tree in sorted order**

Like most any other data structure, there's a set of operations that go along with it. They'll seem familiar to you.

The `init_node` function creates our new `bt_node` in memory and gives a pointer to it. `insert` and `remove` modify the tree, so we have a double pointer just like we did for the modification functions for the linked list data structure. And then we have four query functions, `contains`, `get_node` (which gives you a pointer to a node with a specific value), `size`, and `to_array`. That last one is sort of like the linked list `report` function, except instead of a string, it fills up an integer array that is provided to the function from the calling context.

In the next episodes we'll cover tree traversal, which is vaguely equivalent to how we used a cursor to march through the linked list structure. But tree traversal is way more interesting. Then we'll cover the specific operations that you'll implement in your homework.

Episode 4

Tree Traversal

Now we’re going to dive in to how we examine trees. With linked lists we used a cursor to scroll through it. But with trees, we have decisions to make about which direction to go, and which order we do things in. Examining a tree like this is formally called traversal, but usually we just say ‘walk’.

Three Ways to Walk

Pre-order: visit the parent first, then all the children from left to right.

In-order: visit some children (starting on the left), then visit the parent, then visit the remaining children (also left to right).

Post-order: visit the children left to right, then visit the parent.

When walking any tree, you _visit_ a node and its children. The three different ways of doing it depend on the order that you visit the nodes. And by _visit_ we mean 'do something with', which is a super technical term. Visiting means you're taking some action on the node your cursor is on.

Pre-order: visit the parent first, then all the children from left to right.

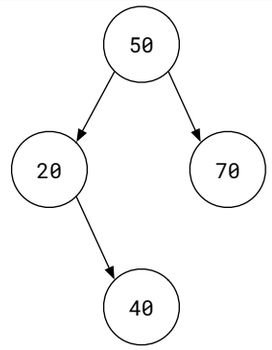
In-order: visit some children (starting on the left), then visit the parent, then visit the remaining children (also left to right).

Post-order: visit the children left to right, then visit the parent.

Apologies for making the in-order definition weird. For a general tree with an arbitrary number of nodes, there's not really a natural way to define in-order. But for binary trees, it makes a lot of sense and you'll actually use it often.

For binary trees, in-order means you visit left child, parent, then right child. The examples I’m about to give are all about binary trees.

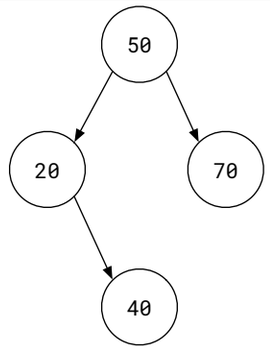
Recursive Traversal




```
void visit(bt_node* t, string how) {
  cout << "visiting node with traversal strategy: " << how <<
          " value: " << t->data << endl;
}
```

In all of these cases, when you visit a child node, you also traverse the entire subtree it represents, using the same kind of traversal. Let's run through this tree we've seen before, with a basic `visit` function that just prints out the strategy and a node's value.

Pre-order Traversal




```
void preorder(bt_node* t) {
  if (t == NULL) {
    return;
  }
  visit(t, "preorder "); // do some action, such as print out t->data
  preorder(t->left);     // recursively traverse left child
  preorder(t->right);    // recursively traverse right child
}
```

```
visiting node with traversal strategy: preorder  value: 50
visiting node with traversal strategy: preorder  value: 20
visiting node with traversal strategy: preorder  value: 40
visiting node with traversal strategy: preorder  value: 70
```

So you'll likely have some function like `preorder(bt_node* t)` that does something like this.

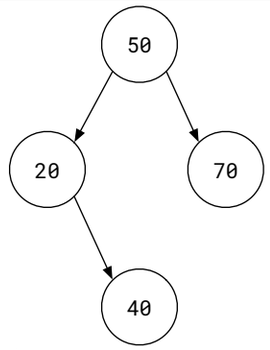
It will print out 50, 20, 40, 70.

Let's walk through that. So, first thing we do is visit the root, which prints 50. Then we visit the left tree with a recursive call to preorder, using the current node's left pointer.

Then we're pointing to that 20 node. Preorder, so print its value, and recursively call preorder on its left tree. It's null, so we just bail out. The right tree isn't null, so it prints out 40, recurses left and right, but they're both null so nothing is printed out. We're then done with the 20 node, so re return to the 50-node, which has been waiting for the recursive call to the left subtree to return.

Well, we're back, so then it recurses down the right subtree. There we encounter the 70 node, print its value first, then recurse left (which is null) and then right (also null). The recursive calls all wind back, and we finish the entire tree.

In-order Traversal



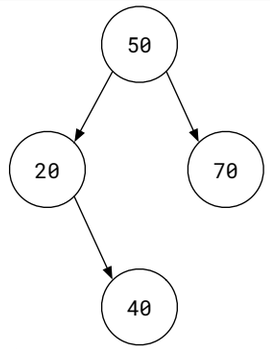

```
void inorder(bt_node* t) {
  if (t == NULL) {
    return;
  }
  inorder(t->left);      // recursively traverse left child
  visit(t, "inorder  "); // do some action, such as print out t->data
  inorder(t->right);     // recursively traverse right child
}
```

```
visiting node with traversal strategy: inorder   value: 20
visiting node with traversal strategy: inorder   value: 40
visiting node with traversal strategy: inorder   value: 50
visiting node with traversal strategy: inorder   value: 70
```

Similarly, here is in-order traversal. This prints out the order 20, 40, 50, 70.

Notice something interesting about that output? It is in ascending sort order! Neat. And that's not an accident, and I'm sure a slightly modified version of this inorder walk will be super useful when you implement the `to_array` function for the homework. Nudge nudge wink wink.

Post-order Traversal




```
void postorder(bt_node* t) {
  if (t == NULL) {
    return;
  }
  postorder(t->left);    // recursively traverse left child
  postorder(t->right);   // recursively traverse right child
  visit(t, "postorder"); // do some action, such as print out t->data
}
```

```
visiting node with traversal strategy: postorder value: 40
visiting node with traversal strategy: postorder value: 20
visiting node with traversal strategy: postorder value: 70
visiting node with traversal strategy: postorder value: 50
```

The last traversal method is post-order, and you can probably guess what it looks like already. It prints out 40, 20, 70, 50.

Traversal Methods are Similar

```c
void preorder(bt_node* t) {
  if (t == NULL) {
    return;
  }
  visit(t, "preorder ");
  preorder(t->left);
  preorder(t->right);
}
```

```c
void inorder(bt_node* t) {
  if (t == NULL) {
    return;
  }
  inorder(t->left);
  visit(t, "inorder  ");
  inorder(t->right);
}
```

```c
void postorder(bt_node* t) {
  if (t == NULL) {
    return;
  }
  postorder(t->left);
  postorder(t->right);
  visit(t, "postorder");
}
```

Let's look at all three functions right next to each other. Super similar. Really we're only adjusting the order that we visit nodes.

All three of these traversal methods have their uses, and we'll encounter all of them as the course progresses. One thing I didn't draw attention to is that first part in all three, where we're checking t for nullness. That's the recursion release valve, where we determine if we can no longer continue. It is really important that your recursive functions have something like that, otherwise they might run forever.

Episode 5

Binary Search Trees: Operations

This episode is all about the functions you're going to implement for the homework.

Remember that we're working with binary trees, meaning nodes have up to two children, named left and right. And specifically we're dealing with binary _search_ trees, which adds the invariant about sort order--go to the left for smaller values, go to the right for all others. I use a silly memnonic to remember this: left is less! Both words start with L and are four letters.

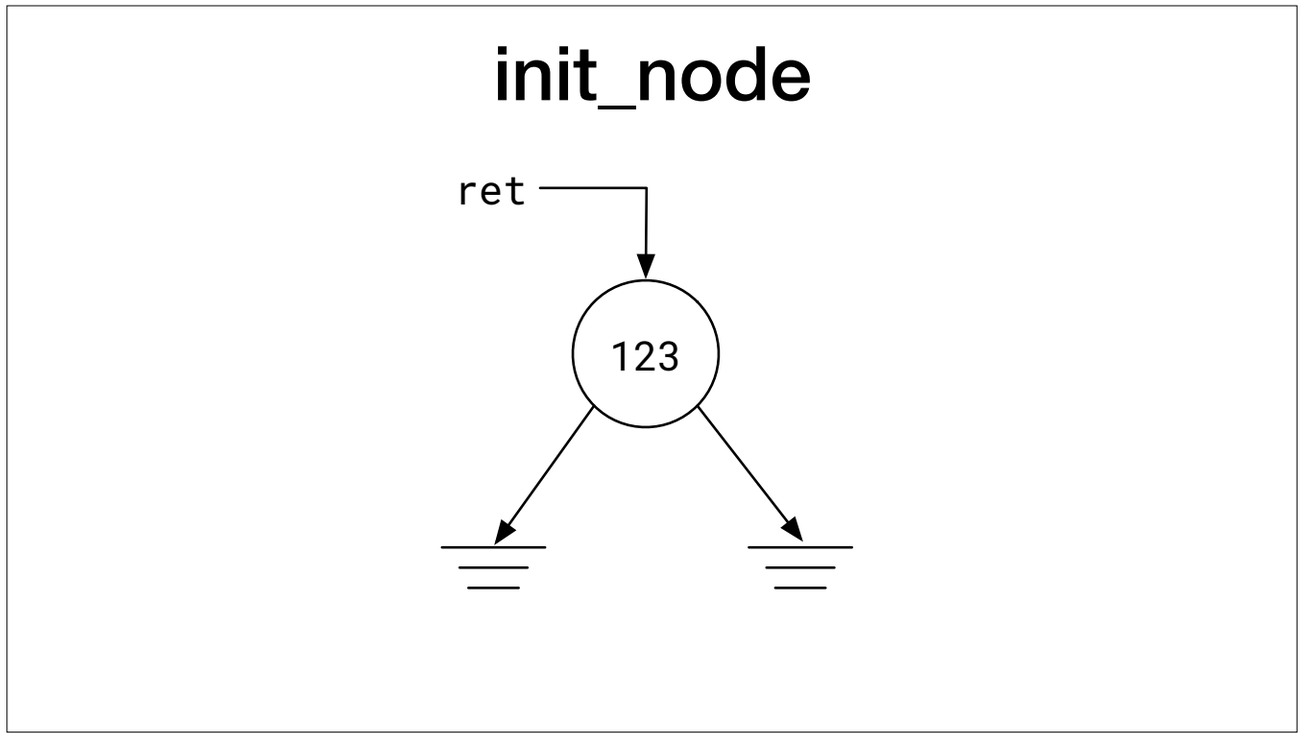
BST Operations

init_node	**create a new node**
insert and insert_data	**add data to the tree**
remove	**remove data from the tree**
contains	**does tree contain specific value?**
get_node	**return a node with a specific value, or NULL**
size	**report the number of nodes in the tree**
to_array	**report the values of the tree in sorted order**

The operation list is like this. There are other operations you can have with binary search trees, obviously. And some of those you'll likely need to implement as helper functions.

There's the one initialize function, which is super similar to the linked list one. Then we have two functions that modify the tree, insert and remove. Those are tricky, especially remove, so I'll cover that in the next episode all by itself.

Then there's the four query functions that should not modify the tree in any way.



The initialize function allocates new heap memory for a `bt_node`, sets the data value to whatever number we're given, and sets the left and right links to NULL. It then returns a pointer to that new node.

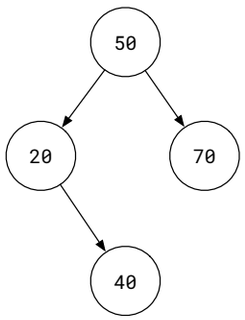
insert

```
void insert(bt_node** top, bt_node* new_node);
```

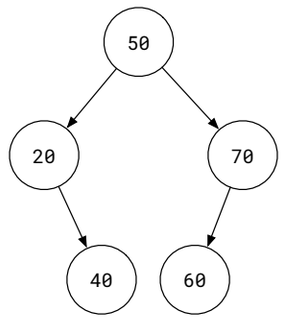

Double pointer, just like in linked list modification operations.

Notice there's a double pointer, just like in the linked list modification operations. This means if there isn't a node in memory yet, we can put one there and the calling context will retain the reference.

insert



Insert 60 into this tree.



Resulting tree after inserting 60.

Remember the invariant: left for less!

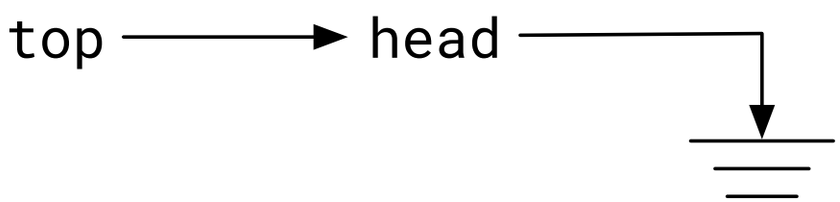
Inserting a node requires us to search through the tree for a spot to put it. Say we're asked to insert a node with value 60 into this tree.

Start with the top of the tree, with node 50. Think about the binary search tree invariant. If we're inserting 60, where should it go? Left or right? Well, 60 is larger than 50, so we search right by recursively calling insert with the right child.

Now we're inserting at node 70. Since that's not NULL, we have to look at its children next. Where should node 60 go in relation to node 70? To its left. Recurse that way.

Now we're inserting at 70's left child, but it is NULL! So we've found the spot to put our node 60. Just set 70's left child pointer at our 60 node, and we're done.

insert into empty tree



Think about how you'll handle the case where you're inserting into an empty tree. And actually, we just did! That last step, where we insert into 70's left child, which was null, is equivalent to inserting into a completely empty tree.

contains

```
bool contains(bt_node* t, int data);
```

Remember the invariant: left for less!

The `contains` function simply seeks out a target value and returns true if it finds it, or false if it can't. This follows the same logic, mostly, as insertion does for finding the insertion site.

get_node

```
bt_node* get_node(bt_node* t, int data);
```

Hint: implement this first, then use it to make ‘contains’ trivial.

The `get_node` function is similar to contains in how it seeks out the target node, but instead of returning a boolean, it returns either a pointer to the node, or NULL if it can't find the node. You might consider implementing get_node first, and then letting the `contains` function use this one to do the work.

size

One idea:

```
int traverse(bt_node* t);
```

Another idea:

```
traverse(bt_node* t, int* count);
```

There are still other ways to do it.

The `size` function is a great place to try out a traversal routine. Every time it visits a node, increment a counter. How you implement that is up to you, but there's at least two ways of doing it. You could have your recursive traversal function return an integer, the number of nodes it has in its subtree, or you could pass an int pointer in to the recursive traversal function, which just increments the counter as a node is visited.

It's up to you. By the way, the size function itself could be the traversal function.

to_array

```
void to_array(bt_node* t, int arr[]);
```

Hint: use a helper function that uses recursive in-order traversal.

Also: carefully read the header file documentation comment for this function.

The header file has an extended comment about how this should work. But an important thing to note is that the array doesn't return anything---it just populates the array that is passed in. That array will already be created and sized appropriately.

Recall the video where we covered recursive in-order traversal. That will totally help here. You'll need to write a helper function.

Episode 6

Binary Search Tree: Remove

Removing a value from a tree is a tricky process, so it gets its own episode.

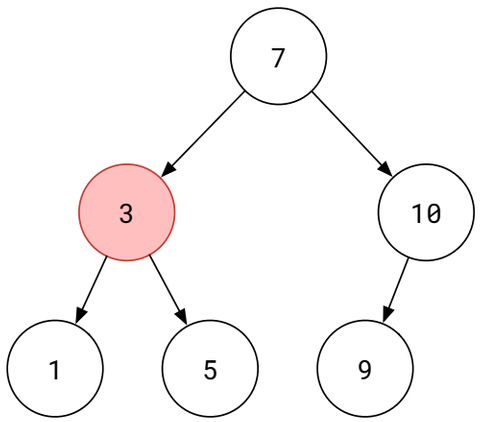
remove

```
void remove(bt_node** top, int data);
```


Double pointer, just like in linked list modification operations.

... so, another double pointer because the calling context needs to see if we've made any changes to the node pointed to by `*top*. This should be getting pretty familiar by now.

remove

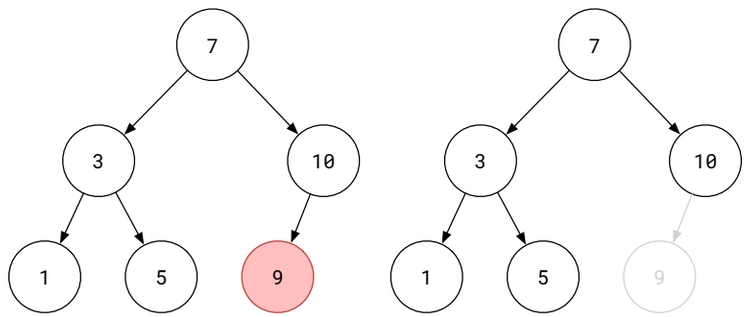


What happens if we want to remove 3? Aaaaargh

First you need to identify if the value is even in the tree, and if it is, where is it? You might consider using some of the query functions to help here.

Once you've found the node to remove, it isn't just a matter of deleting it, because it might have children. And remember the binary search tree invariant: by the time you're done with your remove operation, the nodes must still be in sort order, left is less and all that. And, we can't orphan anything. Only remove one node. So, careful!

remove: no children

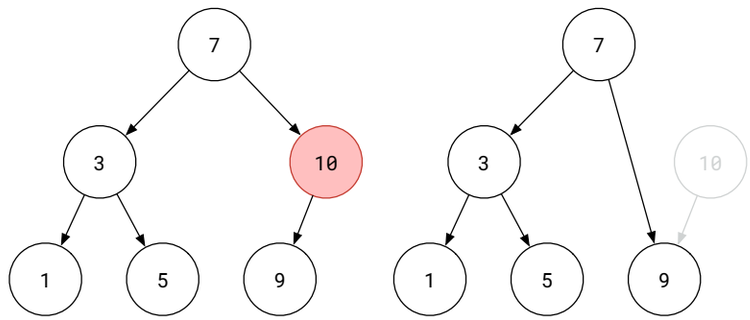


Original Tree

Remove 9 from original tree. 9 has no children, so just remove reference from its parent.

Say we're removing a node that doesn't have any children. In this case, we can just remove the reference and free the memory to the removed node with `delete`.

remove: one child

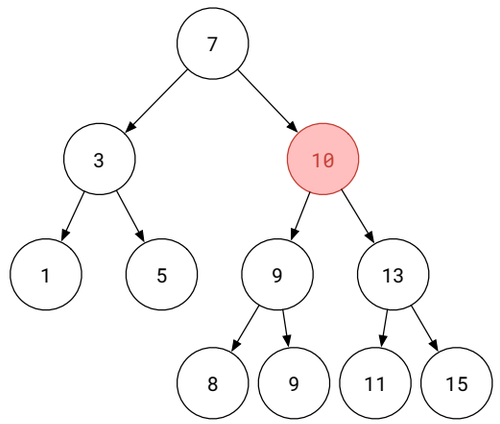


Original Tree

Remove 10 from original tree. 10 has one child, so redirect 10's parent to point to that child.

If the doomed node has one child, it is still pretty easy, we just need to update the reference from the doomed node's parent to the one child of the doomed node. And then remember to reclaim that memory.

remove: two children

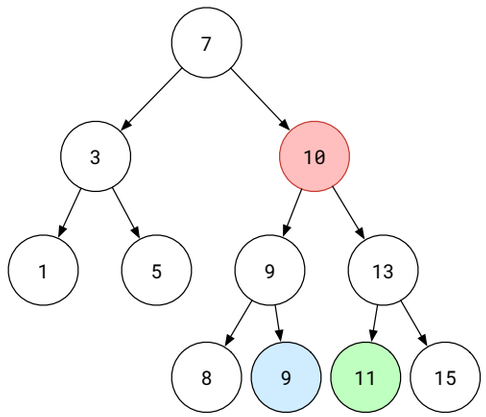


Larger tree, 10
will be removed.

Now things get complicated when we need to remove a node that has two children, because we have to shuffle things around to maintain the invariant.

If we need to remove 10 from this tree, what should it look like afterwards? Stare at that for a second. What should 7's right pointer point to when we're done? How can we maintain sort order, left is less?

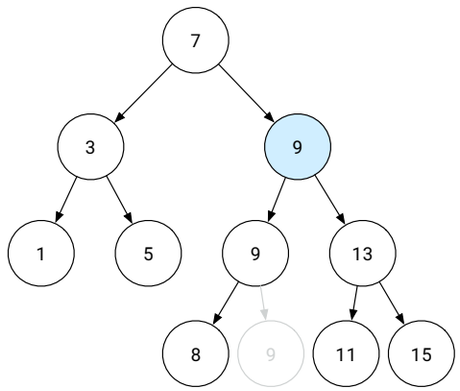
predecessor and successor



Node 10's sort order predecessor (9)
and successor (11), aka *buddies*.

One way to do it is to find a node that is adjacent from a sort-order perspective. The node that sorts just before our doomed node is called the predecessor; the one after is called its successor. I'll explain how to find those in just a second.

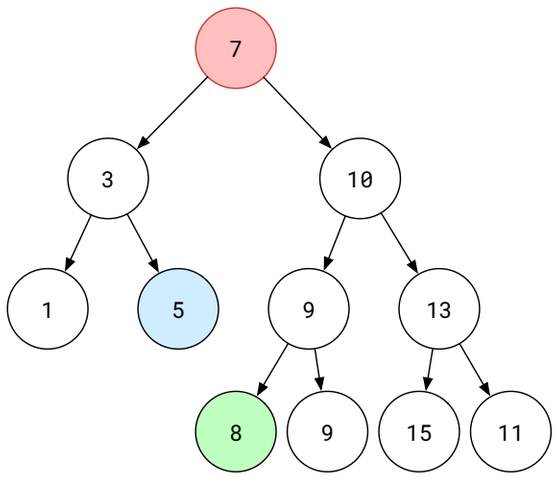
remove: two children



Replace the *value* in the doomed
node with one of the buddy values.
Then remove the buddy.

Pick one of those two, we'll just call it the buddy. Now, take the buddy's value, stick it into our doomed node's value slot, and then remove the buddy. So maybe I shouldn't say 'doomed', since that node in memory is going to stick around, and it is the buddy that gets deleted. Poor little node...

finding predecessor & successors



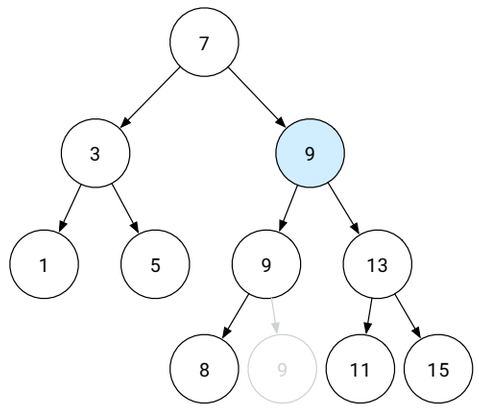
Predecessor to 7: left turn followed by as many right turns as you can make (5).

Successor to 7: right turn followed by as many left turns as you can make (8).

This only works if the BST invariant is intact!

Anyway, here's the trick to finding predecessor and successor. From any node, the predecessor will be a left turn followed by as many right turns as you can make. Successor is the inverse: make a right turn then as many left turns as you can make. And, it is possible that the predecessor or successor is not a leaf node, but it is guaranteed to not have two children.

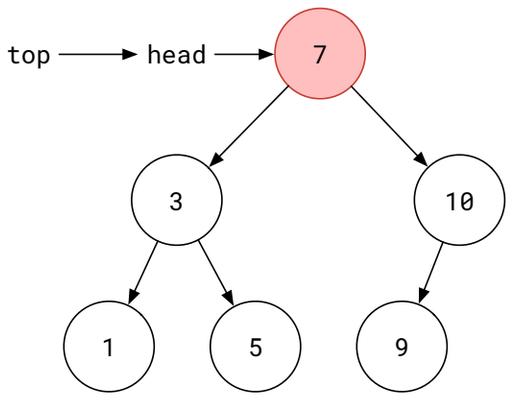
remove: cleaning up



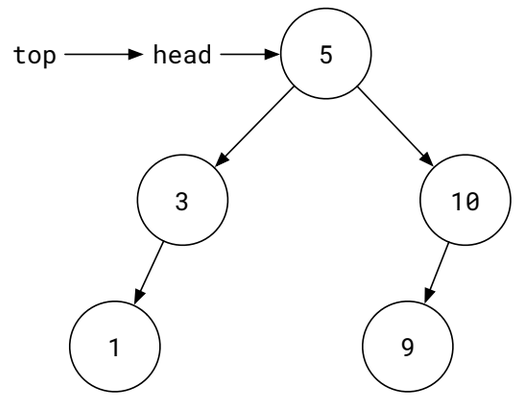
After swapping buddy's value into the node that formerly had `10` in it. How to remove the light gray one? We have this `remove()` function... Call it with the address of the incoming pointer.

When your node to remove has two children, you actually swap in its buddy's value, and then remove the buddy. You have to do this before you return from the remove function, because if you don't you could be violating the binary search tree invariant. Don't worry, it isn't as scary as it sounds. You've already done the process when you solved remove for the case of zero or one children, and the predecessor or successor that you're going to remove is guaranteed to fit that description.

remove: root node



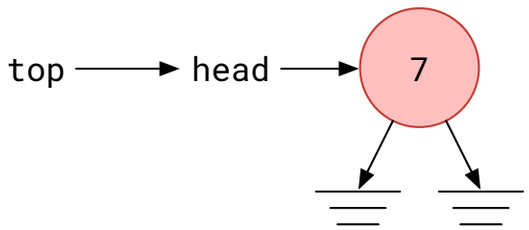
Remove root?



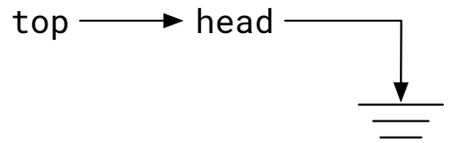
Invariants satisfied :-)

Special care has to be taken when you're removing the root node, just because you'll need to ensure that the pointer to the tree is kept up to date, sort of like you did with linked lists when the head node changed.

remove: root node



Remove root?



Invariants satisfied :-)

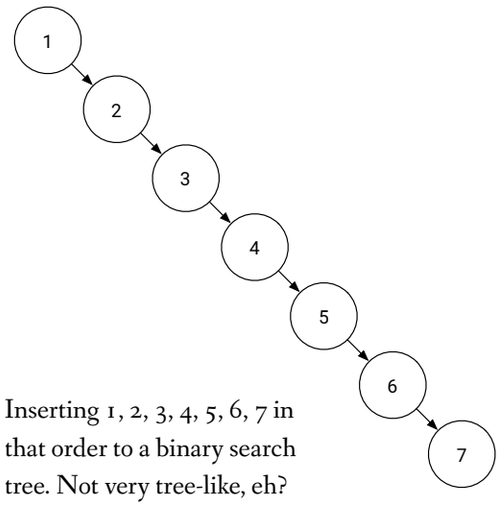
Be sure to catch this situation too.

Episode 7

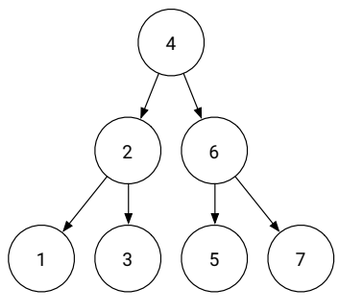
Balanced Trees

This episode and the next one on red-black trees aren’t required for the binary search tree assignment, but they’ll really help out for the B-tree assignment in a couple weeks.

Pathological Trees



Inserting 1, 2, 3, 4, 5, 6, 7 in that order to a binary search tree. Not very tree-like, eh?



The same data with a different structure. Balanced.

If I insert 1 through 7 in that order to a fresh binary search tree, I end up with a tree like this:

[tall tree]

And that's not very tree like. But more importantly, it means if I perform an operation on this tree, I might have to scroll through seven nodes before I can do whatever it is I'm doing. Now, if I inserted 1 through a billion, that's a lot of scrolling.

If instead we insert the first seven numbers in an optimal order, we might have this tree instead:

[balanced tree]

And this tree has a height of 3, so that's less scrolling. In our hypothetical billion-node tree, if we inserted everything in just the right way, that tree would only have a height of about 30. In that case, we can search through the whole tree super fast, relative to how much data it contains.

This second tree is called balanced, because it minimizes the tree's height by making more effective use of each tier, which means we'll minimize the worst-case distance from the root node to the most distant leaf.

Balanced Trees

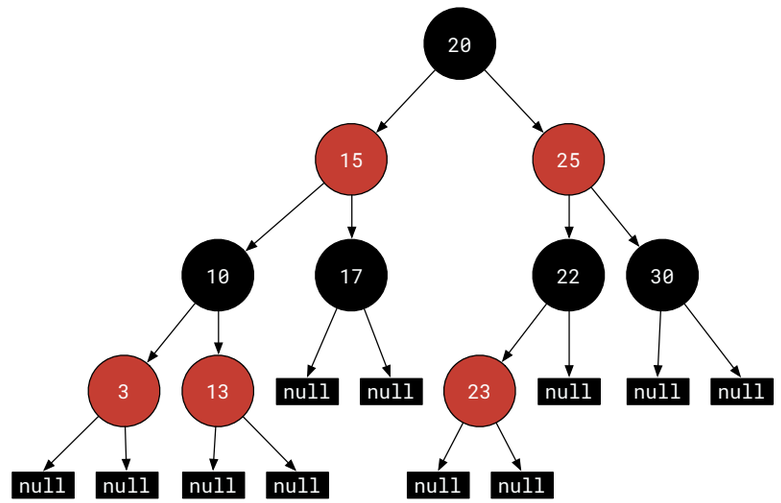
Special tree data structures can be self-balancing!

-AVL Trees
-Red-Black Trees
-Splay Trees
-B-Trees

So since a balanced tree is a good tree, people have devised cunning strategies to build trees that self-balance upon every modification. There are many kinds of self-balancing trees, like AVL trees, red-black trees, and splay trees. Today we're going to look at red-black trees.

You're not going to implement this one, but don't let that be your sign to take a nap. Later on, you'll be implementing B-trees, which are super rad self-balancing trees, and are a lot trickier than this. So try to learn how a simple self-balancing tree works, and you'll have an easier time with the B-tree assignment.

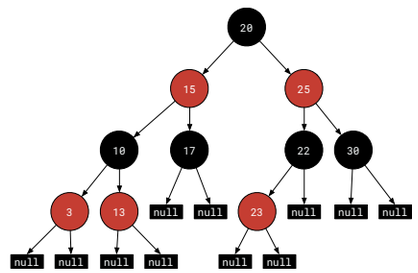
Red-Black Trees



Red-black trees are a specialization on binary search trees. So a red-black tree has the same left=less invariant that a binary search tree does, and has additional constraints that are used in the self-balancing process.

Red-Black Tree Invariants

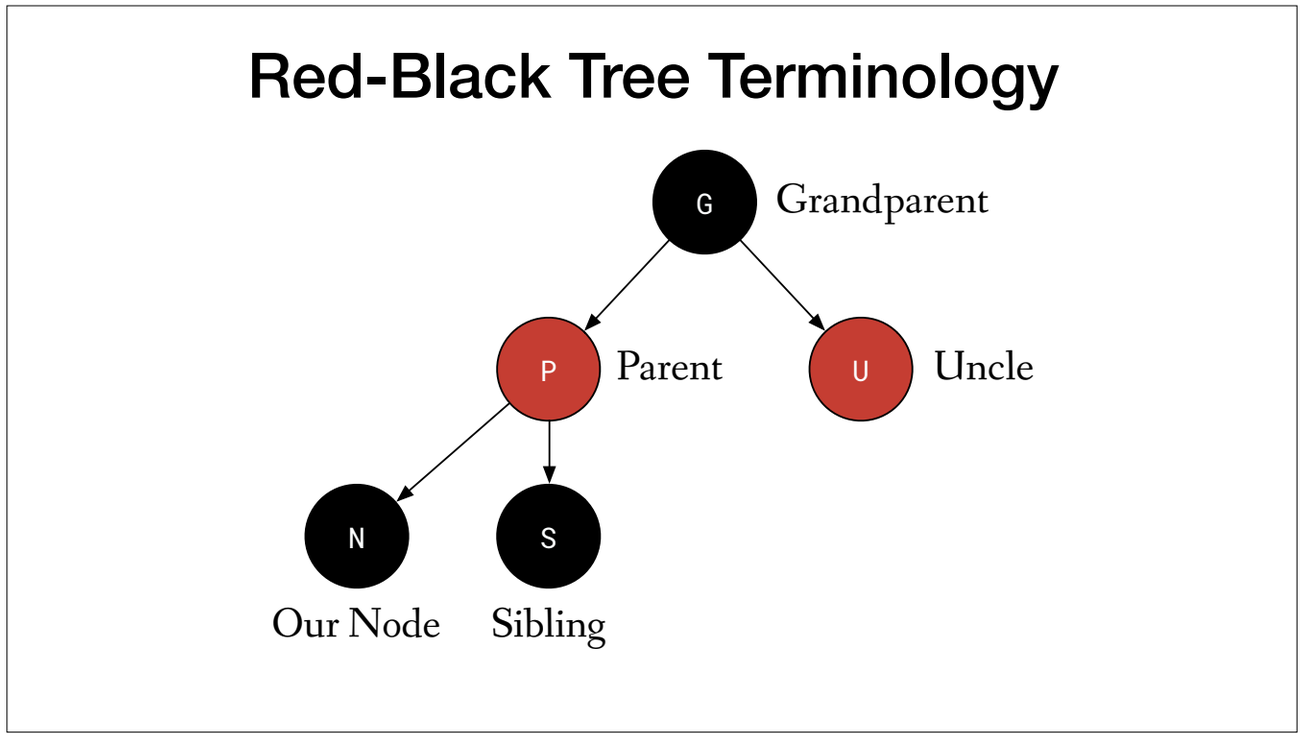
1. Nodes are colored red or black.
2. Root is black.
3. Leaves are null and are considered black.
4. Red nodes are only allowed to have black children.
5. Paths from a node to any descendant leaves have the same number of black nodes.



The additional invariants are:

1. Nodes are colored red or black. They could have picked any two colors, maybe puce and mauve, but that would look bad.
2. Root is black.
3. Leaves are null and are considered black.
4. Red nodes are only allowed to have black children.
5. Paths from a node to any descendant leaves have the same number of black nodes.

That last rule is the heart of the matter that lets us use coloring to self-balance the tree.



Here's the anatomy of a red black tree node, the one marked N, "our node" at the bottom left. The family relationships are actually technical terms used in the balancing process. There's the parent, sibling, uncle, and grandparent nodes.

Family Relations

```
RED = 0
BLACK = 1

class red_black_node:
    color = BLACK
    parent = None
    left = None
    right = None
    value = 0

# returns grandparent of node, or None if not available
def grampa(n):
    if n is not None and n.parent is not None:
        return n.parent.parent
    else:
        return None
```

Here's pseudo-code for a red-black node and function to get a node's grandparent. Python, actually.

So notice that our red-black nodes have left and right pointers, and also a parent pointer and a color. If we maintain the parent pointer properly, we can always get to our grandparent if we have one. Obviously if we're the root node or a child of the root, we won't have a grandparent.

You can imagine similar routines to get at the sibling and uncle nodes as well, being careful to avoid null references in case those relations don't exist.

Operations

Red-Black Trees have the same operations as standard binary search trees.

We will cover the ‘insert’ function and all the stuff that goes with it.

No homework on this, but helpful to understand before attempting the B-Tree assignment.

In the next episode we'll cover inserting into red-black trees. This should give you a feel for what the cleanup process is like. You should study this, and read up on the red-black remove operation, before starting the B-tree assignment.

Episode 8

Red-Black Tree Insert

This episode is about how to insert into a red-black tree. It starts out like a standard binary tree insertion, but then there's a cleanup process. It involves finding our family relations, looking at their colors, figuring out which case to start in. Then it is a matter of performing 'rotations' and re-coloring nodes so that the tree maintains its invariants.

Insert: Five Cases

Case 1 is where the new node is the first one, the tree root. Make it black. If the tree already has a root, use case 2 instead.

Case 2 is where P (the parent) is black, and we're done. If P is red, call case 3.

Cases 3, 4 and 5 are trickier and deserve some diagrams to explain.

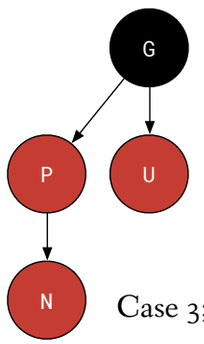
There are five cases you need to consider when inserting a new node. The new node is painted red. Insert it just as you would with a binary search tree, then repair whatever damage you did to the red-black invariant by invoking the cleanup routine chain.

Case 1 is where the new node is the first one, the tree root. Make it black. If the tree already has a root, use case 2 instead.

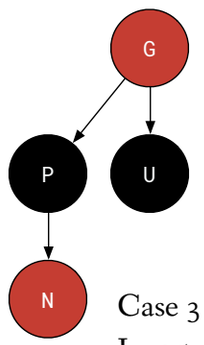
Case 2 is where P (the parent) is black, and we're done. If P is red, call case 3.

Case 3, 4 and 5 are trickier and deserve some diagrams to explain.

Insert: Case 3



Case 3; N is the node we just added

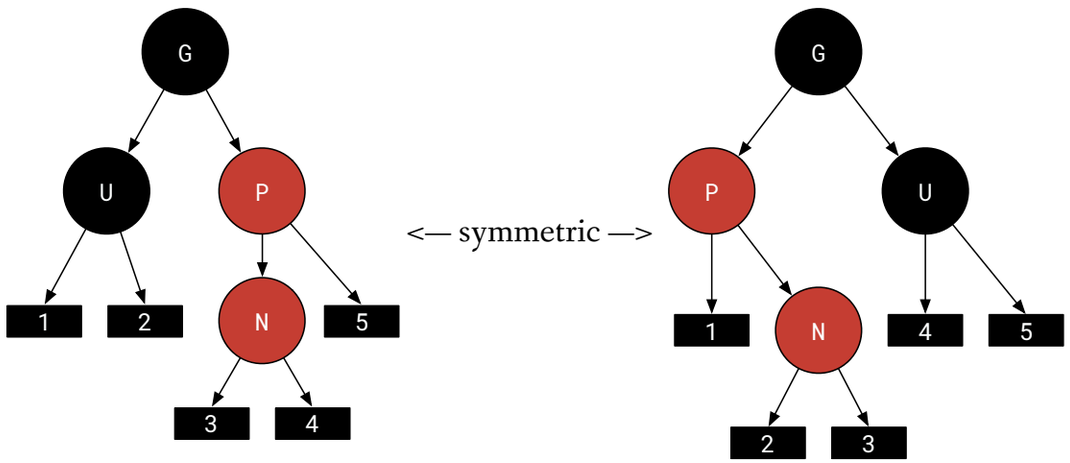


Case 3 cleanup;
Invert colors for G, P, and U.

Case 3 is when P and U are both red. If that doesn't match your situation, call case 4.

In this case, change the colors as shown here. G, P, and U all just invert their colors. G ends up red, P and U end up black.

Symmetry

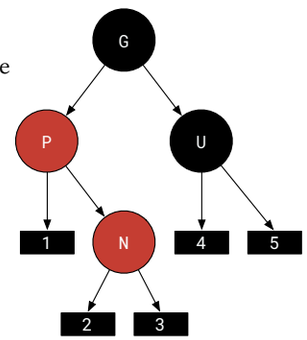


Cases 4 and 5; N is the node we just added.

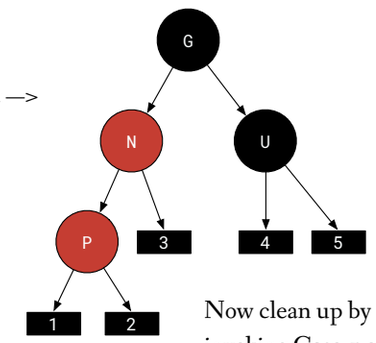
For cases 4 and 5, each has a symmetric 'left' and 'right' case. I'm only showing one of them, where we rotate left. But for each, you should have two cases that are symmetric. Just swap 'left' with 'right' and you'll be good.

Insert Case 4

Case 4:
N is recently added node
P is red
U is black
N is the right child of P
P is the left child of G



Rotate Left —>



Now clean up by
invoking Case 5 on P.

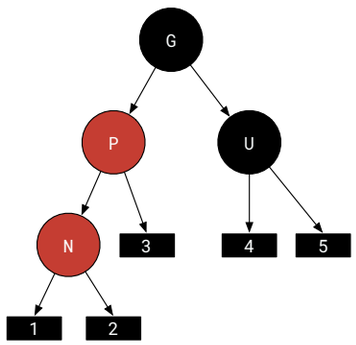
Case 4 is when P is red but U is black, and N is the right child of P, and P is the left child of G. Much easier to look at the drawing.

When we identify case 4, and remember there's a mirror image of this, we perform a left rotation on P. Like I said, it is easier to understand if you look at the drawing. All we're doing here is moving N up to where P is, and moving P down to be a child of N, and adjusting some links to keep the sort order invariant happy. There's a symmetric right rotation as well.

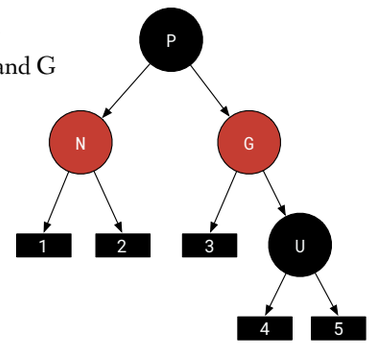
After the rotation, you can see that we have two red nodes in a row, which breaks one of our invariants, so we have to then clean up _that_ mess by invoking case 5 on P.

Insert Case 5

Case 5:
P is red
U is black
N is left child of P
P is left child of G



Rotate G Right—>
Invert colors on P and G



Case 5 is when P is red but U is black, N is the left child of P, and P is the left child of G. Check out the diagram.

We'll rotate G to the right and invert colors for P and G, so P is now black and G is now red. We should now have a cleaned up tree that no longer breaks any of the invariants.

That’s just insert…

Self-balancing cleanup gives nicely balanced trees…

Good for finding the spot you’re looking for quickly, even with billions of nodes.

Head trip! This has secretly been about Computational Complexity, which is the topic for the next sequence.

Next up: Live coding sessions.

Red-Black trees have this cleanup process in both the insert and remove operations. The benefit to this is that we always have a nicely balanced tree so even in the worst case scenario, even with billions of nodes, we don't have to dive deeply to find the spot we're looking for.

And throughout this whole discussion I've been subversively introducing the main topic for the next sequence, which is computational complexity. We'll revisit this after the coding sessions.