Chapter 3

Binary Search Trees

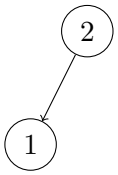
3.1 Introduction

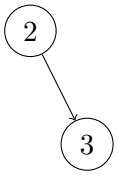
The **binary search tree** is probably the first new concept that you will learn about. Fortunately, computer scientists are not very creative so the name somewhat hints at this data structure's use. Unfortunately, I cannot come up with a real world example of this.

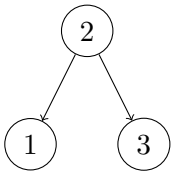
3.2 Terminology

- **node**
  A **binary search tree** is made up 0 or more **nodes**. You can imagine these as small containers with data along with two pointers (because this is a **binary** tree), typically denoted as left and right. In the following diagrams, a **node** will appear as a circle:

- **children**
  A node can have up to two **children** (again, **binary**). These may be referred to as the left and right **children** depending on what side they are on.







- **parent**
  The opposite of a child. If a node A is a child to node B, then node B is node A's **parent**.

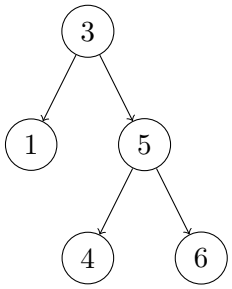
- **ancestor**
  A node that can be reached when traversing a node's parents again and again.

- **descendant (subchild)**
  A node that can be reached when traversing a node's children.

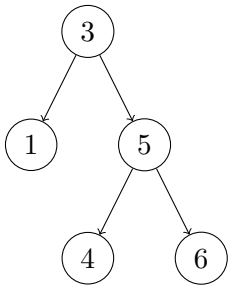
- **leaf (external node)**
  A node with no children.

- **internal node**
  A node with at least 1 child.

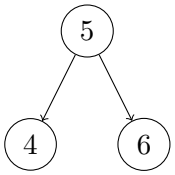
- **degree**
  The number of children a node has.
- **edge**
  The path between two nodes.
- **root**
  Extending the tree analogy is the term **root**. The **root** is the root node is the top node in a tree. In the following diagram, 3 is the root node.



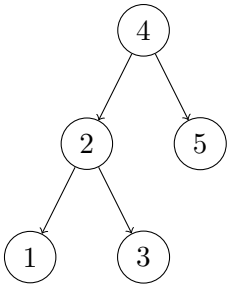
- **level**
  1 + the number of edges between this node and the root. In other words the root is at level 1, its children are at level 2, those nodes' children are at level 3, and etc.
- **depth**
  The number of edges between this node and the root; level - 1.
- **height**
  The **height** of a node is the number of edges on the longest path between this node and a leaf. The **height** of a tree is the **height** of the root node.
- **subtree**
  A tree whose root belongs to another tree. For example in this tree



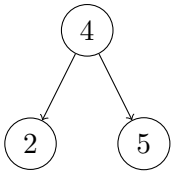
  You could say there is a subtree rooted at node 5, which is



- **full binary tree**
  A **full binary tree** is a tree where every node has two or zero children. For example the tree shown under subtree.
- **complete binary tree**
  A **complete binary tree** is a tree where every level is filled except for the last level (potentially). On the last level, the nodes are all filled in starting from the left.

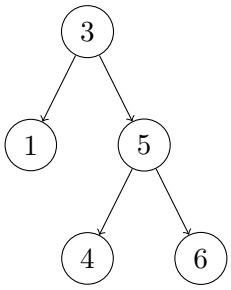


- **perfect binary tree**
  A **perfect binary tree** is a tree where all nodes have two children and all leaves are at the same level.

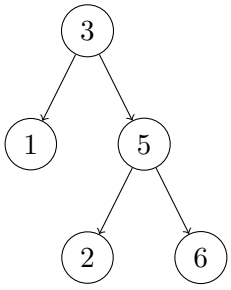


3.3 Description

One important feature of **binary search trees** is that all the nodes in the left subtree of a node A will have values less than the value at node A while the nodes of the right subtree will have greater values. For example, this is a valid **binary search tree**:



However, if we change some of the numbers it may not be valid anymore (in this case it is not valid):



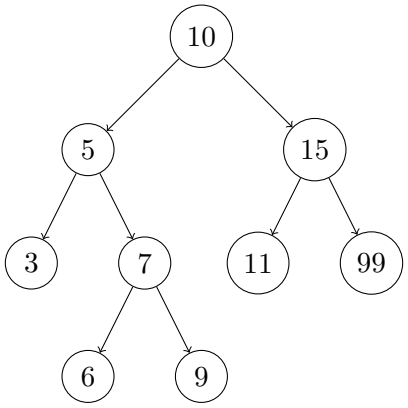
Notice by changing 4 to 2, 2 is in 3's right subtree but is less than 3. Even though the parent-child relationship is still satisfied; namely that the left child is less than the parent ($2 < 5$), this is an invalid **binary search tree**. This is a common mistake I see some students make.

Typically, there are no duplicate nodes in a **binary search tree** so the relationship is strictly less or greater, however, if duplicates are allowed I'd guess it's ok to put them in either the left or right subtrees as long as it is consistent.

Also, a **binary tree** and a **binary search tree** are not the same thing. Only a **binary search tree** has to exhibit this behaviour.

3.4 Searching

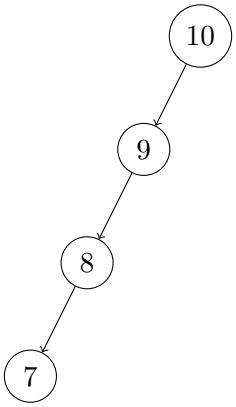
Why is this a constraint on the structure of the **binary search tree**? It's to make searching easier. Consider this slightly larger tree:



With only 3 comparisons, I can tell whether or not 14 is in the tree. Start at the root and ask if 14 is less or greater than 10. It's greater so it must be in the right subtree. Ask yourself the same thing again with the root of that subtree; is 14 greater or less than 15? It's less so it must be in the left subtree. Again, ask yourself the same questions with the new subtree's root and you'll answer that 14 is greater than 11, but 11 is a leaf node so 14 must not be in the tree. Thanks to the relationships between a node and its subtrees, we can quickly find a value if it exists.

How can we find the maximum in the tree? Just keep heading right! What about the minimum? Just go left!

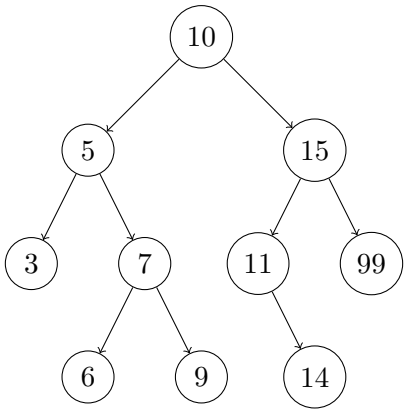
However, there can be cases where your tree is not very useful for searching. Consider something like this:



This is looking like a normal list... Depending on what value we are looking for, we may have to traverse the entire tree before finding whether or not it exists. This kind of tree is known as a **degenerate** tree. It's not very important to know right now.

3.5 Insertion

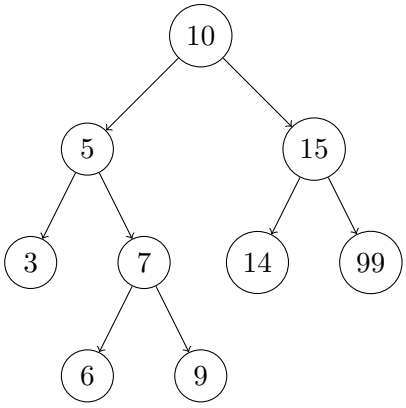
Due to the magic of a **binary search tree**, insertion is very simple. First, search for the value you want to insert and you should eventually end up at a leaf node. Then attach a node to that leaf that follows the rules. In the searching example's tree, let's say we wanted to insert 14. First we searched for it and go to 11. Now we just tack 14 onto 11 and that's that.



3.6 Deletion

Deletion is slightly more complicated. There are 3 cases you want to consider:

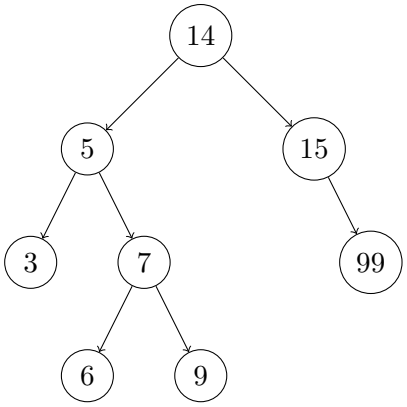
- **no children**
  Want to remove a node with no children? Just remove it! For example remove 14 from the previous tree and you just get rid of it.
- **one child** Want to remove a node with 1 child? Just remove it (and link the child with the node's parent)! For example, removing 11 in the previous example will end up looking like



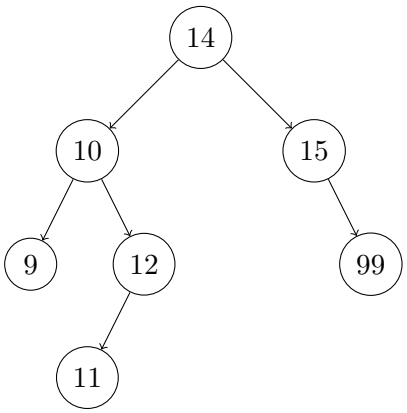
- **two children** This is probably the most difficult case. Let's try removing the root of the above tree. What value should we replace it with? We want a value that is less than all the nodes in the right subtree and greater than all the nodes in the left subtree. So what about the minimum in the right subtree or the maximum in the left subtree? They both work and the one you take depends on the **strategy** you use. One is called the **predecessor replacement strategy** and the other is called the **successor replacement strategy**. To remember which one is which, consider the numbers in the tree in an ordered list

  $$...7 \to 9 \to 10 \to 14 \to ...$$

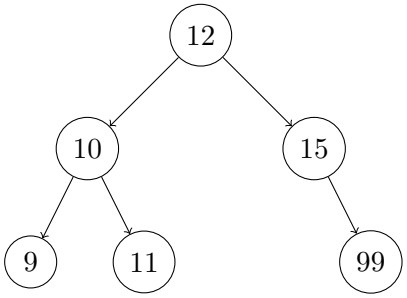
  We are removing 10 and we can replace it with 9 or 14. If we are using the **predecessor replacement strategy**, we replace it with its **predecessor** which is 9 and vice versa. So let's see what the resulting tree looks like when you use the **successor replacement strategy**.



That's not to bad is it? But wait we're not done. We also just removed 14 from that location so it's possible there will be more work (in this case, 14 was a leaf node so it was easy)! Consider this example and the **predecessor replacement strategy**



Ok so we are removing 14. Based on the strategy, we will replace it with 12, but what will happen to 11? Treat moving 12 as the deletion of 12 from where it is, so it's like delting a node with 1 child! We end up with something like this:



But is it possible to have to deal with another case where there will be 2 children? No, because to find the minimum or maximum of a tree you must keep heading right or left. You stop once there is no left or right child so the node you are replacing at cannot have that child.

3.7 Examples

Removed for now.