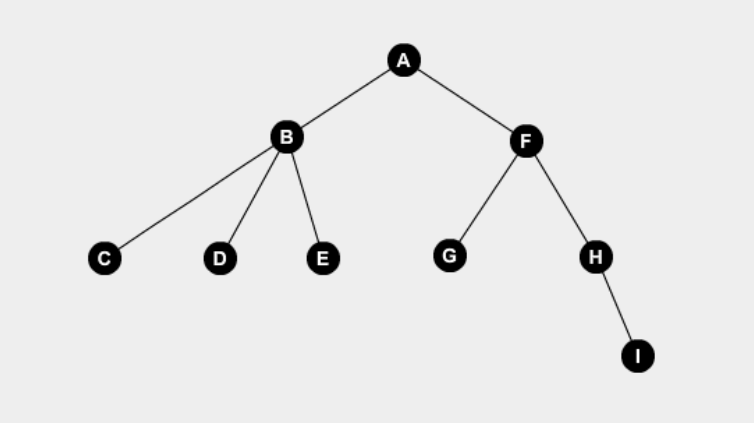
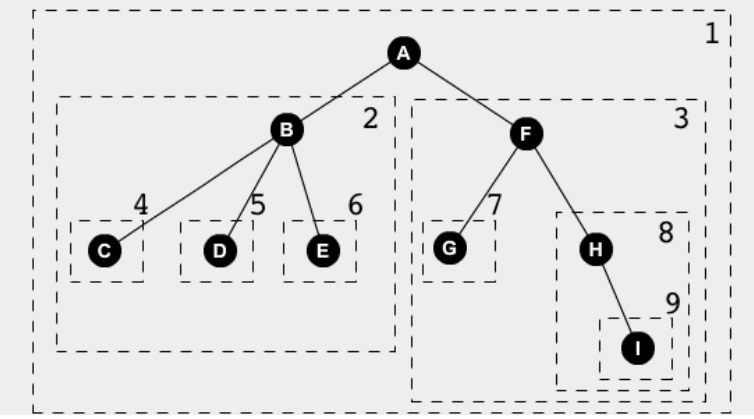
Trees
Berkeley EECS
Neil Lugovoy

What are they?
· a tree has a root and branches
· each branch is also a tree
· a tree can have no children we call this a leaf
· if we are storing data in our tree every root will have a label which holds a value

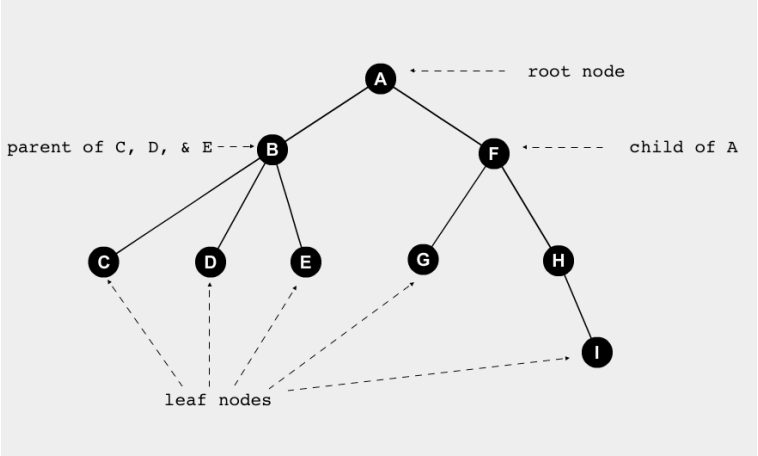
· they can also be thought of as connected locations called nodes connected by edges
· the circles are nodes and the lines are edges
· the node directly above a node is its parent and the nodes directly below a node is its children
· each node can have at most one parent
example



In this tree **A** is the root and its branches are the two trees with roots **B** and **F**.



Notice that in this figure every boxed structure is a tree. Tree **1** is the tree with root **A** and its branches are trees **2** and **3**. Similarly the branches of **3** are **7** and **8**. Note that since **7** has no branches it is a leaf.



We can also think of **A** of having children nodes **B** and **F**. Thus the parent of **B** is **A**. Note that a node can have only one child such as **H**.

What are they good for?
· great for representing relationships
· in artificial intelligence we can model possible outcomes of our choices as decision trees (see CS 188)
· many data structures take advantage of the unique properties of trees for faster operations (see CS 61B)
· in graph theory trees give us useful properties (see CS 70 and CS 170)

What makes them special?
·their definition is recursive (that is it contains itself) this allows for recursive algorithms and functions
·we have strong guarantees about their structure (eg: there is exactly one path between any two nodes, try it!)
Implementation:
If you are CS 61A, the first implementation you will use is this one:

```
def tree(label, branches=[]):
   for branch in branches:
      assert is_tree(branch)
   return [label] + list(branches)

def label(tree):
   return tree[0]

def branches(tree):
   return tree[1:]

def is_tree(tree):
   if type(tree) != list or len(tree) < 1:
      return False
   for branch in branches(tree):
      if not is_tree(branch):
         return False
   return True

def is_leaf(tree):
   return not branches(tree)
```

Thus in this representation a tree is just a list where the first element is the label and the rest of the elements are the branches which are also trees. Note that the recursive structure of trees is being used in our implementation of `is_tree`. We first check that our tree is a list and has a first element (which corresponds

to having a label) and then check if all of the branches are also trees. We are totally justified in calling function again on the branches because the branches are trees!

Imagine for a moment how this would be different if the branches were not trees. We would have to write a whole new function to have to deal with this different kind of structure.

See exercises at the end of this document for practice with this representation.

Once you get to objects you'll use this implementation.

```
class Tree:
   def __init__(self, label, branches=[]):
      self.label = label
      for branch in branches:
         assert isinstance(branch, Tree)
      self.branches = list(branches)

   def is_leaf(self):
      return not self.branches
```

If you are in CS 61B you'll encounter a similar object-oriented representation in Java. It might change from assignment to assignment but it will look something like this

```
import java.util.ArrayList;

public class Tree {
   private int label;
   private ArrayList<Tree> branches;

   public Tree(int label, ArrayList<Tree> branches) {
      this.label = label;
      this.branches = branches;
   }

   public int label() {
      return this.label;
   }

   public ArrayList<Tree> branches() {
      return this.branches;
   }

   public static boolean isLeaf(Tree t) {
      return t.branches.size() == 0;
   }
}
```

Types of trees:

Binary trees:

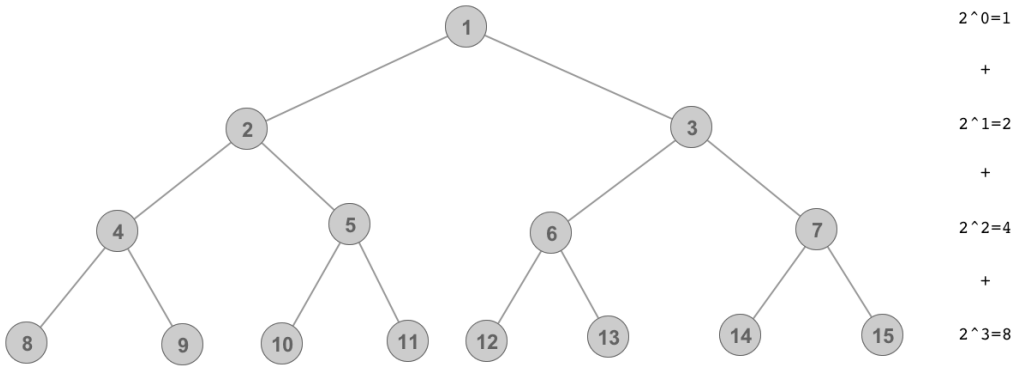
· trees such that every node has at most 2 children.
· In the previous example **3** is a binary tree while **1** is not since **B** has 3 children.

Exercise: Are there any other binary trees in the previous example?

· One of the properties of a binary tree is that there are at most $2^{h+1} - 1$ nodes in it, where h is the height

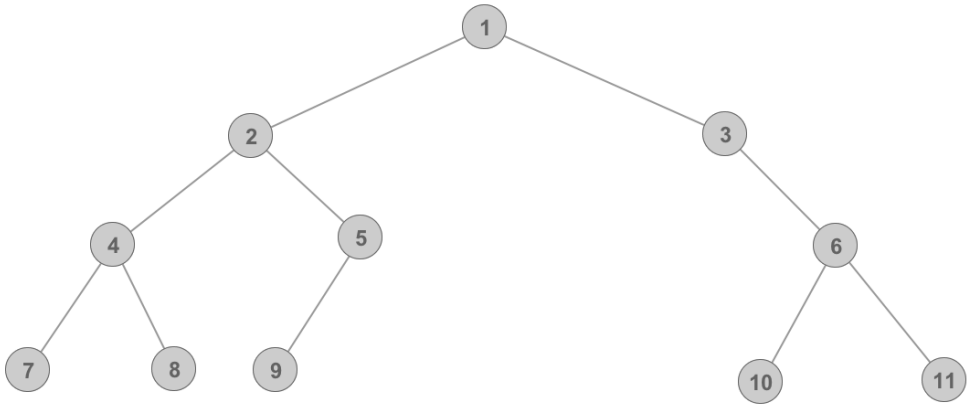
of the tree (we define height as the number of levels - 1). Let's look at why.

example



Notice that each level we have twice the number of nodes as at the previous level. So we can write this as a geometric sum $\sum_{i=0}^{h} 2^i$ which by the formula for geometric sums is $2^{h+1} - 1$. Here the height is 3 so we have 15 nodes.

Note that I said at most. This is because we are looking at what is called a perfect tree. That is every internal node has 2 children and every leaf is on the same level. This tree for example is not perfect but it still has height 4 so we can bound the number of nodes as less than 15.



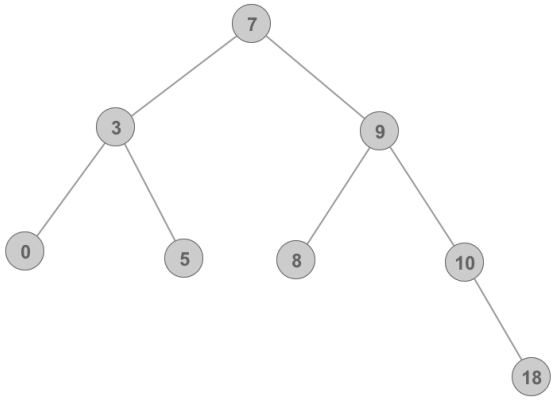
The implementation for binary trees is almost exactly the same as for regular trees except that instead of having an ArrayList or list of branches we instead keep two instance variables that point to the left and right branch. Here is the Java implementation.

```
import java.util.ArrayList;

public class BinaryTree {
   private int label;
   private BinaryTree left;
   private BinaryTree right;

   public BinaryTree(int label, BinaryTree left, BinaryTree right) {
      this.label = label;
      this.left = left;
      this.right = right;
   }

   public int label() {
```

```
      return this.label;
   }

   public BinaryTree leftBranch() {
      return this.left;
   }

   public BinaryTree rightBranch() {
      return this.right;
   }

   public static boolean isLeaf(BinaryTree t) {
      return t.leftBranch() == null && t.rightBranch() == null;
   }
}
```

Binary search trees (BSTs):

· binary trees such that the label of all nodes to the right of a node are greater than or equal to its label and the label of all nodes to the left of a node are less than or equal to its label
example



In this example everything to the left of **7**, which is **0**, **3**, **5**, is less than **7** and everything to the right of **7**, which is **8**, **9**, **10**, and **18**, is greater than **7**. Notice that also everything to the left of **9** is less than it and everything to the right of **9** is greater than it. Notice that the tree with root **9** is a BST and same with the tree with root **3**. In fact in every BST, all the branches are also BSTs!

notes on finer points of BSTs:

In BSTs labels with the same comparative value are allowed and those nodes can go on either side of each other. Also note that our labels need not be numbers, we just need something that we can compare. We could use letters and compare them alphabetically or strings and compare them lexicographically. We can even use more complex objects such as a bank account and compare them based on their available funds or date of birth of account holder or whatever we want!
exercise:
Write a method that takes as input a binary tree and returns true if the tree is a BST and false otherwise. Before you scroll down and look at what is below, give this problem a real try. Sit down and write this out.

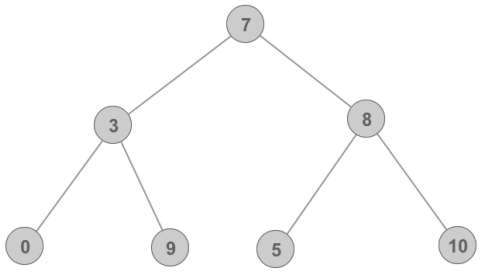
OK you can look now. Did your implementation come out to be something like this?

```
public static boolean isBST(BinaryTree T) {
```

```
   if (T == null) {
      return true;
   }
   else if (T.left != null && T.left.label > T.label) {
      return false;
   }
   else if (T.right != null && T.right.label < T.label) {
      return false;
   }
   else {
      return isBST(T.left) && isBST(T.right);
   }
}
```

If it didn't, why does this not work? If it did, this doesn't work. Can you see why? Look back at our original definition of BSTs.

Hint: try it on this tree



The problem is that everything to left a node must be less than it. This method just checks the label of the left child node (and similarly for greater and right child node). Thus even though **3** is less than **7** and **9** is greater than **3**, the problem is that **9** is to the left of **7** and it is greater than it. Similarly **5** is to the right of **7** and it is less than it. Thus when we make our recursive call we can't throw away the information that the root is **7**. Think about how you might solve this. The solution is at the end of this document.

What are BSTs good for?
To find an item in a normal length n unsorted array takes $\Theta(n)$ time on average: we just have to look through every element in the array until we find the one we are looking for or get to the end and find it isn't there. On average we go through half the array.

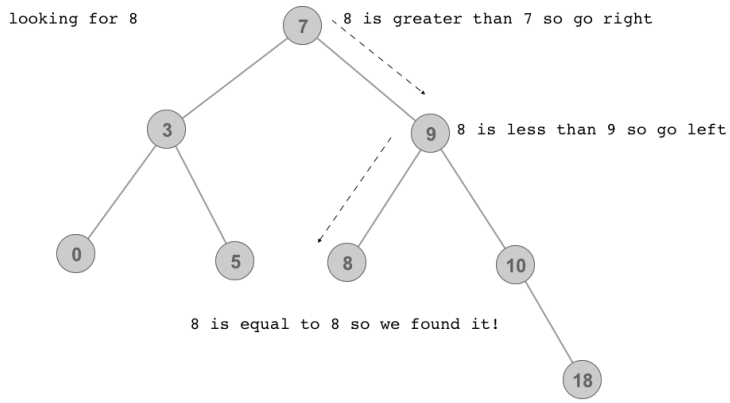
A BST lets us disregard elements we don't need to look through, by using **binary search**. To use binary search we start at the root node. If the value we are looking for is greater than the root we perform binary search on the right branch, if the value we are looking for is less than the root, we perform binary search on the left branch. If the value is equal to the root node we have found it. If we get to a leaf node that is not equal to the value we are looking for, it is not in the BST.

```
public static BST binarySearch(BST t, int item) {
   //returns the BST in t with label equal to item or null if item is not in t
   if (t == null) {
      return null;
   }
   else if (item == t.label()) {
      return t;
   }
   else if (item > t.label()) {
```

```
      return binarySearch(t.rightBranch(), item);
    }
    else {
      return binarySearch(t.leftBranch(), item);
    }
}
```

---

example:



Notice that we decided that we didn't need to look at **3**, **0**, and **5** because they were to the left of **7** and thus less than what we were looking for. Similarly we decided we didn't need to look at **10** and **18** because they were to the right of **9** and thus greater than what we were looking for.

Runtime:
Thus if our BST has about the same number of items on each side of every node, we are throwing out about half of our problem each time. Thus our runtime is proportional to the number of times we can divide n by 2 before it is less than 1. Therefore binary search on a BST with about the same number of items on each side of every node runs in $O(\log n)$ time.

Another way to think about it is that at each step we go down one level of nodes. The maximum number of steps we can take is the maximum number of levels there are in the tree minus 1. We call this the height of the tree and in a binary tree where each node has about the same number of items on each side of it, this is proportional to the base 2 logarithm of the number of nodes in the tree (see section on number of items in a binary tree). Since there are n nodes we have $O(\log n)$.

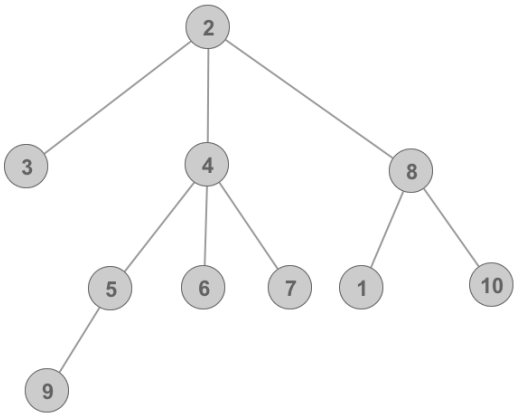
Notes on finer points of binary search:
Note that we used a big O bound because the algorithm can terminate early if we find the value we are looking for before we get to a leaf node (imagine if the value we were looking for was at the root). Also note that we kept requiring our BST to have about the same number of items on each side of every node. This property is called being balanced. What would happen if the BST was not balanced?

exercise: If the BST were not balanced then how long could binary search take? Draw out the worst case BST structure.

You'll learn about the rigorous definition of balanced and how to build self-balancing BSTs later in the course.

Exercises and solutions

1. a Using the Python list representation construct a tree that looks like this and assign it to a variable `t`.



   b Print the label of `t`.

   c Assign the circled branch to a variable `b`.

   d Draw the tree resulting from this call `a = tree(4, [tree(1), tree(2,[tree(3)]), tree(10, [tree(6,[tree(7),tree(8)])])])`

2. Are there any other binary trees in the figure on page 1 besides **3**?

3. Write a method that takes as input a binary tree and returns true if the tree is a BST and false otherwise.

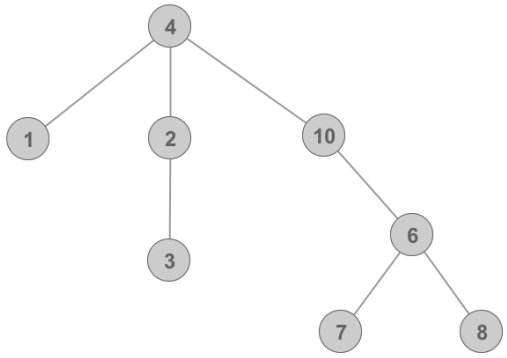
4. If the BST were not balanced then how long could binary search take? Draw out the worst case BST structure.

Solutions:

1. a
```
t = tree(2, [tree(3), tree(4, [tree(5,
[tree(9)]), tree(6), tree(7)]), tree(8, [tree(1), tree(10)])])
```

   b
```
print(label(t))
```

   c
```
b=branches(t)[1]
```



d

2. Yes **4**, **5**, **6**, **7**, **8**, and **9** are all binary trees since they all have at max 2 branches and all of their branches have at max 2 branches.
3.

```
public static boolean isBST(BinaryTree T) {
   return isBSTHelper(T, Integer.MIN_VALUE, Integer.MAX_VALUE);
}

public static boolean isBSTHelper(BinaryTree T, int min, int max) {
   if (T == null) {
      return true;
   } else if (T.label < min || T.label > max) {
      return false;
   } else {
      return isBST(T.left, min, T.label) && isBST(T.right, T.label, max);
   }
}
```

4. Imagine if all of the items were added to the BST in sorted order or reverse sorted order. Then we would just have a line of nodes each with only one parent and one child except for the end nodes. This is just a linked list!