1. Consider the Tree class below. Suppose we would like to write a method for this
Tree class, getAncestor(int k, Node target). This method takes in an integer
k and a Node target, and returns the k’th ancestor of target in our tree (you
may assume such an ancestor exists). You may also assume that \( k \geq 0 \), that
target \(!=\) null, and that there are no cycles in our tree before we call this method.

```java
public class Tree<T> {
    private Node root;

    private class Node{
        public T item;
        public ArrayList<Node> children;
    }

    public Node getAncestor(int k, Node target) {
        List<Node> list = ______________________;
        ancestorHelper(___________, ___________, __________);
        return list.get(__________________________);
    }

    private boolean ancestorHelper(______________, ______________, ______________) {
        ______________________;
        if (___________________) {
            return true;
        }
        for (______________________________) {
            if (____________________________)
                return true;
        }
        _______________________;
        return false;
    }
}
```

2. Give a bound on the runtime of getAncestor(int k, Node target) in the best
and worst cases in \( \Theta(\cdot) \) notation in terms of \( N \) and \( k \), for a tree with \( N \) nodes. How
does our choice of list implementation on line 10 affect our runtime?
Kontakte

We're going to make our own Contacts application! The application must perform two operations: `addName(String name)`, which stores a new contact, and `countPartial(String partial)`, which returns the number of contacts whose names begin with `partial`. Implement both of these methods in the `Contacts` class below. You may find the work already done in the private `Node` class, as well as the method `String::charAt(int index)` useful.

```java
public class Contacts {

    private class Node {
        public int ______________;
        public Map<Character, Node> children;

        public Node() {
            ______________;
            children = new HashMap<Character, Node>();
        }
    }

    Node root;

    public Contacts() {root = new Node();}

    public void addName(String name) {
        Node current = root;
        for (____________; ______________; ______) {
            if (______________________________) {
                Node n = new Node();
                ______________;
            }
            ______________;
        }
    }

    public int countPartial(String partial) {
        Node current = root;
        for (____________; ______________; ______) {
            if (______________________________) {
            _________________________;
            }
            else {return 0;}
        }
        return ______________;
    }
}
```
3 KND Trees

A $k$-d tree is a binary tree where each node contains a point of dimension $k$. Our goal is to create a tree of points which, when given a $k$-dimensional coordinate, can find the point closest to that coordinate (i.e. "what is the closest point to $(a, b)$?").

Each node also has a splitting plane, which is one of these $k$ dimensions. Say a node $n$ has splitting plane $x$. Then everything to the left of $n$ will have an $x$-coordinate less than or equal to $n$’s. Similarly, everything to the right of $n$ will have an equal or greater $x$-coordinate. If $n$ instead split on $y$, then the above holds for $y$-coordinates.

From the Wikipedia page for $k$-d trees, "As one moves down the tree, one cycles through the $k$ axes used to select the splitting planes."

This means in a 3-dimensional tree:
- the root would have an $x$-aligned plane
- the roots children would both have $y$-aligned planes
- the roots grandchildren would all have $z$-aligned planes
- the roots great-grandchildren would all have $x$-aligned planes, etc.

1. Consider a 2-d tree in which the root splits on $x$. Normally, we want to turn a fixed set of points into a $k$-d Tree, and we don’t have to worry about later additions. This makes it easier to make our Tree bushy. Discuss how you may do this efficiently, and draw a balanced $k$-d Tree of the points $(2, 3), (5, 4), (9, 6), (4, 7), (8, 1), (7, 2), (10, 10)$

2. What is the closest point in our tree to the coordinate $(3, 6)$? What about $(2, 5)$? What can you conclude about the worst-case runtime for closest point (otherwise known as nearest neighbor) search in a reasonably bushy $k$-d tree?