CS 61B
Spring 2019

Tries, K-D Trees, Traversals

Exam Prep 9: March 18, 2019

1 Mushu

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.

```
1   public class Tree<T> {
2       private Node root;
3       private class Node{
4           public T item;
5           public ArrayList<Node> children;
6       }
7       public Node getAncestor(int k, Node target) {
8           List<Node> list = new LinkedList<Node>();
9           ancestorHelper(target, root, list);
10          return list.get(list.size() - 1 - k);
11      }
12      private boolean ancestorHelper(Node target, Node current, List<Node> list) {
13          list.add(current);
14          if (current == target) {
15              return true;
16          }
17          for (Node child : current.children) {
18              if (ancestorHelper(target, child, list))
19                  return true;
20          }
21          list.remove(current);
22          return false;
23      }
24  }
25
```

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?

Best case is $\Theta(1)$ when `target == root`. Worst case is $\Theta(N)$ when we traverse the entire tree. A `LinkedList` is optimal here since we only add/remove from the end of the list, whereas we would still have resizing penalties if using an `ArrayList`.

2 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.

```
public class Contacts {

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

        public Node() {
            numWords = 0;
            children = new HashMap<Character, Node>()
    }

    Node root;

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

    public void addName(String name) {
        Node current = root;
        for (int i = 0; i < name.length(); i++) {
            if (!current.children.containsKey(s.charAt(i))) {
                Node n = new Node()
                current.children.put(s.charAt(i), n);
            }
            current = current.children.get(s.charAt(i));
            current.numWords++;
        }
    }
    public int countPartial(String partial) {
        Node current = root;
        for(int i = 0; i < partial.length(); i++) {
            if(current.children.containsKey(partial.charAt(i))) {
                current = current.children.get(s.charAt(i));
            }
            else {return 0;}
        }
        return current.numWords;
    }
}
```

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.

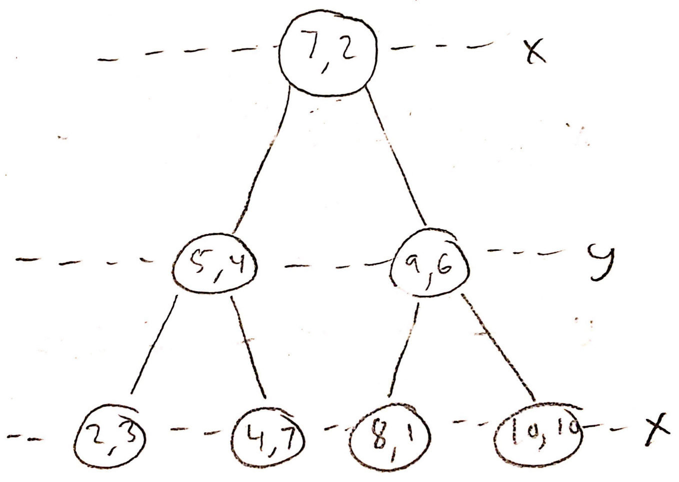
From the Wikipedia page for k-d trees, "As one moves down the tree, one cycles through the k axes used to select the splitting planes.".

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.

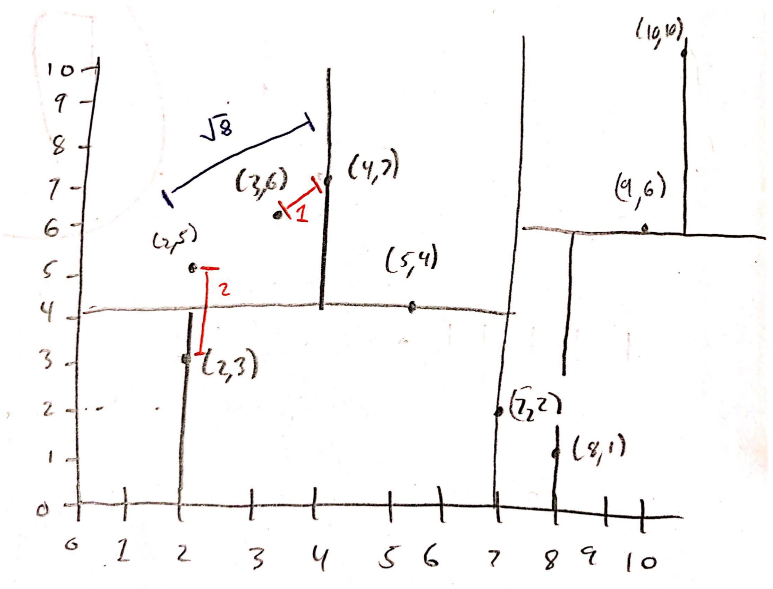
1. Consider a 2-d tree in which the root splits on x. Normally, we want to turn a fixed set of points into a k-d Tree, and we don't have to worry about later additions. This makes it easier to make our Tree bushy. Discuss how you may do this efficiently, and draw a balanced k-d Tree of the points $(2,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

To efficiently construct a k-d Tree from a list of points, we first find the point with the median x value, $(7, 2)$, and elect it as our root. To determine the subtrees to the left and right of this root, we recurse on each half of the list which straddles this median (i.e. $\{(2, 3), (4, 7), (5, 4)\}$, and $\{(8, 1), (9, 6), (10, 10)\}$). We also now consider the y plane. This means we would elect the points with the median y value in each of this list, $(5, 4)$ and $(9, 6)$ as the roots of these subtrees. Our final k-d tree looks like this:



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?

Here's what the points look like on the $x - y$ plane:



It turns out that $(3, 6)$ and $(2, 5)$ are the same corner of the plane but have different nearest neighbors. The closest point to $(3, 6)$ is $(4, 7)$, which happens to be the last node we would have passed in looking for $(3, 6)$ greedily. However, the closest point to $(2, 5)$ is $(2, 3)$. Indeed, we cannot just greedily traverse a k-d tree in the same way we traverse a BST, and in the worst case would visit $\Theta(N)$ points. This could occur, for example, when our coordinate lies on the left side of the root, but is closest to a point on the opposite side, we may have to visit roughly $\frac{1}{2}N$ points.

For rules on how to perform nearest neighbor search, as well as when branches can be pruned, check out this section of the wiki as well as Professor Hug's lecture slides

Here is an example of how we may verbosely calculate the nearest neighbor of $(3, 6)$:

1. $\texttt{currPoint} := (7,2), \texttt{bestDist} := \infty, \texttt{bestPoint} := \texttt{null}$
2. Calculate $\texttt{d} := \texttt{dist}((3,6), \texttt{currPoint}) = \texttt{dist}((3,6),(7,2)) \approx 5.7$
3. Since $5.7 < \infty$, we set $\texttt{bestDist} = 5.7, \texttt{bestPoint} = (7,2)$
4. Since $3 < 7$, we recurse on the left subtree and set $\texttt{currPoint} = (5,4)$
5. Calculate $\texttt{d} := \texttt{dist}((3,6),(5,4)) \approx 2.8$
6. Since $2.8 < 5.7$, we set $\texttt{bestDist} = 2.8, \texttt{bestPoint} = (5,4)$
7. Since $6 > 4$, we recurse on the right subtree and set $\texttt{currPoint} = (4,7)$
8. Calculate $\texttt{d} := \texttt{dist}((3,6), \texttt{currPoint}) = \texttt{dist}((3,6),(4,7)) = \sqrt{2}$
9. Since $\sqrt{2} < 2.8$, we set $\texttt{bestDist} = \sqrt{2}, \texttt{bestPoint} = (4,7)$

`dist`$((3,6),(3,4)) = 2 \geq \sqrt{2}$, meaning we don't have to check below $(5,4)$. Note that if we instead found that `dist`$((3,6),(3,4)) <$ `bestDist`, then we would recurse on the left subtree of $(5,4)$. As well, `dist`$((3,6),(7,6)) = 4 \geq \sqrt{2}$, so we don't have to check to the right of $(7,2)$. So, we're done and `nearest`$((3,6)) = (4,7)$.