CS2040S: Data Structures and Algorithms

Discussion Group Problems for Week 7

For: March 3–March 8

1 Check in

Discuss questions, if you have any, with the tutor and the rest of the class, about the material and content so far.

2 Problems

Problem 1. AVL vs Trie

Discuss the trade-offs of using AVL and Trie to store strings.

Solution:

- Time complexity for insert, delete and find a word with length L to a collection of N words: $O(L \log N)$ for AVL and $O(L)$ for trie.
- Space complexity is $O(L_{\text{total}})$ for both AVL and trie where $L_{\text{total}} = total\_string\_length$. However, trie tends to have more overhead cost.

Problem 2. kd-Trees

A kd-tree is another simple way to store geometric data in a tree. Let's think about 2-dimensional data points, i.e., points (x, y) in the plane. The basic idea behind a kd-tree is that each node represents a rectangle of the plane. A node has two children which divide the rectangle into two pieces, either vertically or horizontally.

For example, some node v in the tree may split the space vertically around the line $x = 10$: all the points with x-coordinates ≤ 10 go to the left child, and all the points with x-coordinates > 10 go to the right child.

Typically, a kd-tree will alternate splitting the space horizontally and vertically. For example, nodes at even levels split the space vertically and nodes at odd levels split the space horizontally. This helps to ensure that the data is well divided, no matter which dimension is more important.

There are 2 main ways to implement kd-tree.

In one way, internal nodes of the tree corresponds to the horizontal or vertical splitting lines; When you have a region with only one data point, instead of dividing further, simply create a leaf which corresponds to the only data point. Figure 1 is an example of a kd-tree that contains 14 points (internal nodes without any label can be omitted):

Another way to implement kd-tree uses horizontal or vertical lines that pass through data points for splitting. In this case, each tree node (internal and external) represents one data point. Figure 2 is an example of a kd-tree that contains 10 points:

Problem 2.a. How do you search for a point in a kd-tree? What is the running time?

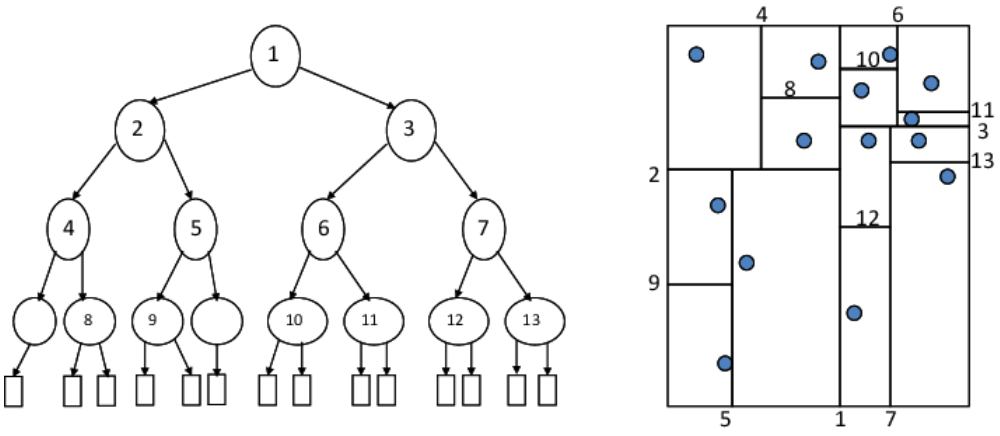

Figure 1: On the left: the points in the input. On the right: how the points are stored in the kd-tree

Solution: Start at the root. At each node, there is a horizontal or a vertical split. If it is a horizontal split, then compare the x-coordinate to the split value, and branch left or right. Similarly, for a vertical split, compare the y-coordinate to the split value, and branch up or down. The running time is just $O(h)$, the height of the tree.

Problem 2.b. You are given an (unordered) array of points. What would be a good way to build a kd-tree? Think about what would keep the tree nicely balanced. What is the running time of the construction algorithm?

Solution: Basic approach:
We can think of the construction recursively. At a given node, we have a set of points, and we need to split it horizontally or vertically. (We have no choice: that depends on whether it is an even or odd level.) Therefore, you might sort the data by the x or y coordinate (depending on whether it is a horizontal or vertical split), choose the median as the split value, and then partition the points among the left and right children. The running time of this is $O(n \log^2(n))$, since you spend $O(n \log n)$ at every level of the tree to do the partitioning, i.e., the recurrence is $T(n) = 2T(n/2) \ + O(n \log n)$.

Solution: How to do better:
Instead of sorting at every level, we could either (1) choose a random split key (if the data points are uniformly scattered), or (2) Use QuickSelect to find the Median. Then, the partitioning step is only $O(n)$, and so the total cost is $T(n) = 2T(n/2) \ + O(n) = O(nlogn)$.

Problem 2.c. How would you find the element with the minimum (or maximum) x-coordinate in a kd-tree? How expensive can it be, if the tree is perfectly balanced?

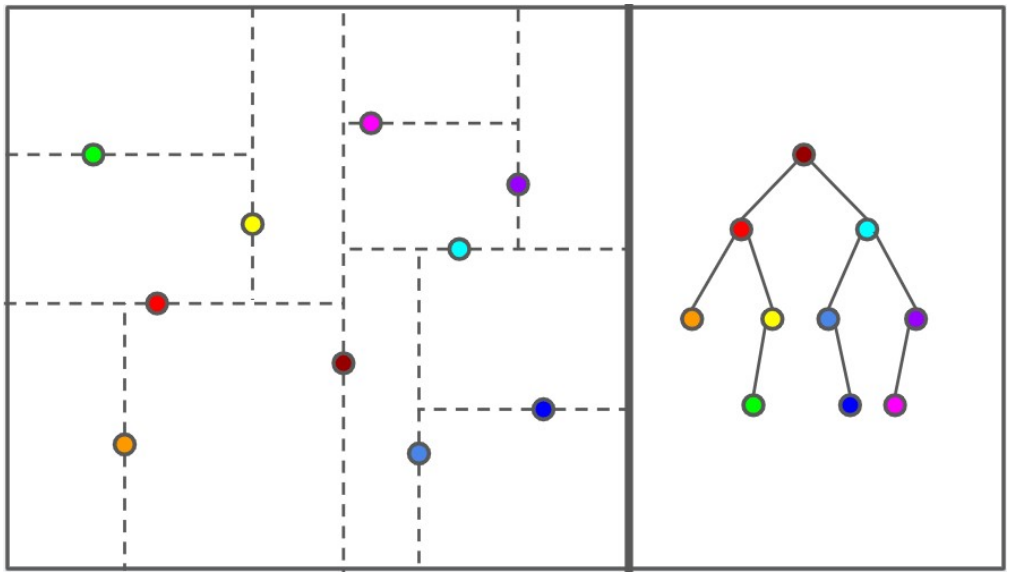

Figure 2: On the left: the points in the input. On the right: how the points are stored in the kd-tree

Solution: To find the minimum, if you are at a horizontal split, it is easy: simply recurse on the left child. But, if you are at a vertical node, you have to recurse on both children, since the minimum could be in either the top half or the bottom half. (Write out the recursive pseudocode.) To find the running time, let's look at the recurrence from taking two steps of the search (one horizontal and one vertical): $T(n) = 2T(n/4) + O(1)$. At each step down the tree, the number of points divides in half, i.e., $n/2$ after one step and $n/4$ after two steps. After two steps of the search, there are two more recursive searches to do. Solving this recurrence, you get a recursion tree that is depth $\log(n)/2$, each node has cost $O(1)$, and there are $O(2^{\log(n)/2})$ nodes in the tree, so the total cost is $O(\sqrt{n})$.
For those of you who want to get a head start on more future applications (CS2109S being one of them) and more kd-tree fun, think about how you would search for the nearest neighbor of a point P! Main idea is to recurse down the k-d tree with a variable of the closest point C found so far. Do not recurse subtrees whose bounding rectangles cannot contain any point closer than C (this is called pruning). We can further improve this by searching the subtrees in order that maximizes the chances for pruning. Though the runtime can still be $O(n)$ in the worst case, it is closer to $O(\log n)$ for 2d data. Similiar idea can also be used to find k-nearest neighbors. k-d tree is commonly used for optimising the implementation of k-nearst-neighbor, one of the most popular and simplest supervised learning classifier.

Problem 3. Tries(a.k.a Radix Trees)

Coming up with a good name for your baby is hard. You don't want it to be too popular. You don't want it to be too rare. You don't want it to be too old. You don't want it to be too weird.[1]

Imagine you want to build a data structure to help answer these types of questions. Your data

[1]The website https://www.babynamewizard.com/voyager let's you explore the history of baby name popularity!

structure should support the following operations:

- `insert(name, gender, count)`: adds a name of a given gender, with a count of how many babies have that name.
- `countName(name, gender)`: returns the number of babies with that name and gender.
- `countPrefix(prefix, gender)`: returns the number of babies with that prefix of their name and gender.
- `countBetween(begin, end, gender)`: returns the number of babies with names that are lexicographically after `begin` and before `end` that have the proper gender.

In queries, the gender can be either boy, girl, or either. Ideally, the time for `countPrefix` should not depend on the number of names that have that prefix, but instead run in time only dependent on the length of the longest name.

Solution: The point here is to use a trie, and store in each node in the trie the count of names under that node, for each gender. Don't forget that when you insert strings into the trie, you have to update the counts at the nodes in the trie.

- `countPrefix` just involves going down to the node and return the count stored at the node.
- For `countName`, search for the node, check if the end of word flag is set to `true`. If it is false, return 0. Otherwise, the answer is given by subtracting the count of the node by the total count of its child nodes.
- For `countBetween`, find `rank(begin, gender)` and `rank(end, gender)` respectively by traversing upwards from `begin` and `end` first. Then return `rank(end, gender) - rank(begin, gender) - countName(begin, gender)`, where `rank` is defined as follows:

```
rank(node, gender):
r ← 0
Traverse from x = node to root:
   If parent of x is a name:
      r ← r + parent.countName(gender)
   For all left siblings x' of x:
      r ← r + x'.count(gender)
return r
```

(Note: parent.countName(gender) is not the same as defined in the question)

Problem 4. A Trie Question?

Problem 4.a. Your task is simple. Given an array of 32 bits unsigned positive integers, find 2 numbers such that their XOR is maximum.

- Hint 1: Look at the title of the question.

- Hint 2: Think of numbers bit by bit from the most significant bit.

Solution: Here we introduce the concept of a Bit Trie. A bit trie basically stores integers in their binary representation from their most significant bit to the lowest bit. We thus want to support 2 queries: 1) Add an integer to the data structure. 2) Given an integer x, find a value y in that trie such that x XOR y is maximised. Suppose we have such a data structure, the question is trivially solved. Adding an integer is simple, we just add from the most significant bit to the least significant, like how you would do any trie. Finding the best value y to XOR with x is slightly more tricky, you would want to go down the path that represents the most optimal number when XORed with x. You can do this greedily. For example, if your current bit value is 0, you would want to go down the bit with 1 if it exists, if not you go down the 0 bit path.
For more details, you can refer to: https://www.geeksforgeeks.org/maximum-xor-of-two-numbers-in-an-array/

Problem 4.b. (Bonus, optional, difficult) (Source: https://codeforces.com/contest/1777/problem/F)

Now, we modify the question. You are given an array of 32 bits unsigned positive integers A. Find a subarray of A(l,r) such that the following value is maximised:

$$A_l \oplus A_{l+1} \cdots \oplus A_r \oplus max(A_l, A_{l+1}, \ldots A_r)$$

Solution: This is a rather challenging question so feel free to skip it. Here, we assume knowledge of the Bit Trie as defined in the previous part. Also, it features a clever use of Divide and Conquer which may be new to most students.
Similar to merge sort, we can do divide and conquer to solve problems like this. Suppose we split the array at the middle into 2 sections, X and Y. Suppose we know the maximum answer from X and Y. (The divide part). Now, we are left to find a maximum subarray such that it is not contained within X and Y, ie, the left bound is in X and the right bound is in Y. (the Combine part). The answer is thus maximum from X, Y and the the combination part. Now, we need to solve the combine part fast and then the answer will just be an additional log factor. We can show that by using the data structure above, we can solve that part in $O(32n)$ time, giving us a time complexity of $O(32nlogn)$.
You start with 2 pointers starting at the centre, one would go to the right, the other the left. We assume first the maximum value will be at the right part. Then, for everytime we move the right pointer, we do the following, we update our current maximum and the current right XORSUM, then we would keep moving the left pointer to cover all XORSUMs in the left which does not contain a value bigger than the right. We add all these XORSUMs in the bit trie. Then, once we are down, we just query it with the XOR of current max and current right XORSUM. This will give us the optimal answer if the array ends at pointer r. We do this until the end. Afterwards, we repeat but considering the maximum to be on the left.
Note: The editorial uses a slightly different approach and uses small to large merging. The approach mentioned above is what the I used to AC the problem. For more details, you can refer to my original submission: https://codeforces.com/contest/1777/submission/190038613 (in C++)