Meilin, Rebecca, Sommer, Katie
FINAL PROJECT

Table of contents:

In this project, we will implement two data structures in C, then analyze their potential for overlap in various scenarios. We will conduct research on how they may have been used together in other scenarios and design several small tests to analyze what ideas have potential for overlap, highlighting key features of the two structures.
Link to GitHub Repo: https://github.ccs.neu.edu/lowenk/5008_GroupDataStructures

The two data structures we have chosen to analyze for the final project are a K-D Tree and a Trie.

1. K-D Tree Introduction and Background

We plan to construct two variations of an implementation of a k-d tree. The first implementation will work with two dimensional coordinates (x,y) and the second will work with three dimensional coordinates (x,y,z).

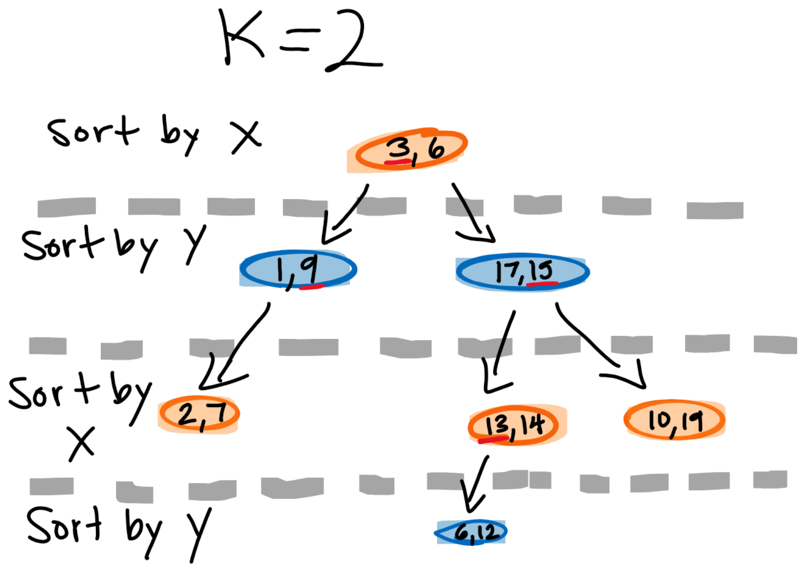
Functions in the Implementations:

- Add/Insert
- Search
- Delete
- Print
- Find Nearest Neighbour

K-d trees, where k represents the number of dimensions in a data set, are well known space partitioning trees. In a k-d tree, the node values come as a coordinate. For example, if we were looking at a two dimensional space, our coordinates would be presented as (x,y). For placements in the tree, the structure alternates its comparing of x or y values on each level in the tree. This structure lends itself differently than other tree structures as it allows us to find the nearest neighbour of a given coordinate in our tree. The function of finding a nearest neighbour lends itself well to graphical contexts such as ray tracing, finding the nearest photon queries in photon mapping, and using the nearest neighbor search in point cloud modeling and particle-based fluid simulation.

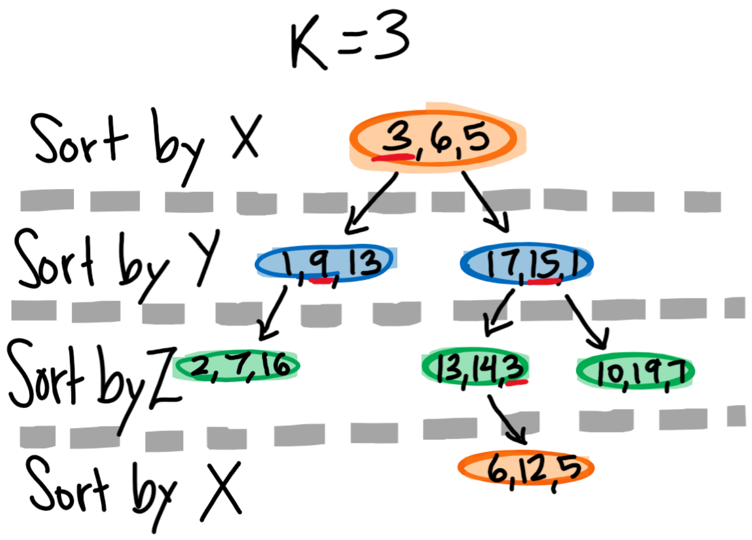
With an interest in Mixed Reality, the understanding of spatial dimensions and how they are related to one another sparked an interest to explore the construction and application of a k-d tree. One of the main applications of a k-d tree is photon mapping. Photon mapping uses the global illumination rendering algorithm to add more realistic lighting to three dimensional scenes. The algorithm is used to “realistically simulate the interaction of light with different types of objects,” and a k-d tree is then able to plot this information in relation to other plot points.

In a k-d tree, k refers to the number of elements each point has. For example, if we are just working with (x,y) coordinates, k=2 while if we are working with (x, y, z) coordinates, k=3. As shown below, we will alternate layers by sorting for one of the given coordinates. So, when k=2 we alternate sorting by x and y, while when k=3, we alternate sorting by x, y, and z. The following k=2 k-d tree is for the data set: (3, 6), (17, 15), (13, 14), (6, 12), (1, 9), (2, 7), (10, 19)

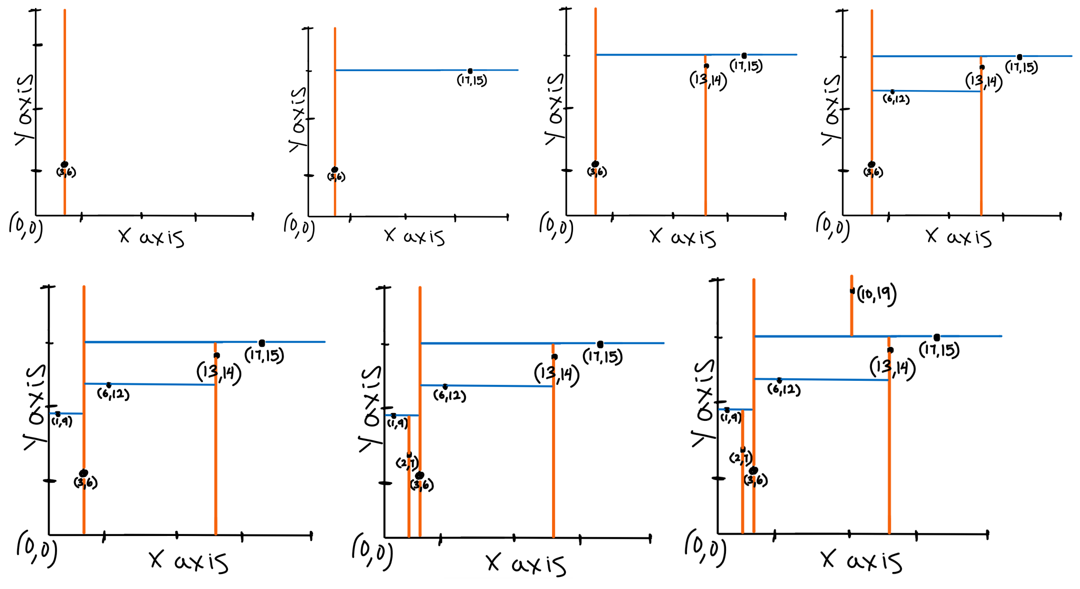


Note how we start with our first value, (3, 6) at the root. When we add the second value, (17, 15), we start by comparing the x value we are adding with the x value of the root: 17 > 3. This means we can place this node to the right, where larger values go. When we add our next value (13, 14), we start in the same fashion, comparing x values with the x we are adding and the root: 13 > 3. So we traverse to the right child. Next we need to compare the node here with the coordinate we are adding. However, this time we compare y values instead of x values: 14 < 15, so we can traverse to the left and place our coordinate there. As the above diagram illustrates, this pattern continues, alternating whether we sort by comparing x or y, depending on the depth of the tree.

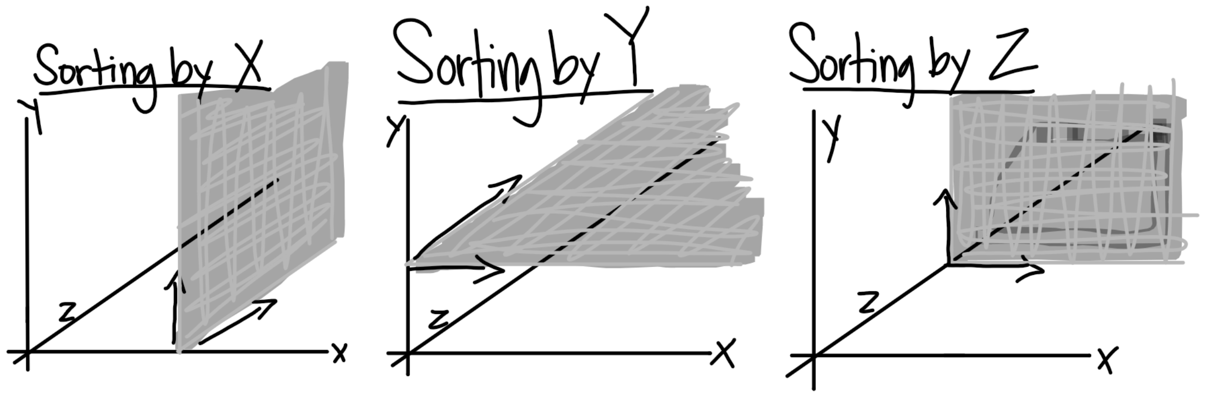
When we have a k of 3, we follow this same pattern, only now we will have three layers of sorting that we alternate among: x, y, and z. This can be observed in the tree below, where we used the following dataset: (3, 6, 5), (17, 15, 1), (13, 14, 3), (6, 12, 5), (1, 9, 13), (2, 7, 16), (10, 19, 7)



Another way to visualize this data is on a graph. We could create a two dimensional graph with the k=2 data, or a three dimensional graph with the k=3 data. For now, let us return to our k=2 data set and use a graph to visualize what our k-d tree is doing when it organizes the coordinates into a tree:



Note that in the first image, we insert out first and root node (3, 6). All the area to the right of it is larger than 3, and all the area to the left is smaller than 3. We draw a vertical line here to represent the x value that we sort by at the root node. We then insert our second point, (17, 15). 17 is larger than 3, so our point will be on the right, naturally. Now we draw a horizontal line at y = 15, because this is the y value we will use to sort all items that come to the right side of our initial x = 3 line. Note that our y = 15 line is only drawn on the right side of y = 3. We can see how this pattern continues for the rest of the graph, with a vertical line being added for all nodes that are sorting by x and a horizontal line added for all nodes that are sorting by y. In the case of k=3, we could represent this with a 3d graph, where the lines are planes: to represent sorting by x we would have a y-z plane instead a of a vertical line, to represent sorting by y we would have an x-z plane, and to represent sorting by z, we would have an x-y plane:



As k increases, visualization in this fashion becomes increasingly difficult to draw. To understand a k-d tree with an increasing k, it can help to think of it in context. Imagine a patient database with ten unique statistics for each patient. Say, we want to find other patients that have the most similar conditions for all of the 10 categories. In this case, we can use a k-d tree to find those nearest neighbors. Our k would be 10, for the 10 categories, and each node would represent a person. We will explore potential applications later in this paper, but now we will explore our design choices in our own implementation.

2. K-D Implementation and Design Tradeoffs

Insert Functionality (x, y)
In insert() we check to see if there is a tree root node that exists. If there is none, we add in our newNode as the root node. If there is a root, we invoke insertHelperX(), as we start with the x values in this tree.

With insertHelperX(), we compare the x values in newNode and the root node (the name treeNode refers to the node currently in the tree we are comparing with. In this case treeNode would be our root node). If the newNode x data is less than the x data in the root node, then we traverse left. Otherwise, we traverse right. This traversal updates our treeNode value.

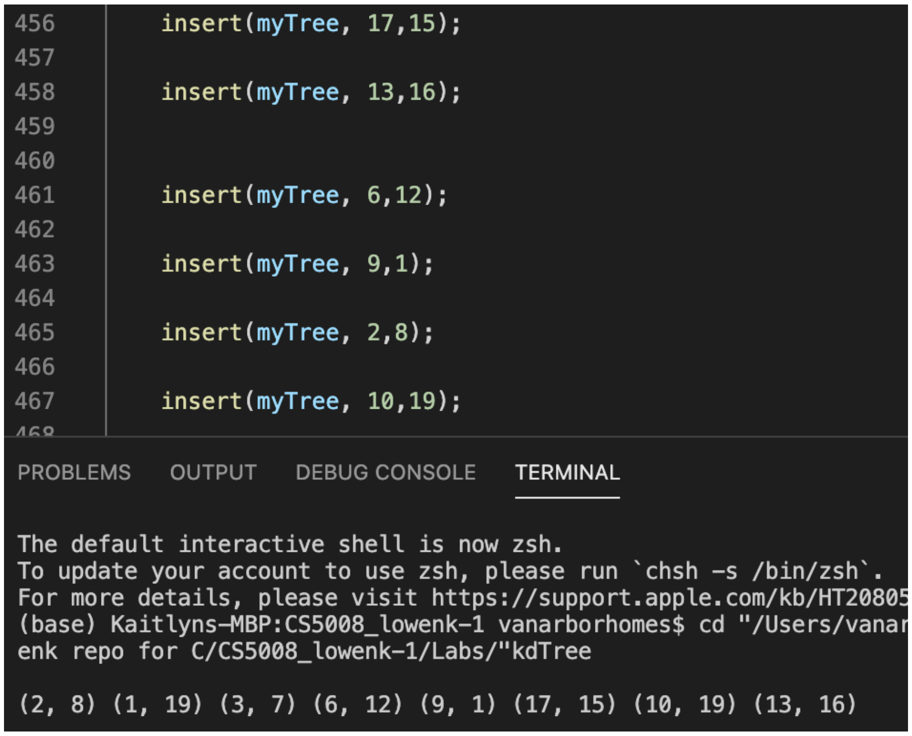
After traversing left, for example, we check to see if there is a node there (check for NULL value.) If there is no node, we add our newNode here. If there is a node, this node becomes treeNode, and we invoke insertHelperY().

With insertHelperY(), we compare the y values in newNode and the y value of treeNode (the left child of our previous step). If newNode's y is less than treeNode's y, we traverse to the left child of treeNode. If newNode's y is greater than treeNode's y, we traverse to the right child of treeNode. The child node we traverse to becomes our new treeNode.

We will keep alternating back and forth between insertHelperX() and insertHelperY() until we reach a NULL point in the tree. Here, we will add newNode into our tree.

It is worth noting that we will never have to insert a node into the middle of the tree and refactor everything downstream of that node, because our tree is not self balancing. The only thing that bounds the placement of the node is our current nodes having at most two children.

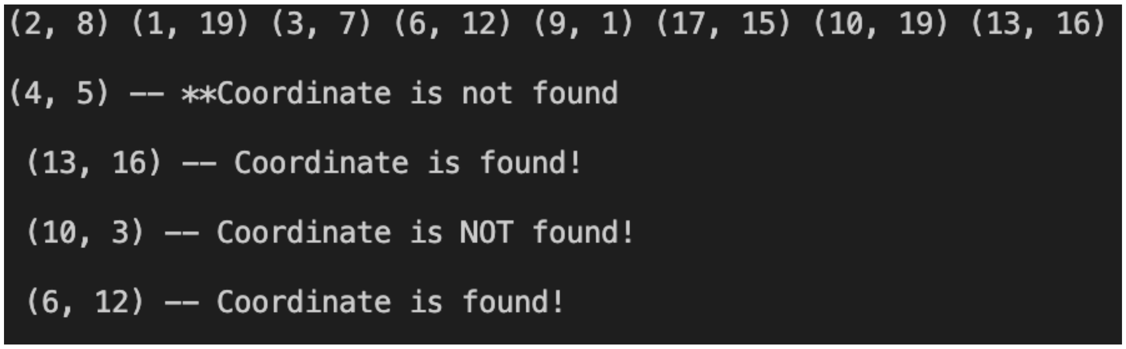
In the best case scenario, best case time complexity for insertion would be log(n) time because that is the height of the ideal binary search tree, with two children per node. In the worst case time complexity for insertion will be O(n), if all of the nodes are only right or only left children.



```
    insert(myTree, 17,15);

    insert(myTree, 13,16);


    insert(myTree, 6,12);

    insert(myTree, 9,1);

    insert(myTree, 2,8);

    insert(myTree, 10,19);
```

```
The default interactive shell is now zsh.
To update your account to use zsh, please run `chsh -s /bin/zsh`.
For more details, please visit https://support.apple.com/kb/HT20805
(base) Kaitlyns-MBP:CS5008_lowenk-1 vanarborhomes$ cd "/Users/vanar
enk repo for C/CS5008_lowenk-1/Labs/"kdTree

(2, 8) (1, 19) (3, 7) (6, 12) (9, 1) (17, 15) (10, 19) (13, 16)
```

Search Functionality (x, y)
The insert functionality follows almost exactly the same form as the search functionality, however, instead of looking for a place to insert our value, we are looking to see if our value already exists in the tree. Thus, there will be a conditional to return true if our x and y values happen to be equal to a given point in the tree. The best case time complexity for search will be log(n), and in the worst case O(n) similar to insert because of the similarities in their structures.



```
(2, 8) (1, 19) (3, 7) (6, 12) (9, 1) (17, 15) (10, 19) (13, 16)

(4, 5) -- **Coordinate is not found

 (13, 16) -- Coordinate is found!

 (10, 3) -- Coordinate is NOT found!

 (6, 12) -- Coordinate is found!
```

Print Functionality (x, y)
For the print function, we can print our tree, or single x and y values. We included the functionalities printX() and printY() for the case where we might want to print one coordinate but not the other.

For printing the tree, we start at the root and print the (x, y) coordinate pair. Next, we do an in order traversal, printing left to right on each layer. We will note here that because our tree levels alternate being sorted by x and y, at first glance our print statement may not look like a typical in order traversal.

In our print function, we first check if the tree is null, and if it is not we call our printHelperTree() on the root of the tree. PrintHelperTree() takes in a node and recursively calls the function on the left subtree, and then on the right subtree. Print time complexity will be O(n) because we print each node once.

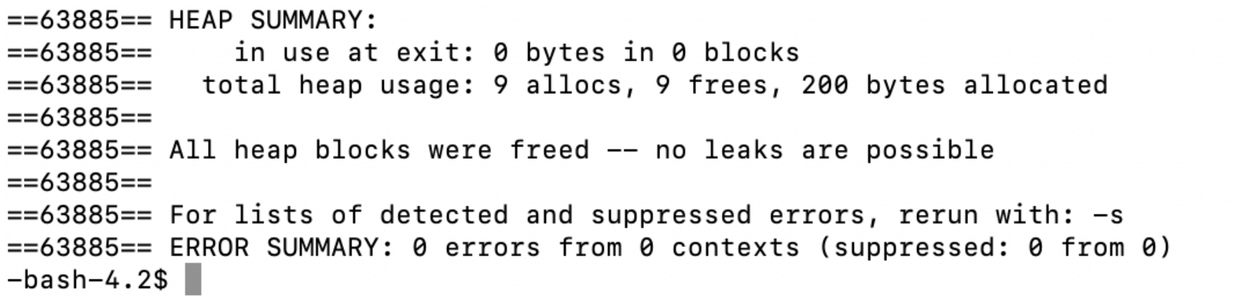
Nearest Neighbor Functionality (x, y)
We call this method findNearestNode(). First, it checks to see if the tree is null. If the tree is not null, then it moves to a set of conditionals.

Our nearest neighbor takes in a tree and data point, then calls nearestNeighbor() on the root. NearestNeighbor() does a BFS search through the tree to establish static variables: smallest distance, and the set of (x,y) points that it belongs to.

When we find a smaller distance, update the distance and the (x,y) coordinates that belong to the point that contains the smallest distance.

The time complexity for the nearest neighbor functionality will be O(n), as we iterate through each of the points and use that data to calculate the distances.

Freeing the Tree
The freeing function follows a BFS recursive structure. Each node is visited and freed individually, and finally the tree itself is freed. The photo below is the summary of our valgrind analysis, confirming that we have no memory leakage.



```
==63885== HEAP SUMMARY:
==63885==     in use at exit: 0 bytes in 0 blocks
==63885==   total heap usage: 9 allocs, 9 frees, 200 bytes allocated
==63885==
==63885== All heap blocks were freed -- no leaks are possible
==63885==
==63885== For lists of detected and suppressed errors, rerun with: -s
==63885== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
-bash-4.2$
```

K-d tree tradeoffs
When creating our k-d tree, there were several design decisions and tradeoffs that we had to consider.

The first tradeoff was balancing optimization with writing code that would be easier to debug and understand. We ran into this trade off when we were designing our helper functions, helperX() and helperY(). At first we planned to write one helper function that contained functionalities for both x and y. However, we decided that because we were writing an implementation for just a k = 2 k-d tree, we could hard code the helpers for x and y into our program to make it easier for us to understand. This decoupled some of the functionality and though it led to more functions and perhaps more code, it adhered to the principle of one responsibility per function.

This approach of hard coding in a helper function per node elements works for a k-d tree with k = 2 or k = 3, which would give us x and y helper functions or x, y, and z helper functions. However, if we are creating an implementation for an unknown k that actually takes in a k value from the user, the problem becomes much more difficult, and in that case it would help drastically to have a single helper function, rather than needing to automate the building of k helper functions.

One of the benefits of using a k-d tree is that it can help with the nearest neighbor function. Starting with the root node, a k-d tree can split a data set in half, eliminating the half of the data set that does not contain the point. As we were coding our nearest neighbor function, we chose to use a breadth first search to get the data for each point and use a distance equation to find the distance between our point in question and each of the other points. This gave us a complexity of O(n) because we had to go through each of the points. We realized after creating this function and analyzing it that this was not the best way to search the nearest neighbor- we had looked over one of the key features of the k-d tree, which is that it is a binary tree and can eliminate half of the elements in question from our search. If we had created this function by using this feature that the tree provides, our time complexity could have been closer to log(n), the height of the tree. However, our approach to coding this function did help with understanding the importance of the key features in a k-d tree.

In this implementation, we decided not to create a delete function. Instead, we focused on the other components of the implementation. Our rationale was that the primary purpose of this tree was to search for a nearest neighbor, and that it was less relevant to be able to delete data points from the tree. Further, while it would take much longer than simply deleting a point, it is possible to simply create a new tree, leaving out the data point that we don't want.

A final significant tradeoff in this implementation is that our tree is not self balancing. This allowed us for easier coding as we created the tree, but it does mean that we miss out on having the benefits of a self balancing tree.

NOTE: We are using the C math.h header file in our code. To compile and run the kdTree code, use the following in the command line:

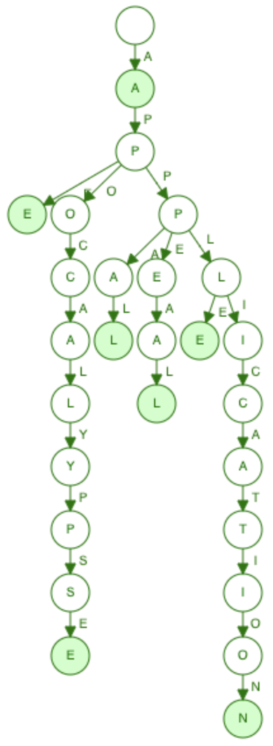
```
gcc kdTree.c -o kdTree -lm
```

3. Trie Introduction and Background

We plan to construct an implementation of a Trie. This data structure is in nature a search tree where each node has a maximum of 26 children (determined by the size of the alphabet) and strings are stored in a top to bottom manner.

Functions in the Implementations:

- Insert
- Search
- Delete
- Print
- Find Nearest Full Word(s)



The visual to the right is from https://www.cs.usfca.edu/~galles/visualization/Trie.html.

Tries, coming from the word "re**trie**ve", are well-balanced data structures that are used in predictive text and autocorrect algorithms. They store strings character-by-character in a tree structure, which makes it easy and fast to look up words and determine what the remaining characters will be in a word given a certain prefix (tries are also known as prefix trees). Compared to hash tables, tries make searching for elements faster and they do not have the potential for data collisions, even though they do use more memory.

String handling and processing are some of the most important topics in programming. Many real time applications are based on string processing like search engine results optimization, data analytics, and sentimental analysis in Natural Language Processing. Considering the time and space required for processing gazillions of data being typed on web pages from smart devices every second, efficiency is of critical importance to any company that provides such type of service. Its tree-like design ensures high efficiency in data retrieval for dynamic sets and associative arrays where keys are usually strings.

Tries also seem to be a low-floor high-ceiling data structure that could be refined to adapt to different applications, like a Ternary Search Tree for dictionary implementation, which optimizes a trie with a balancing tree feature to save space.

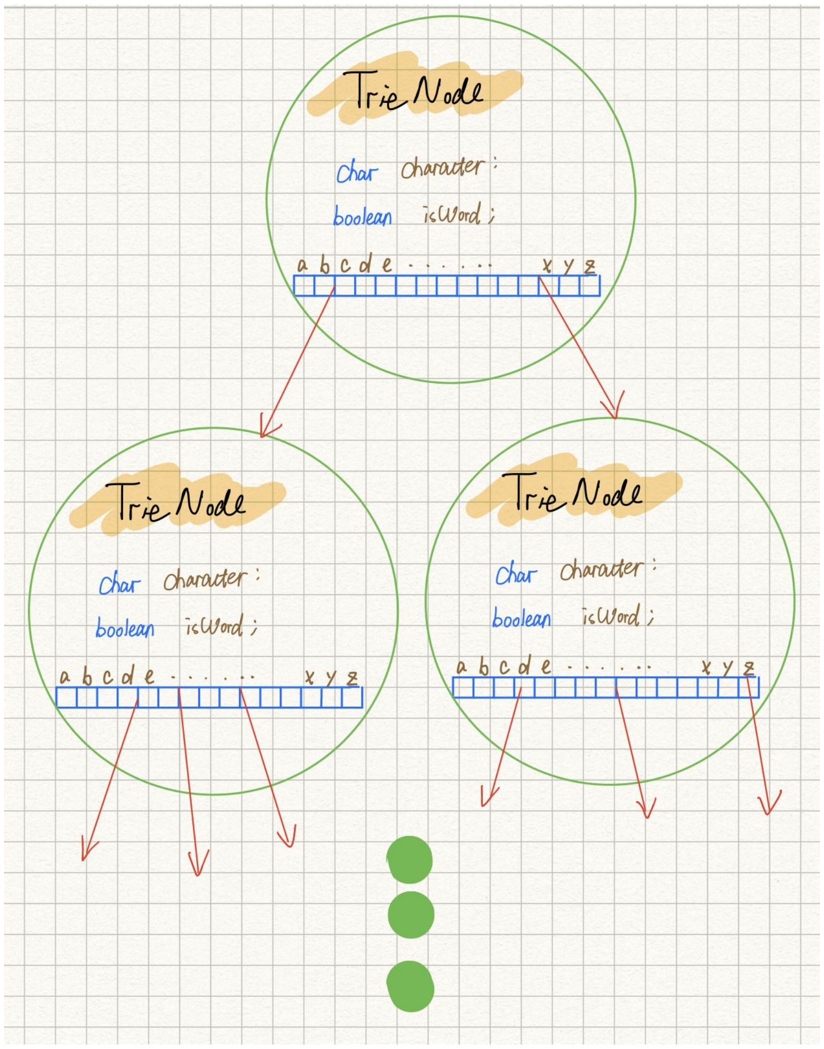
Tries have many applications in the real world, ranging from auto-complete and auto-correct to longest prefix matching and browser history retrieval. The applications in auto-complete are especially immense. It could predict the rest of a word a user types in whether on a web browser or in an email, provide suggestion lists in a Graphical User Interface or source code editing environment, auto-complete query information in database query tools, and as we're writing this document now, the word processor is using an auto-complete feature to reduce the repetition of the same words/phrases. We are likely also training the application for better word recommendation and correction.

4. Trie Implementation and Design Tradeoffs

Basic Structure

The building blocks of a trie are its nodes. Each node (except the root) contains a letter and an array of pointers. When a node is initialized, all of the pointers in the array are set to NULL. There is a pointer for every letter of the alphabet in each node's array.

If a second letter is added after an existing letter in the trie, the existing node's pointer array is populated at the proper index with the address to the new letter's node. Tries use the ASCII values associated with each letter to determine where to place the pointer in the array. The ASCII decimal value for 'a' is 97, so assuming we are creating an all-lower-case trie, we will subtract 97 from the new letter's ASCII decimal value to arrive at its proper index value. For example, any node containing the letter 'b' will be found in its parent node array at the following position: 98 - 97 = **1**.



Tries also need to indicate where the end of a word is. This can be done as an additional parameter within the node, or a node with a special character can be used to indicate to the algorithm that it has reached the end of a word. Either way, it is important to remember that one branch may have several words within it. Consider the word “joyful.” The trie should reflect that “joy” and “joyful” are both words without duplicating the nodes that make up “joy.”

Insert Functionality (root, word)
The insert function starts at the root of the trie and looks up the index value for the first letter in the word in the root’s array of pointers. Let’s say we are inserting the word “apple.” If the index for ‘a’ is not NULL, the algorithm recursively calls insert again using the existing node for ‘a’ as the root and “pple” as the word. Each time the insert function is called recursively, it removes the first letter of the word before it passes it into the function as a parameter. When the function is eventually called with an empty string as the word, the function sets the given root as the end of the word. If the function encounters an index for a letter that is NULL, it creates a new node for the letter and continues to call insert recursively as it would have if the node had previously existed.

For a word to be inserted, it will take O(m) time and O(m) space, where m is the length of the word, since each character will take O(1) time and O(1) space to be created.

Search Functionality (root, word)
The search function mimics the insert function in how it traverses the trie. It begins at the root and looks for the index of the first letter of the word. If the index is not NULL, it continues to call the search function recursively, dropping the first letter of the word each time to look up the proper letter at each node. The only difference is when the search function encounters a NULL index where there should be a pointer to an existing node. The search function will return false when this happens. If the search function is able to find each letter in the word, it will then check if the trie has stored the last letter as the end of the word. If so, the search function will return true; otherwise, it will return false.

It will take O(m) time to search for a word, where m is the length of a word, no matter how many words are present in the Trie, because the search function will do an equivalent of “early return” when a character is not found along the tree stem.

Delete Functionality (root, word)
The delete function, after using the search function to verify the given word is in the trie, traverses the trie to the end of the word. From there, it checks each node in reverse order. If the node has an empty array of pointers, meaning no letters follow it, the function frees the node. Otherwise, the function returns. Checking the array for non-null pointers makes sure we do not unintentionally delete letters that are in words other than the one we want to delete.

Since the delete function first searches if the word exists in the Trie and then checks if the last node contains an empty array of pointers, the big O time complexity is on the same order as Search functionality, which is O(m).

Print Functionality (root)
The print function uses an array of characters as a string to print each word. The function begins at the root of the trie and looks for the first non-NULL pointer in its array of pointers. It follows the pointer to the next node and adds the node’s letter to the string. If the letter is the end of a word, it prints the current string and a new line. The print function calls itself to continue traversing a word and adding its letters to the string until it comes to a node without any more non-NULL pointers. In this case, it sets the most recent addition to the string to NULL as it works its way back up the trie towards the root.

Since the print functionality prints out the whole trie, it takes O(M*N) time to accomplish it, where M is the average length of the words and N is the total number of words in a trie.

Nearest Full Word Functionality (root, prefix)
One of the key functions of a trie (also known as a prefix tree) is to find the nearest full word (or words) to a given prefix. This function begins at the root of the trie and looks for each letter in

the prefix as it traverses the trie. When it comes to a point where either the next letter in the prefix is not found in the trie or it has reached the end of the prefix, it will look for the next non-null pointer in the current node. For each subsequent node, it will check for the end-of-word indicator. Depending on the implementation, the function may print either the first full word found or all words found after the prefix.

The time to find the nearest full word is O(m), since it will immediately return the word once a trie node with the given prefix is marked as the end-of-word. But it could take O(M * L) to find all the words with certain prefix, where M is the average length of the words with such a prefix and L is the number of words with such a prefix.

Trade-offs

Memory usage and speed are considered two major performance metrics of a Trie structure.

Tries could be memory inefficient when the data size is small, since there will be a lot of unused overhead in a Trie node (every Trie node contains an alphabet-sized array of pointers to trie nodes). Compared to Tries, the HashMap is a better choice when the data size is relatively small. But as the data size grows, the Trie’s advantage in speed shadows the HashMap due to its extremely efficient searching performance. It will only take O(m) (where m is the length of the string) to search for a word, whatever the size of the data structure, avoiding the down speed issue when the data size is close to the load factor in a HashMap or any data collision issues. Besides, Tries are more memory-efficient when many words in the dataset share a small amount of prefixes. It doesn’t need to store repeated characters as a HashMap does.

An alternative way is to store a character and its subsequent Trie node pointers in a HashMap within a Trie node. This way, there won’t be unused word slots anymore and space is only allocated and used when a new character is inserted.

Deletion: Our implementation doesn’t really delete characters in the Trie and we accomplish “deleting” by marking it not as a word. Therefore, all the words that have been inserted in the Trie will remain in the memory even if they have been “deleted”. This makes sense from the perspective that the prefix of the word-to-be-deleted might be shared by other words. But this way of implementation means we don’t have any place to free memory, and the size could only go up. One way to solve this problem is by checking from the last character of the word-to-be -deleted if it is a prefix to another word, if it is not, we could safely delete it. But this will take at most word length of time to check. So if we would like a more memory-efficient Trie, deletion will be slower. Also, if the words we deleted turns out to contain a popular prefix, then we will not be able to reuse what has been inserted before.

5. **Research on potential overlap**

Module 8 defines an algorithm as: *a process or set of rules to be followed in calculations or other problem-solving operations, especially by a computer.* While we see algorithms applied in some of the functionalities of our trees, we also wanted to begin to explore the scientific method as an algorithm. One of the first steps in the scientific method is to pose a testable question. We are going to explore and analyze potential applications where the use of K-D tree and trie algorithms overlap. In other words, we will explore the question: how can we use the concept of nearest neighbor (from k-d tree) to inform a trie search approach, or vice versa?

Much of our research consisted of developing our own trees to familiarize ourselves with how these structures operate. One commonality between these two structures is that they are both trees, and trees are great for searching. The big difference between the two trees is the types of data they hold. While a trie holds words, and the search functionality is used to determine whether or not a word is contained in the list, a k-d tree holds data assigned to values with k being the number of types of values.

In this paper, our research on these trees reached the experiment design stage of the scientific process. Our hypothesis was that we would be able to combine these trees in a meaningful way. The following paragraphs explore how me might set up and design the combination, and smaller experiments could ultimately be derived from those designs. It is worth noting that our hypothesis and experiments are in the process of refinement and will be easier to carry out and analyze in an algorithmic way once the hypothesis and experimental steps are clarified.

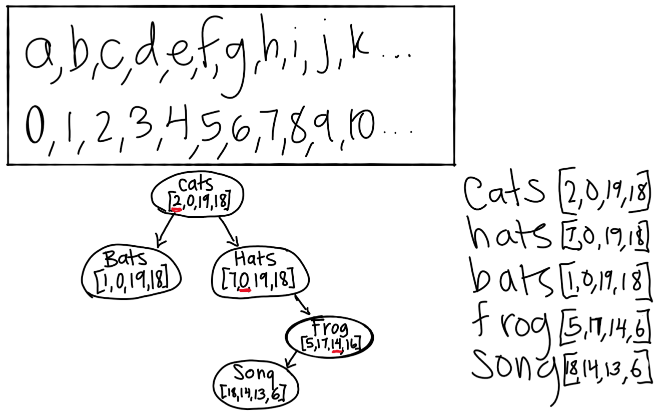
The nodes of a trie are flexible in that they can hold any data the designer wants to track. In the paper “Traversals, Tries, K-d Trees,” the author explains that a trie can track how many full words follow a certain node. Instead of tracking this information in a trie, though, we could track it in a k-d tree. Knowing the number of words that come from certain combinations of letters would indicate the frequency with which two characters follow each other. A keyboard designer may wish to place those characters near each other. Because a k-d tree can weigh more than one element at once, it could factor in frequency and available space when determining the optimal position of the keys.

Potential Application 1: There is the possibility for overlap of these two algorithms in designing an efficient keyboard. We would be able to combine the trie’s ability to store lists of words and identify common characters at the start of words with the k-d tree’s ability to find neighboring words with similar collections of characters. One of the key questions in this design is, what is the best way to map letters onto a k-d tree, given that it typically takes numbers. One approach is to have the indices be letters in a given word, and another approach would be to have the indices be letters of the alphabet:

In the first approach, the index is the index into the word and the number is the index in the alphabet. For example, our first index refers to the ‘b’ in ‘banana’ and the 2 refers to the fact that ‘b’ is the second letter in the alphabet.

"banana"
[2, 1, 14, 1, 14, 1]
(2: B, 1: A, 14: N)

Using the approach above, we can use the trie to store a list of words that we can then search for in the k-d tree. We can set k to be the index of the letter of the word we are on and search for all of the other words in the list that are nearby (k will range from 0 to the number of letters in the longest word). In other words, our k can represent a full word, such as 'banana', k=6, or a fragment of a word, such as 'bana', k=4. This may allow us to search a list of words to find letters that have the most common letters. This information could be used to inform a keyboard.



This image above represents element in k-d nodes as letters composing words or parts of words. Letters typed = k. The value assigned to the index can be 0-26, where 0 is no letter and 1-26 are letters of the alphabet.

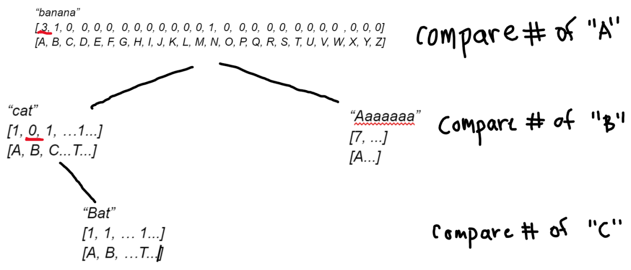
The trie data structure may not be the best way to determine the similarities between the words above because they all begin with different letters. This is a limitation of the trie data structure -- it only represents relationships between words that share a prefix. Even though "cats" and "hats" are one letter off from each other, they are on completely separate branches of the trie. If we were looking at "cats" and "cars," a trie would better represent the relationship between those words because they share the prefix "ca-." This is one of the benefits of using a k-d tree in this scenario.

One limitation with the k-d tree in this application is that when we are looking at the most common letters, we want to be able to see if two characters are the same, but don't really care about finding letters that are close in the alphabet. For example, if we have an r in our first index of a word, we only care if other words also have an r there, not if they have a q or an s. In a sense, we are looking for a binary value rather than a number value. This brings our entire use of a k-d tree into question here. Does it make sense to use a k-d tree if we are looking for two elements that are the same rather than two elements that are close together? This prompts finding another way to represent letters on a node in a k-d tree.

In this second approach, k=26. Our indices are letters in the alphabet. If the letter does not show up, we simply note a frequency of 0. Otherwise we mark the number of times the element shows up. For example, in ‘banana’, a shows up three times, and is at index 1 in the alphabet, so we mark index 1 with a 3.

“banana”
[3, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0 , 0, 0, 0] (frequency)
[A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P, Q, R, S, T, U, V, W, X, Y, Z] (spot is letter)

We can see how creating the nodes in the above fashion might lead to the following form of tree.



There are many possibilities for framing the tree above. We could think of each node as representing a word, where we track the most common letters in each word. We could think of each node as a particular language, take a writing sample from that language, and track the frequency of letters that occurs. Then, when we receive a new word or new language sample, we can find the nearest neighbors, or sample with the most similar frequencies of letters. This approach of mapping letters to numbers integrates the functionality of a k-d much better than the previous example, because in this case having a similar amount of letters is great information. For example, if we have 12 ‘E’s in one word and want a word with a similar number of ‘E’s, it is helpful to know words with 13 and 11 ‘E’s.

The application to creating a keyboard here is that if we can track the number of letters for given language samples in a node, we can create customizable keyboards that belong to these nodes and select one of these keyboards based on an incoming sample, and find the node that has the most similar qualities to it.

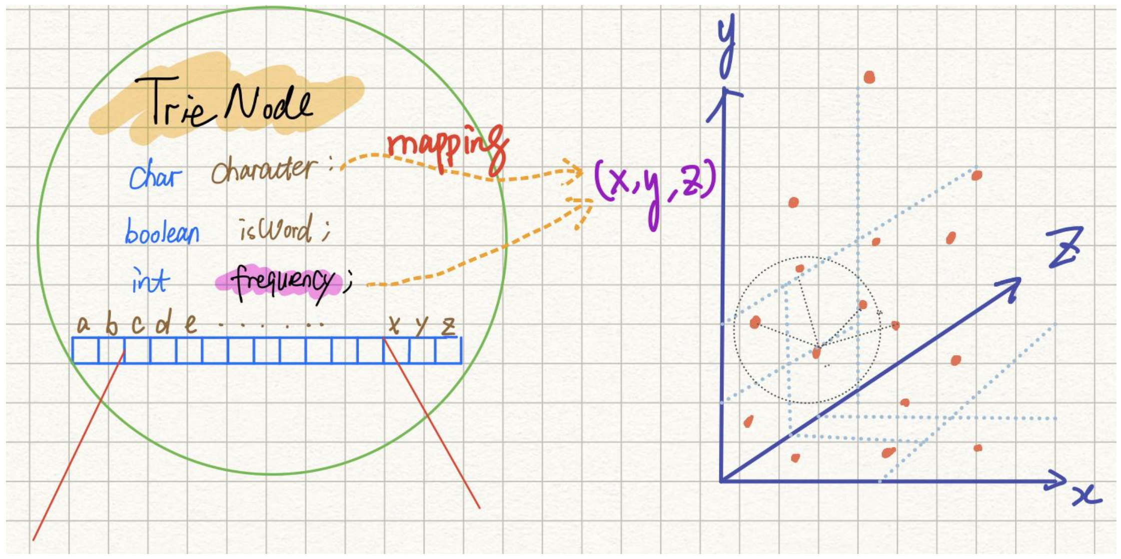
One area for exploration here is if it is possible to assign a specific trie to each node, and find a trie best suited (most similar) to a given text sample, using the nearest neighbor function.

<u>Potential application 2:</u> Using frequency to search for the nearest full words in a trie. Given that there could be several answers, we would need a standard to sieve the word out. That standard could be the frequency a word is used, the length of a word, or closest next full word. In our K-d tree, the mapping of words to coordinates could be determined by how frequently a word is used.

We could use a trie to hold all the words in a given context, for example, top 100 most cited papers on topics of Spatial Data Structure, and in each trie node, we keep a record of how many times a word has been used by using an integer frequency, and then we map the frequency of words to coordinates in a k-d tree. See the following picture for an illustration, where a 3-d tree is used as an example.

Once all the words are mapped to a 3-d tree, we could clearly see how words are clustered in terms of frequency and this might be a start for lexical and semantic analysis in machine translation and interpretation or automatic article generation.

Especially in the context of multi-language translation and interpreting, it is a challenge for AI-based translation programs to come up with words in target language with equivalent meaning. The words patterns shown in 3-d tree might be helpful for the programs to find the equivalence quickly in the target language more precisely and quickly.



6. Analysis and Conclusion

The specific implication of this overlap is that we could potentially design keyboards that would be more efficient given a particular lexicon or given a specific language. The more general-- and perhaps more potent-- implication is that we can do amazing and creative things when combining data structures. The intersection of two data structures is a particularly rich area to explore because it requires creative thinking and contextual thinking to discern a situation where the data structures could be used together.

There are several limitations to this exploration that are worth mentioning. The first limitation is that a customized keyboard is not necessarily more efficient because if someone is typing on a standard keyboard, it may take a great deal of effort and work to establish new typing patterns enough to make it worthwhile to develop a customizable keyboard. However, if customizable keyboards were made at a large enough scale that they were used regularly, such as developing customizable keyboards for different languages that have different frequencies of letters, it may make sense. There are also some pre-existing fields, such as court reporting, that have customizable keyboards to fit the environment, so these technologies do exist and make sense in come cases.

Areas for future research/learning:

Aside from continuing the scientific method in our experiment design, there are some other prominent areas we have identified for our own future research:

We explored a k-d tree for k = 2 and a little bit for k = 3. A further area to explore is how we would code the k-d tree if we took k in as an input, and customized the dimensions of the tree on the spot.

With more time, we would have liked to code an interactive trie program where a user could develop and search a trie from the command line. We started to convert our shell code from earlier in the semester to create this sort of program, but it was beyond the scope of this project.

We could make our trie implementations more memory-efficient by replacing the alphabet-sized pointer array with a hash map so that we only allocate memory when needed.

Preliminary Research Resources:
K-d tree:
Links:
https://www.youtube.com/watch?v=TLxWtXEbtFE
https://www.youtube.com/watch?v=XG4zpiJAkD4
https://scholar.google.ca/scholar?hl=en&as_sdt=0%2C5&q=kd+tree+XR&btnG=
https://dl.acm.org/doi/pdf/10.1145/1457515.1409079
https://en.wikipedia.org/wiki/Ray_tracing_(graphics)
https://en.wikipedia.org/wiki/Photon_mapping

Trie:
Links:
https://www.youtube.com/watch?v=zIjfhVPRZCg
https://www.youtube.com/watch?v=giiaIofn31A
https://www.geeksforgeeks.org/advantages-trie-data-structure/
https://www.geeksforgeeks.org/trie-insert-and-search/
https://iq.opengenus.org/applications-of-trie/
https://iq.opengenus.org/tries/
https://medium.com/basecs/trying-to-understand-tries-3ec6bede0014
https://www.cs.usfca.edu/~galles/visualization/Trie.html
https://named-data.net/wp-content/uploads/2019/02/19-ToN-TrieGranularity.pdf