**Project: String Database using the Radix Tree Data Structure**

Name: Ron Durham

Class and Semester: CSPB2270 , Spring

When researching which data structure I would want to do for the final project, I kept seeing that understanding the concept of Tries would be important in interviews in the future. After studying Tries I happened to come across the Radix Tree data structure and like how it flows together. It is great for gathering a collection of strings and placing them in an order that is quickly searchable while also minimizing the unnecessary roots where the rest of the word has no more extensions. Similarly with prefixes to the words, strings can be split up in a more uniform and concise manner.

Furthering my research, there looks to be a flaw in the Radix tree data structure that lead to the invention of the Adaptive Radix Tree data structure. This allows for the amount of nodes to grow looks to have many applications and practical uses, but may be a bit much for me to understand an implement at this time. So for this project, I will focus on the implementation of a Radix tree with names of local surf spots to throw a personal spin on something that has wide real world application. While, through the creation of the Radix tree, I may find ability to alter my progression and use an Adaptive Radix tree if I’m able to gather enough information about it.

I would be creating a RadixTree Class which I would suggest to have the following member functions and variables.

**Data Structs**

* Node struct
  + Contains the following:
    - Pointer to the character/string
    - Length of the string
    - Parent and child pointers
    - Possibly also a constructor to copy/delete the node

**Publics Member Functions for user to interact with:**

* shared\_ptr<nodestruct> InitNode(char x)
  + This function would establish a new node with reference to the character in question and the applicable structure elements
* int prefix(shared\_ptr<char> x, int n, shared\_ptr<char> key, int m)
  + This function will calculate the length of the biggest common prefix
* shared\_ptr<nodestruct> find(shared\_ptr<nodestruct> t, shared\_ptr<char> x, int n = 0)
  + This function will search for the x key in the t tree
* void split(shared\_ptr<nodestruct> t, int k);
  + This function will create space for the new node to be inserted
* shared\_ptr<nodestruct> insert(shared\_ptr<nodestruct> t, shared\_ptr<char> x, int n = 0)
  + This function will insert the new node while the string is not already in the tree
* void join(shared\_ptr<nodestruct> t);
  + This function will merge nodes after one has been removed
* shared\_ptr<nodestruct> remove(shared\_ptr<nodestruct> t, shared\_ptr<char> x, int n = 0)
  + This function removes the node in question then will call the join function to keep the tree compressed

**Private Member Functions Hidden From User:**

* shared\_ptr<nodestruct> root\_ptr()
  + This will establish the root pointer of the radix tree

In order to test this algorithm I’m planning to run it through an example, by building a radix tree and testing the cases of the public functions.

**Testing Criteria:**

* I will use tests to build a Radix tree containing strings of various categories (Pet names, surf spots, countries of the world, etc.)
* The tests will create radix trees that will use all of the functions and determine if elements are already in the tree, or could be removed from the tree, etc.

References:

1. Zybook chapter 7
2. <https://en.wikipedia.org/wiki/Radix_tree>
3. <https://blog.sqreen.com/demystifying-radix-trees/>
4. <https://arxiv.org/ftp/arxiv/papers/1911/1911.01763.pdf>
5. <https://db.in.tum.de/~leis/papers/ART.pdf> (This is a very interesting concept and I hope to be able to incorporate it in this project or one in the future)

I’m sure I’ll find more resources as the project progresses and I’ll be sure to include them in my final submission.