CS214
Authors:
Allen Tung
Anthony Wong

Project 3: Indexer

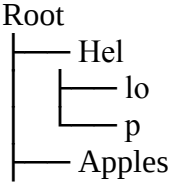
Design Decisions

First and foremost we must look at the data structure chosen to store indexes. There were two options considered a Hash Table or a Radix Tree.
For the hash table we considered making a table with 26 keys representing the first letter of each token allowing us to keep the indexes in sorted order. However, with this came major inefficiencies; with a large amount of tokens we would have far too many collisions to make it remotely useful. Secondly, we also considered making a hashing function that would attempt to keep things in pseudo order but, this was also far too time inefficient to code and memory inefficient.
So, we turned a Radix tree data structure which resolves all the problems above. Every token inserted would be in sorted order by nature and Nodes are only made when needed so memory is not wasted.

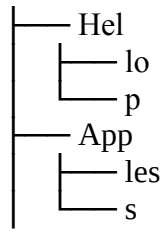
Specifics of the Radix Tree

Our Radix Tree implementation contains Nodes with a prefix string, child, and next and links the children nodes as a singly linked list.
Diagram of Our Radix Tree Implementation

if our tree contained the words Help, Hello, Apples



After inserting a new token Apps



When a Node also spells out a token then, a sorted list pointer will point to a bucket that contains a file path name and frequency of the token.

Overall Structure of Indexer

The Indexer utilizes a tokenizer, radix tree, and sorted list. First, it receives a path name from user and runs it through the function dirTrav which checks the validity of the path. If the path is a file it will pass the file to the tokenizer which will return tokenized strings. The indexer will then place these strings into the tree. If the path is a directory then dirTrav will enter the directory and traverse recursively. If the path is a symlink it will attempt to follow it and handle the errors as needed

After inputting all the tokens into the tree, it will write the indexes to an output file specified by the user. This is done by having the tree perform a preorder traverse and returning a sortedlist of the tokens and buckets.

Algorithmic Analysis

Inserting a token has a worst case O(m) where m is the length of the string. It asymptotic time complexity is of O(log(n)) where n total number of members. Thus to index n tokens requires O(nm) time in the worst case and O(nlog n) time in the average. Traversing through the tree to write an output file will require O(n) time. The space complexity is of order of O(vn) where v represents a v-way branch.