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 and 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. We further considered making a hashing function that would attempt to keep things in pseudo order, producing 26^n keys with each rehash, dependent on the ratio of entries to keys, but this was also far too time inefficient to code and memory inefficient.

So, we turned to 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. It would be much slower than the hash table in the printing to a file, but as the data structure access is only used once, the increase in creation speed is significant. Large files should be handled without an exponential increase in time.

Specifics of the Radix Tree

Our Radix Tree implementation contains Nodes with a prefix string, child Node pointer, and next Node pointer. The children nodes are implemented as a singly linked list.

Diagram of Our Radix Tree Implementation:

if our tree contained the words Help, Hello, Apples

```
Root
├── Hel
│   ├── lo
│   └── p
├── Apples
```

After inserting a new token Apps

```
Root
├── Hel
│   ├── lo
│   └── p
├── App
│   ├── les
│   └── s
│
```

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. First, it attempts to follow the path as a symlink, and operates on it if the flag is set. If it can't, then it tries to follow it as a directory, and recursively calls dirTrav on each entry in the directory. If both the symlink and directory attempts fail, then the path name is a file- all attempts in determining the binary or ASCII-ness of the file resulted in failure, so we will assume the file is a valid ASCII file. dirTrav then passes the tokens from the tokenizer to the indexer, which places the strings into the tree along with a file name and frequency.

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 traversal and returning a sorted list of the tokens and buckets.

Algorithmic Analysis

Inserting a token has a worst case O(m) where m is the length of the string. Its asymptotic time complexity is of O(log(n)) where n total number of members. Thus indexing n tokens requires n*(O(m)) time in the worst case and n*O(log n) time. Traversing through the tree to write an output file will require O(n) time.

Memory Leaks

They exist. Sorry. Valgrind reports 2kB memory leakage testing the indexer on the testb directory. There were several malloced character arrays in the preorder traversal function that could not be freed, because they produced a bus error if they were to be freed. For the sake of functionality, some memory had to be sacrificed.