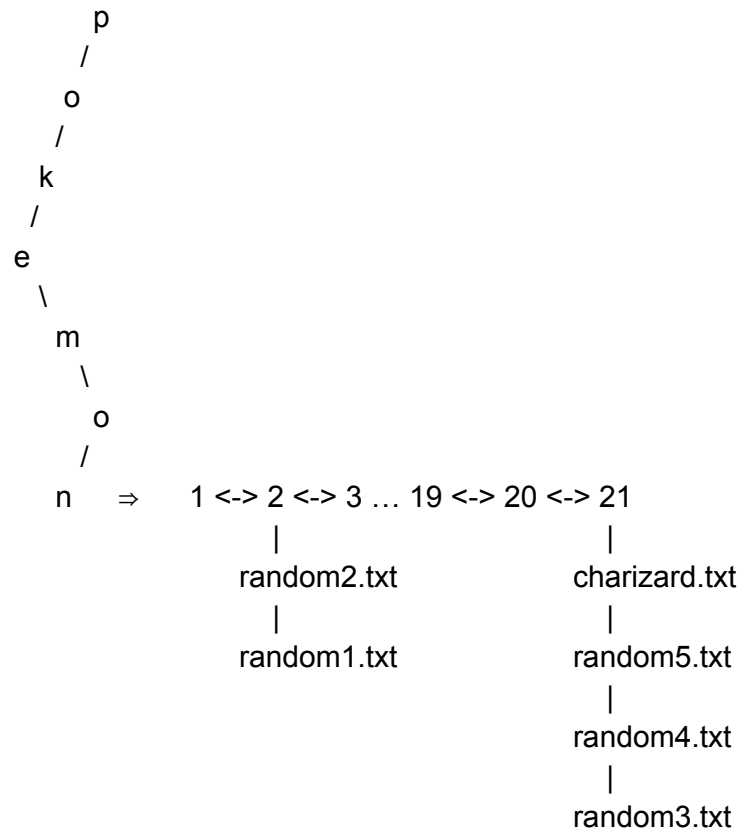
Indexer

Algorithm Design

This program attached takes in a directory and generate a new file all the alphanumeric strings in all “regular” files in the given directory or any sub-directories it contains. Along with printing out the strings in alphabetical order, the files in which strings appeared and their frequencies are printed as well. There are two main data structures at work in this program, the first being a prefix tree, which allows for easy storage of strings in alphabetical order. A prefix tree is essentially a “36-ary” tree, where every node has an array or next pointers, where index 0 points to “a” 1 to “b” 2 to “c” etc. After word is entered the next word is added from the root, so any overlapping prefixes share node, allowing large amounts of words to be stored without taking up to much memory. A feature of this structure is that by doing a simple recursive inorder traversal all words in the tree are accessed alphabetically.

The second data structure used is are two doubly linked lists, which store file names and frequency. The first double linked list acts as the “frequency list”. Each node in this list has a frequency number attached to each one, as well as a second double linked list. This second double linked list contains the filenames pertaining to that frequency. So for instance, if a filename “charizard.txt” contains 21 instances of the word “pokemon”, the double-linked-list that is attached to the “pokemon” path of the prefix tree would contain a node called “charizard.txt” at the 21st frequency node. The structure containing the DLL has a “recent” ptr for instant access.



Run-Time and Space Complexity

An interesting feature of a prefix tree is its fast runtime for insertion and lookup. Because all node have random access to the following nodes via an array, and if a prefix already exists new node do not need to be created, the run-time for a prefix tree is O(k) for these two methods. The only draw back for prefix trees however is that they can potentially take up a lot of space, because every letter present in every file takes up its own node. However because each prefix is "recycled" as a file become bigger the prefix tree actually becomes increasingly efficient. At most, the prefix tree takes up 2*k space, where k is the total character count.

The double linked list inserts in O(1) time for every word, therefore resulting in a time complexity of O(n) for n words. Since the first double linked list has a worst case scenario of taking up as much space as the number of words in the entire file structure which is being inserted, or trading that space to store a new distinct word or file, the space complexity is O(n), where n is the total amount of words in the file structure. However, this can be optimized by removing empty frequency nodes (nodes that do not have a filename DLL attached to them), therefore still being O(n), however only depending the the number of distinct words and files.

Real Time Testing

With "./index testresults.txt ./testdir/", on a 1200 MHz i7 machine, the time it took to completely run all processes was 2.300 seconds. The amount of memory it allocated was 78,724,783 bytes before freeing. Note that ./testdir did contain an instance of big.txt.