Ellis Saupe ems236
Progress Report

1.
So far I have built and tested working add, isWord, delete, and autoComplete methods for each data structure meant to store Strings.

The array stores whatever String is added at the end of the array. If it reaches its capacity, it doubles the size of the array. Searching linearly searches the array. The delete method searches for the word, removes it from the array, and shifts the rest of the array to the left. The autoComplete method searches the array for substrings that match the parameter and returns the first match.

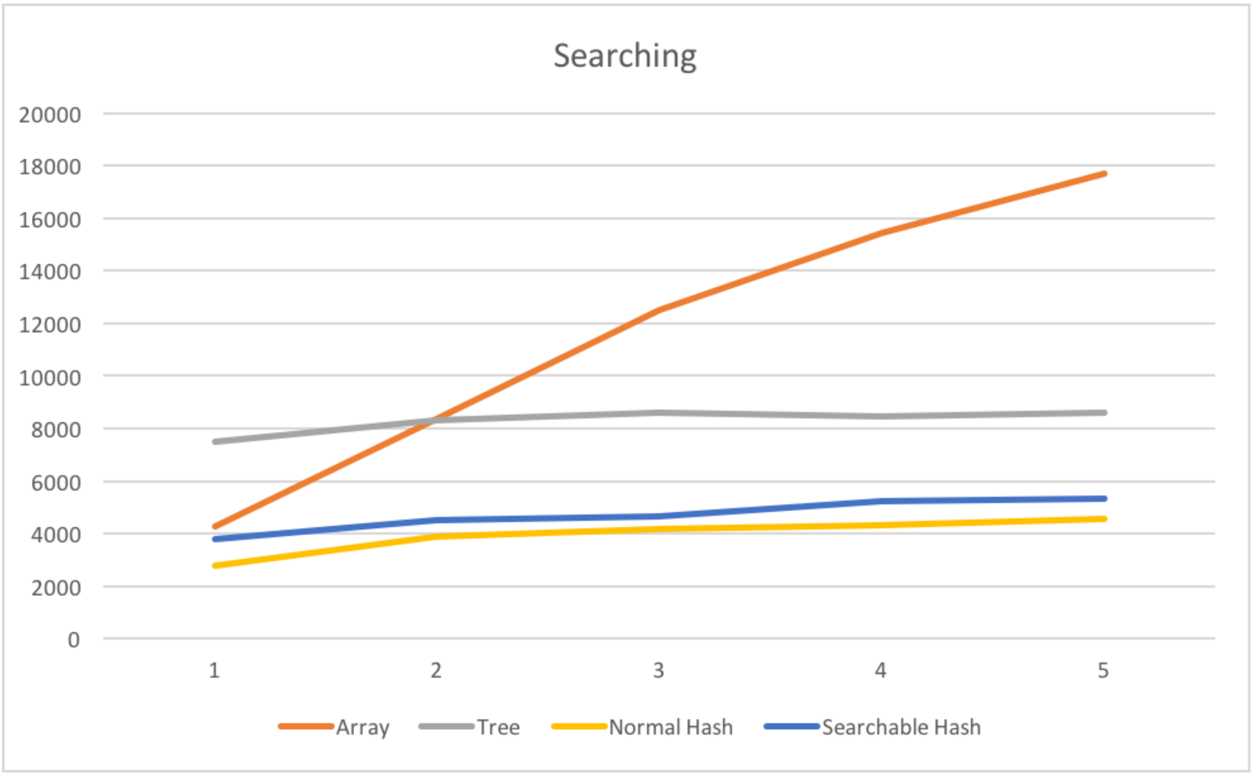
The tree works as follows: The root of the tree contains no data. Each node of the tree contains an array of 26 references to its child nodes, and a boolean value signifying if this position on the tree is a valid word. To traverse the tree starting from the root, each character of the string corresponds to the index of the child – the String "ab" is found by starting at the root, setting the cursor to the a=0th child of the root, then the b=1st child of that node, and so on. The add word method traverses the tree using whatever word is being adding, initializes each node that is part of the transversal while setting the isWord boolean to false if necessary, and either initializes the node associated with the complete word if necessary and sets isWord to true. Every substring of a word must be initialized in the tree to access its children and store the word, therefore, a null value at some position in the tree signifies that no valid words contain the String associated with that position. The isWord method searches the tree for a word by traversing traverse it until either the node associated with the word or a null value is found returns either the isWord boolean of the node or false respectively. To delete method, search for the node in the tree, and sets isWord to false if it is found. The autocomplete method traverses every branch with a non-zero child count until it finds a valid word. It does not check every child of every node, it moves the cursor every time a child has a non-zero child count.

There are 370099 words in the English dictionary I'm using. Each hash table currently has a max size of 2000003, and each hash table double hashes with 2000001. I'll probably make that dynamic to accommodate bigger datasets that I may use depending on how I decide to analyze the add method of the hash tables. These tables each store an object in the table. The String only hash table doesn't really need its own object, but it's more flexible if I decide to add more functionality. The add method of first hash table hashes each word and stores it in the hash table. The isWord method uses double hashing to search for the String and returns false if there is either a null, or the data attribute of the object is null (the word has been deleted). The delete method uses double hashing to search for the word in the hash table and sets the data attribute of the object to null if it finds it. Because the data is hashed and hashes cannot be inversed, it would be impossible to use hashing for an autocomplete method without storing additional data in each hash object, which the other hash table does. The autocomplete method of this hash table ignores hashing and parses the entire array searching for matching substrings in the same way as the array data structure. I tend to call this structure the normal hash, and the slightly more complex hash table the searchable hash because it's easier to search for related words.

The second hash table also stores an isWord boolean, and the number of children in each object that is hashed. The children integer refers to children in the same way is in the tree even though these objects do not actually have children, children is the count of valid words that contain this substring. Similar to adding to the tree data structure, the add method of this hash table not only adds the word to the table with isWord set to true, but also stores every substring of every word in the table. Adding uses hashing to search for every substring, if it finds is found, the child count of the object is increased. Otherwise the hashDataObject is initialized, isWord is set to false, and the count is set to one. Deleting from this hash table requires checking hashing to search for the word twice. The first just searches for the word, the second search also finds every substring of the word and decreases the child count by one, and then sets the isWord property of the word to be deleted to false. The autocomplete method of this hash table searches for the String parameter in the hash table, and while the children count is greater than 0, it proceeds in the same manner as the tree. It iterates through the alphabet adding one letter to the word. If the new String is found in the hash table and is a word the new String is returned. If the new String is not a word but it still has children, the process is repeated adding another character. The hash table ends with around 880000 values. It's not near full, but the hash table could benefit from its maximum size being increased.

2.
I have analyzed the run time of the isWord method for each data structure. I wrote a method to build each data structure with every 5th word, every 2/5 words, and so on. This provides linearly spaced N values to use in runtime analysis. Using these partial dictionaries, I repeatedly called isWord for every word in this randomly generated list. String[] testWords =
{ "decade" , "cooperate", "magnitude", "contract", "switch", "throat", "sausage", "climate", "berry", "program", "cotton", "achievement", "double", "encourage", "reform", "ministry", "talkative" , "unfortunate", "transition", "iemxyngygu", "pcsxvqgwvh", "wfhsrvfwlg", "rxsuzockdk", "qdsyy", "gvabn", "njzzt", "yskoj" };
I used the randomly generated list to get a slightly better idea of the average case for each search. Using a variety of words including words that aren't words eliminates differences in run time from something like the word being the first one in the array. The results are shown below.

```
Dict 0 N: 1 SearchTime: 4262
Dict 1 N: 1 SearchTime: 7486
Dict 2 N: 1 SearchTime: 2776
Dict 3 N: 1 SearchTime: 3807
Dict 0 N: 2 SearchTime: 7928
Dict 1 N: 2 SearchTime: 8335
Dict 2 N: 2 SearchTime: 3883
Dict 3 N: 2 SearchTime: 4530
Dict 0 N: 3 SearchTime: 12497
Dict 1 N: 3 SearchTime: 8618
Dict 2 N: 3 SearchTime: 4158
Dict 3 N: 3 SearchTime: 4644
Dict 0 N: 4 SearchTime: 15460
Dict 1 N: 4 SearchTime: 8475
Dict 2 N: 4 SearchTime: 4309
Dict 3 N: 4 SearchTime: 5239
Dict 0 N: 5 SearchTime: 17704
Dict 1 N: 5 SearchTime: 8616
Dict 2 N: 5 SearchTime: 4579
Dict 3 N: 5 SearchTime: 5340
```

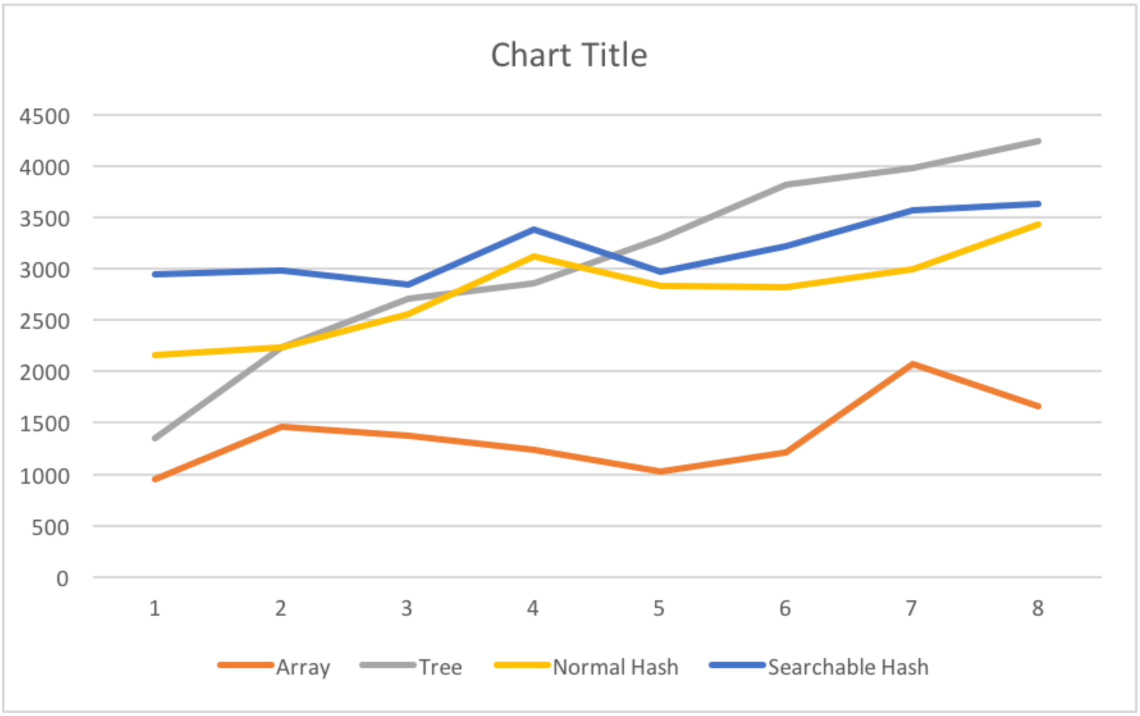


IsWord() appears to be O(N) for the array where N is the size of the data structure, and either O(1) or logarithmic for the hash tables and tree. The tree especially should be dependent on the length of the word being searched.

To test this, I made a list of words of varying character lengths and recorded data for words of w characters. I only used one word and one word that wouldn't be found for each length, so this test is much more subject to some words being found faster than the average value. However it should still show a trend in dependence on word length if one exists.

The data is below.

```
Dict 0 N: 1 SearchTime: 945
Dict 1 N: 1 SearchTime: 1349
Dict 2 N: 1 SearchTime: 2158
Dict 3 N: 1 SearchTime: 2946
Dict 0 N: 2 SearchTime: 1463
Dict 1 N: 2 SearchTime: 2239
Dict 2 N: 2 SearchTime: 2379
Dict 3 N: 2 SearchTime: 2978
Dict 0 N: 3 SearchTime: 1373
Dict 1 N: 3 SearchTime: 2714
Dict 2 N: 3 SearchTime: 2561
Dict 3 N: 3 SearchTime: 2850
Dict 0 N: 4 SearchTime: 1243
Dict 1 N: 4 SearchTime: 2856
Dict 2 N: 4 SearchTime: 3119
Dict 3 N: 4 SearchTime: 3382
Dict 0 N: 5 SearchTime: 1023
Dict 1 N: 5 SearchTime: 3288
Dict 2 N: 5 SearchTime: 2835
Dict 3 N: 5 SearchTime: 2965
Dict 0 N: 6 SearchTime: 1216
Dict 1 N: 6 SearchTime: 3816
Dict 2 N: 6 SearchTime: 2816
Dict 3 N: 6 SearchTime: 3218
Dict 0 N: 7 SearchTime: 2068
Dict 1 N: 7 SearchTime: 3977
Dict 2 N: 7 SearchTime: 2996
Dict 3 N: 7 SearchTime: 3573
Dict 0 N: 8 SearchTime: 1659
Dict 1 N: 8 SearchTime: 4244
Dict 2 N: 8 SearchTime: 3434
Dict 3 N: 8 SearchTime: 3625
```



This shows a reasonably linear dependence on word length for the tree, possibly some correlation between word length and searching the hash tables, and no reasonable relationship between word length and run time of searching the array.

3.
program files:
Dict.txt: A text file containing an English dictionary with one word per line. It's mostly alphabetized but not entirely.
Dict.java: an interface for these data structures.
DictParser.java: the file with my main method. I do all of the run time analysis here.
CharTreeNode.java: the class for the CharTreeNode object to be used in the tree.
CharTree.java: Implements Dict, the tree data structure and associated methods.
DictList.java: Implements Dict, the array data structure and associated methods.
StringHashObject.java: The object being stored in the simpler/normal hash table.
StringHash.java: Implements Dict, the normal hash table and associated methods.
SearchHashObject.java: The object being stored in the more searchable hash table.
SearchHash.java: Implements Dict, the more searchable hash table and associated methods.