Sorting Strings

1. Briefly describe least significant digit radix sort in your own words.

Solution: Least significant digit radix sort works by considering only a single digit (i.e. position) within a set of strings. It starts from the ends of the individual strings (i.e. least significant digit) and applies counting sort to each "column" of characters. When the first "column" is sorted, it moves on to do the same in the next "column". When all "columns" have been sorted, the full set of data is sorted.

2. Briefly describe maximum significant digit radix sort in your own words.

Solution: Maximum significant digit radix sort works by considering only a single digit (i.e. position) within a set of strings. It starts from the beginning of individual strings (i.e. maximum significant digit) and applies counting sort to each "column" of characters. However, unlike least significant digit radix sort, it doesn't immediately move on to the next "column". Instead, after sorting a single "column", it segments the strings based on equal characters. In other words, all words beginning with character 'a' are placed into a set, all words beginning with character 'b' are placed into another set, and so on. Then, maximum significant digit radix sort recursively calls itself on each individual set to complete the sort.

3. Suppose we need to sort a large set of short, fixed-length strings. Would maximum significant digit radix sort or least significant digit radix sort be a better choice to sort such a set of strings? Why?

Solution: Least significant digit radix sort would be a better choice to sort a large set of short, fixed-length strings. The key in selecting least significant digit radix sort is that we are using fixed-length strings. With fixed-length strings, least significant digit radix sort will have to do the same amount of work for each string. Furthermore, the cost of recursion for small strings in maximum significant digit radix sort is quite high. In this problem, the strings are short, and we want to avoid paying the high cost of recursion. Least significant digit radix sort doesn't require recursion.

4. A colleague suggests to use least significant digit radix sort to sort a set of strings. However, they suggest making a slight change: once there are only 3 letters left, cut off to insertion sort for the remaining sort. Why is this a bad idea?

Solution: Insertion sort performs best if the dataset is partially sorted. However, least significant digit radix sort sorts from the rightmost column to the leftmost column. Doing so leaves values in relatively unsorted order until the full least significant digit radix sort is complete. If we were to cut off to insertion sort at any point along the way, it would be equivalent to not having done the least significant digit radix sort at all and redoing all the work again. Furthermore, cutting off to insertion sort is useful in other algorithms because the cost of recursion and function calls is too large for small arrays. In the case of least significant digit radix sort, we aren't doing any recursion so it doesn't make any sense to cut off to insertion sort.

5. Describe how you would apply the ideas found in maximum significant digit radix sort and least significant digit radix sort to sort a set of integers instead.

Solution: We can apply the same ideas we use in sorting strings with maximum or least significant digit radix sort. In strings, we can access characters individually given their location. With numbers, we cannot do that. However, instead, we can use some simple mathematics to get access to the digit in a given place. For example, for the number 4,321, we can use integer division ($\frac{4321}{1000} = 1$ to arrive at the value in the thousandths place. In this way, maximum significant digit radix sort on integers turns into the same algorithm as on strings.

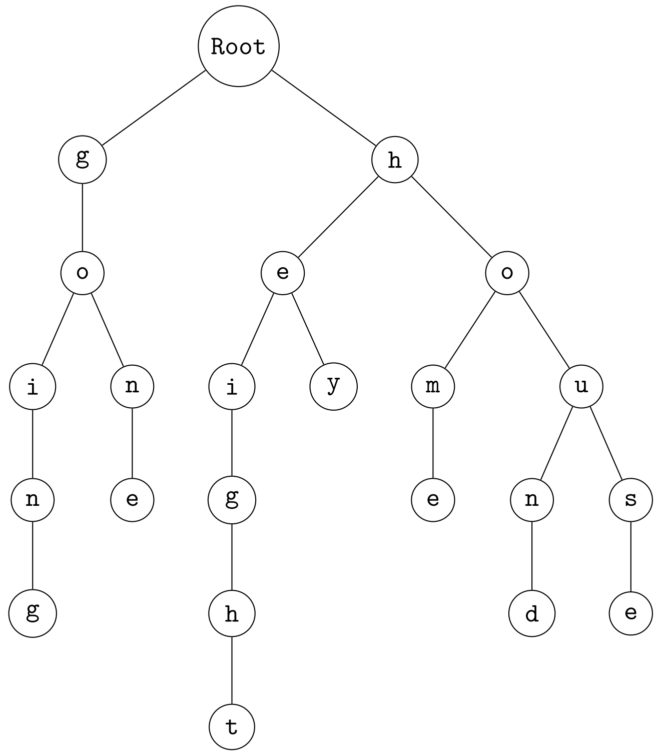
Tries

6. Describe an R-way trie in your own words.

Solution: An R-way trie is similar to a binary tree, except that each node has R children. Values are not stored in the nodes of an R-way trie. Instead, the position of the node, and the path that it is a part of through the trie, indicates the value that a node holds.

7. Insert the following strings into an R-way trie.

- go
- going
- gone
- height
- hey
- home
- hound
- house



Solution:

8. Why are tries usually faster at searching for strings than hash tables are?

Solution: In a hash table, you must hash the key to obtain its hash value. Hashing the key requires examing every part of the key (e.g. character in a string, digit in a number, etc). The complexity of hashing, then, is $O(L)$, where L is the length of the key. In a trie, a search hit takes the same amount of time. However, in the case where the trie doesn't contain the key, we can break out of the search much faster than $O(L)$. For cases where there are a large amount of search misses, a trie will be much faster than a hash table. Moreover, a trie doesn't have to resize like a hash table does. Therefore, the lookup time in a trie is guaranteed to be $O(L)$ instead of the amortized $O(L)$ in a hash table. Furthermore, tries don't have to handle collisions and can share paths for keys that have common prefixes.

9. In class, we discussed associating a value to a particular node in a trie. This effectively creates a mapping from a key to a value. However, suppose we wanted to only store information about set membership (i.e. the word is in the trie or not). How would you change the trie data structure to be able to store this information? Note: be careful to support words that are substrings of each other (e.g. going and go).

Solution: Instead of storing a value on each node, simply use a boolean `true` or `false` as a terminal marker. A boolean `true` means that the node is the last character of some string that was inserted into the trie. This effectively supports the substring case as well.

10. One of the drawbacks of an R-way search trie is that each node stores links for every letter in the alphabet, whether they are used or not. Describe a method to modify an R-way search trie to achieve space savings at each node without introducing any complex insertion and traversal.

Solution: The main drawback of a standard R-way search trie is that it allocates an array of R values for each node, whether they are used or not. If we used another data structure like a hashmap or a treemap instead of an array, we could achieve similar space savings without adding complex insertion and traversal.

11. Describe how you would apply the ideas found within trie search to search for an integer in a large set of them.

Solution: We can apply the same ideas we use in sorting strings with maximum or least significant digit radix sort. In strings, we can access characters individually given their location. With numbers, we cannot do that. However, instead, we can use some simple mathematics to get access to the digit in a given place. For example, for the number 4,321, we can use integer division ($\frac{4321}{1000} = 1$) to arrive at the value in the thousandths place. In this way, we can insert integers into a trie data structure almost as if they are strings.

12. A set of words have been inserted into an R-way trie. Given a string prefix, how can you find all words in the trie that begin with the given prefix?

Solution: Traverse through the trie until the we find the end of the prefix. If, along the way, we find a `null` link, the prefix is not a part of the trie. The node signifying the end of the prefix is the root of the subtree of all words in the trie that begin with the given prefix. Use depth first search to iterate through all nodes within the subtree. At the same time, keep track of the word being built as iteration progresses.

13. Suppose you are given two strings, a and b. Design a method to quickly determine whether a is a

substring of b or b is a substring of a.

Solution: Create two suffix tries, a' and b' where a' is the suffix trie of all suffixes that make up string a and b' is the suffix trie of all suffixes that make up string b. Then, attempt to find a in b'. If found, a is a substring of b. If not found, attempt to find b in a'. If found, b is a substring of a. If not found, neither a is a substring of b nor b is a substring of a.

14. Least significant digit radix sort and maximum significant digit radix sort are two ways to break the $O(n \log_2 n)$ lower bound for sorting strings or integers. Apply the ideas found in these algorithms to R-way tries. In other words, describe a method to sort integers or strings using an R-way trie.

Solution: Insert all integers or strings into the R-way trie. Pre-order traversal through the trie while keeping track of the integer or word formed will result in a sorted ordering of the original set of integers or strings.