Section 5: Hashing & Sorting

0. Sorting Hat

Suppose we sort an array of numbers, but it turns out every element of the array is the same, e.g.,{17, 17, 17, ..., 17}. (So, in hindsight, the sorting is useless.)

a) What is the asymptotic running time of **insertion** sort in this case?

O(n) - This is the best case runtime of insertion sort as it only requires one pass through the data. Insertion sort will traverse the array but since each element is not less than the one before it, no extra computations are necessary.

b) What is the asymptotic running time of **selection** sort in this case?

O(n²) - Selection sort always has *n²* runtime, regardless of the nature of data

c) What is the asymptotic running time of **merge** sort in this case?

O(n∗log(n)) - Merge sort always has *n∗log(n)* runtime, regardless of the nature of data

d) What is the asymptotic running time of **quick** sort in this case?

O(n²) - This is the worst case runtime of quick sort. When partitioning, every element is going to fall to the same side of the pivot since they all have the same value which essentially only sorts 1 element per iteration of quicksort, leading to the *n²* runtime.

1. Another Sort of Sorting…

Given an array of integers as such: {11, 13, 55, 67, 79, 10, 8, 6, 4, 2}. Please answer the following questions (assume all sorts to be done in ascending order):

a. What is the asymptotic running time of insertion sort for this array?
   This takes the worst case runtime of O(n^2) because the array is not already in sorted order.

b. What is the asymptotic running time of **selection** sort in this case?

O(n²) - Selection sort always has *n²* runtime, regardless of the nature of data

c. What is the asymptotic running time of **merge** sort in this case?

O(n∗log(n)) - Merge sort always has *n∗log(n)* runtime, regardless of the nature of data

d. What is the asymptotic running time of **quick** sort in this case (assuming that we choose the leftmost element as the pivot each time)?

$O(n^2)$

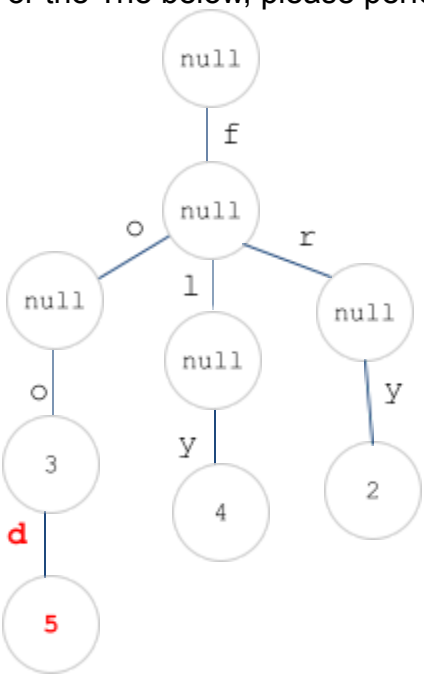
Case 1: upon choosing a good pivot (one that partitions the array into roughly equal halves), one of the subarrays would be completely in reverse sorted order, so it's O(n) to sort one of the subarrays at each level, resulting in a $O(n^2)$ asymptotic running time.

Case 2: choosing a bad pivot (the smallest or largest element) each time would result in the worst case runtime of $O(n^2)$ as well.

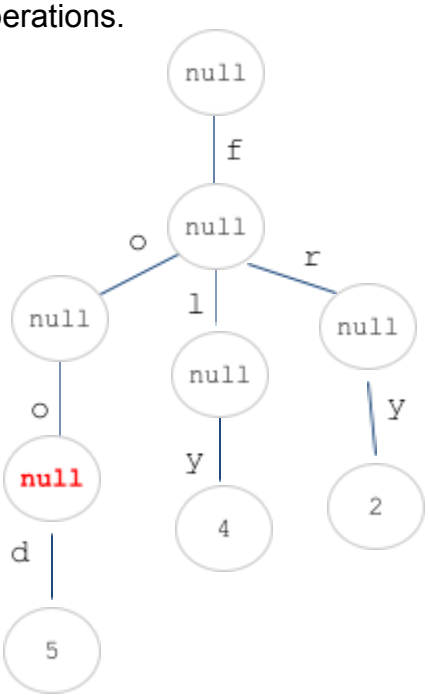
Tries

1. Let's give it a Trie!

For the Trie below, please perform the following operations.



(part a)



(part b)

(for parts a and b, modify the trie above)

a. insert("food", 5)

a. delete("foo")

b. what does the call to find("fry") return?

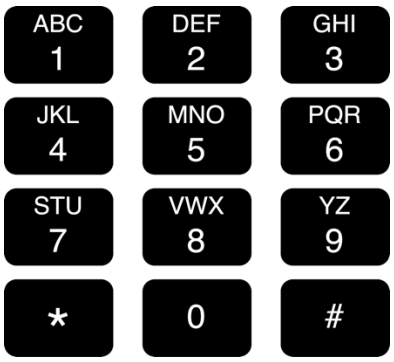
returns 2

c. List all key-value pairs in the final trie.

(“food”, 5) (“fly”, 4) (“fry”, 2)

2. Let’s Trie to be Old School!

Text on nine keys (T9)’s objective is to make it easier to type text messages with 9 keys. It allows words to be entered by a single keypress for each letter in which several letters are associated with each key. It combines the groups of letters on each phone key with a fast-access dictionary of words. It looks up in the dictionary all words corresponding to the sequence of keypresses and orders them by frequency of use.
So for example, the input ‘1554’ could be the words *book*, *cook*, or *cool*. Describe how you would implement a T9 dictionary for a mobile phone.



T9 example

Solution: There are multiple solutions to this problem. One such solution is to use a Trie, where typed digits represent the path to the corresponding node, and nodes store a list of all words ordered by frequency corresponding to the typed digits. ● To populate the Trie, iterate through each word in the dictionary and convert it into the appropriate sequence of digits. Traverse through the trie and add the word to the corresponding node’s list. Then sort the list to maintain the ordering by frequency.