CSE 332: Data Structures and Parallelism

Section 1: `WorkLists` Solutions

`WorkList` ADT

`add(`**work**`)`	Notifies the worklist that it must handle **work**
`peek()`	Returns the next item to work on
`next()`	Removes and returns the next item to work on
`hasWork()`	Returns true if there's any work left and false otherwise

0. $\mathcal{O}$dd Jobs

For each of the following scenarios, choose
(1) an ADT: `Stack` or `Queue`, and
(2) a data structure: `Array` or `LinkedList with front` or `LinkedList with front and back`
Then, explain why your choice works better than the other options.

`WorkList` Situations

(a) You're designing a tool that checks code to verify that all opening brackets, braces, parentheses, ... have closing counterparts.

Solution:

We'd use the `Stack` ADT, because we want to match the *most recent* bracket we've seen first.

Since `Stack`s push and pop on the same end, there is no reason to use an implementation with two pointers. (We don't need access to the "back" ever.)

Asymptotically, there is no difference between the `LinkedList` with a `front` pointer and the `Array` implementation, but *cache locality* will likely be a problem with the `LinkedList`. (Remember, arrays are contiguous in memory, but linked lists are stored using arbitrary pointers.)

(b) Disneyland has hired you to find a way to improve the processing efficiency of their long lines at attractions. There is no way to forecast how long the lines will be.

Solution:

We'd use the `Queue` ADT here, because we're dealing with... a line.

The important thing to note here is that if we try to use the implementation of a `LinkedList` with *only a `front` pointer*, either *add* or *next* will be very slow. That is clearly not a good choice.

Arguably, the `LinkedList` implementation with both pointers is better than the array implementation because we will never have to resize it.

(c) A sandwich shop wants to serve customers in the order that they arrived, but also wants to look ahead to know what people have ordered (ej. 2nd person, 3rd person, ..., last person in line).

Solution:

This is still clearly the `Queue` ADT. However, if we used a `LinkedList` like in part b, if we were to find a customer far from the front of the line, we would have to traverse almost the entire list.

An `Array` implementation would be more efficient to find specific people in line using array indexing, which is constant time.

1. `Trie` to Delete 0's and 1's?

Suppose we inserted all possible binary strings of length 0-3 (ej. 1, 0, 10, ..., 110, 111) into a `Trie`.

(a) If we deleted all binary numbers of length 2, how many nodes would we have to delete?

Solution:

Zero nodes! We still need the nodes storing the value for binary strings of length 2 because they have pointers to the nodes for binary strings of length 3, which still exist in the `Trie`. Therefore, we only set the value to null in the node to remove the key-value pair from the `Trie`.

(b) After part a, if we deleted all binary numbers of length 3, how many nodes would we have to delete?

Solution:

12 nodes. Since the binary strings of length 3 are all leaf nodes, they do not have any pointers to other relevant nodes in the `Trie`, so we can delete them, which is 8 nodes. However, in part a, the nodes that use to store the binary strings of length 2 are now empty, they do not store any values, so we can delete those as well, which is 4 nodes for a total of 12 nodes deleted.

2. Call Me Maybe

(a) Suppose you want to transfer someone's phone book to a data structure so that you can call all the phone numbers with a particular area code efficiently. What data structure would you use? How would you implement it?

Solution:

One way to solve this would be using a `HashMap` where the keys are the area codes and the values is a list of corresponding phone numbers. We will need to parse the phone number to get the first three numbers.

Another way to solve this is by using a `Trie`. We would use the entire phone number as the "route" and insert all numbers into the trie. Then, to find all the phone numbers to call, we would use the area code to partially travel down the `Trie`, then visit all children nodes to find up the phone numbers to print.

(b) What is the time complexity of your solution? What is the space complexity?

Solution:

If we compare the HashMap and TrieMap approaches, both will have the same runtime efficiency, but the `Trie` will be more space-efficient in the average case.

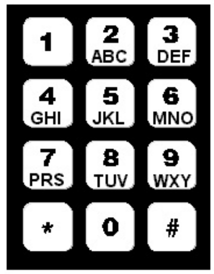
If we let n be the total number of phone numbers and e be the expected number of phone numbers per area code, we can find that it takes $\Theta(n)$ time to build either the `HashMap` or the `Trie`. Likewise, given some area code, it takes $\Theta(e)$ time to visit and call each phone number.

(Initially, it may seem like the `Trie` would be slower due to the traversals. However, recall that the depth of the trie is always equal to the length of a phone number, which is a constant value.)

The reason why the `Trie` turns out to be more space-efficient on average is because the `Trie` is capable of storing near-duplicate phone numbers in less space then the `HashMap`. If we have the phone numbers `123-456-7890`, `123-456-7891`, and `123-456-7892`, the map must store each number individually whereas the `Trie` is able to combine them together and only branch for the very last number.

3. 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 '2665' could be the words {`book, cook, cool`}. Describe how you would implement a T9 dictionary for a mobile phone.



T9 Example

Solution:

One way to implement this would be by using a `Trie`. The routes (branches) are represented by the digits and the node's values are collection of words. So if you typed in 2, 6, 6, 5, you would choose the child representing 2, then 6, then 6, then 5, traveling four layers deep into the `Trie`.

Then, that child node's value would contain a collection of all dictionary words corresponding to this particular sequence of numbers.

To populate the `Trie`, you would iterate through each word in the dictionary, and first convert the word into the appropriate sequence of numbers.

Then, you would use that sequence as the key or "route" to traverse the `Trie` and add the word.