CSE 332: Data Structures and Parallelism

Section 1: `WorkLists`

0. Odd Jobs

For each of the following scenarios, choose the

(1) ***ADT*:** `Stack` or `Queue`

(2) **and underlying *data structure*:** `Array`, `LinkedList` with front, or `LinkedList` with front and back*

then (3) **give a reason for each decision** (think about runtime, space, and simplicity).

*i.e. front and back pointers for $\mathcal{O}(1)$ access to the front and back. Assume a singly-linked list.

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

ADT:	`Stack`	We want to match the most recent bracket we've seen first so we want LIFO properties.
underlying data structure:	Either `Array` or `LinkedList` with front	We can use an Array to efficiently simulate a stack by adding and removing elements from the back. A LinkedList will work if we add and remove elements from the front but LinkedList with front is simpler.

(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.

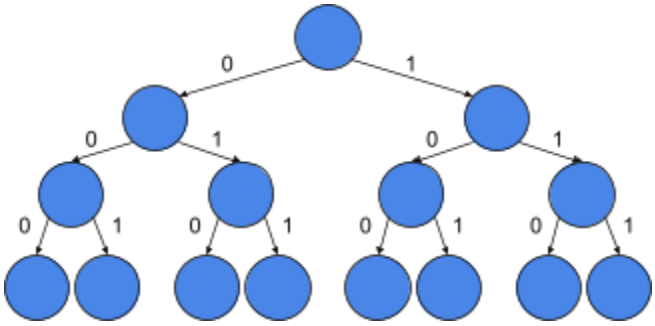
ADT:	`Queue`	We're dealing with a line that wants FIFO properties.
underlying data structure:	`Array` or `LinkedList` with front and back	The `Array` implementation will work if we implement it as a `CircularArrayFIFOQueue` (will be talked about later). We can use a `LinkedList` with front and back to efficiently simulate a Queue by adding elements to the front and removing elements from the back (or vice versa)). There is no way to make the `LinkedList` with front work efficiently; either adding or removing from the `Queue` will be slow.

(c) A sandwich shop wants to serve customers in the order that they arrived, but also wants to frequently look ahead to know what people have ordered (e.g. checking 1st person, 2nd person, ..., last person in line).

ADT:	`Queue`	Same as before, we're dealing with a line that wants FIFO properties.
underlying data structure:	`Array`	We need to access both ends of the data structure but also want to know what someone has ordered at a specific index. Only the array will let us do this. As a bonus, the adding or removing from the `Queue` can work efficiently if implemented as a `CircularArrayFIFOQueue`.

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

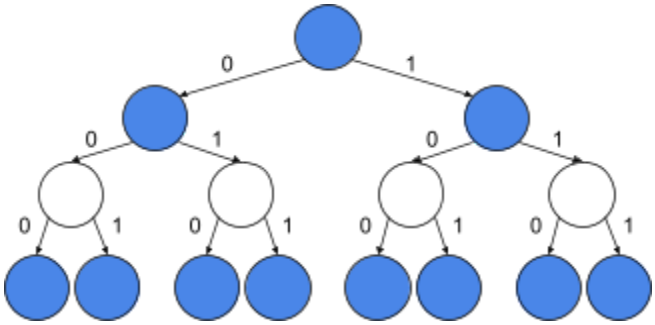
(a) Insert all possible binary strings of lengths 0-3 (i.e. "", "1", "0", "10", ..., "110", "111") into a `Trie`.



(b) From here, remove all binary strings of length 2 (e.g. "00"). How many nodes would disappear? Why?

0 nodes

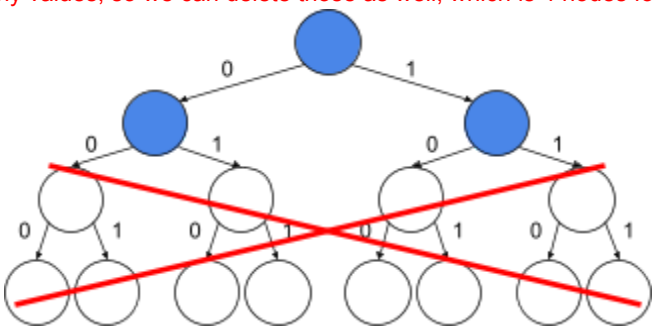
We still need the nodes storing the value for binary strings of length 2 because they have pointers to the nodes for binary strings of length 3, which still exist in the `Trie`. Therefore, we only set the value to `null` in the node to remove the key-value pair from the `Trie`.



(c) From here, remove all binary strings of length 3 (e.g. "000"). How many nodes would disappear? Why?

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 used 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? There are a few answers here.

One way to solve this would be using a `HashMap` where the keys are the area codes and the values are 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 the phone numbers to print.

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

If we compare the `HashMap` and `TrieMap` approaches, both will have the same runtime efficiency of $\Theta(n)$ to build and $\Theta(e)$ to call all the phone numbers with a particular area code efficiently.

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.

(c) What is the space complexity?

Asymptotically, the `Trie` will generally be more space-efficient.

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.

However, in practice, because each of the `Trie` nodes stores a pointer to the next node, it can quickly add up and take up a lot of memory.

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

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 a 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.