CSE 332: Data Structures and Parallelism

Section 1: `WorkLists`

0. 𝒪dd Jobs

For each of the following scenarios, choose
1. an ADT: Stack or Queue
2. a data structure: Array, singly LinkedList w/ front ptr, or singly LinkedList w/ front and back ptr

Then, justify your choice. Think about runtime, space, and simplicity.

`WorkList` Situations

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

Idea: Store opening symbols as we encounter them. When we come across a closing symbol, check if the most recent opening symbol stored matches the closing symbol. If it does, remove the opening symbol and continue. Otherwise, we have a mismatch.

Solution:
ADT: Stack
We need LIFO properties since we want to match the most recent opening symbol stored.
Data Structure: Array or LinkedList w/ front ptr
- Array can efficiently simulate a stack by adding and removing elements from the back.
- LinkedList can efficiently simulate a stack by adding and removing elements from the front.
- LinkedList approach 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.

Solution:
ADT: Queue
We need FIFO properties since we're dealing with a line.
Data Structure: Array or LinkedList w/ front and back ptr
- Array can efficiently simulate a queue if implemented as a CircularArrayFIFOQueue (will cover this later).
- LinkedList can efficiently simulate a queue by adding elements to the front and removing elements from the back.
- Without both the front and back ptr, either adding or removing from the queue will be slow.

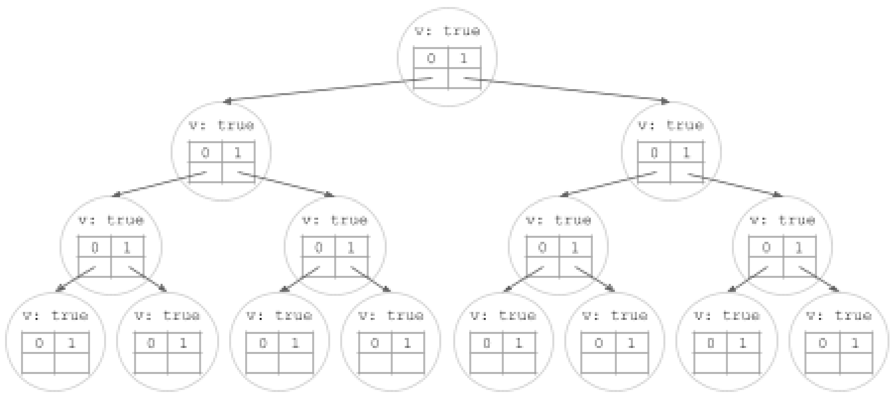
(c) A sandwich shop wants to serve customers in the order that they arrived, but also wants to look ahead to know what any other person has ordered (e.g. 5th, 3rd, or 8th person in line).

Solution:
ADT: Queue
Just like in Problem 0b, we need FIFO properties since we're dealing with a line.
Data Structure: Array
- Not only do we need to be able to efficiently access both ends of the data structure, but also every element in between.
- Both LinkedList options will not allow us to access a specific index efficiently.

1. `Trie` to Delete 0’s and 1’s?

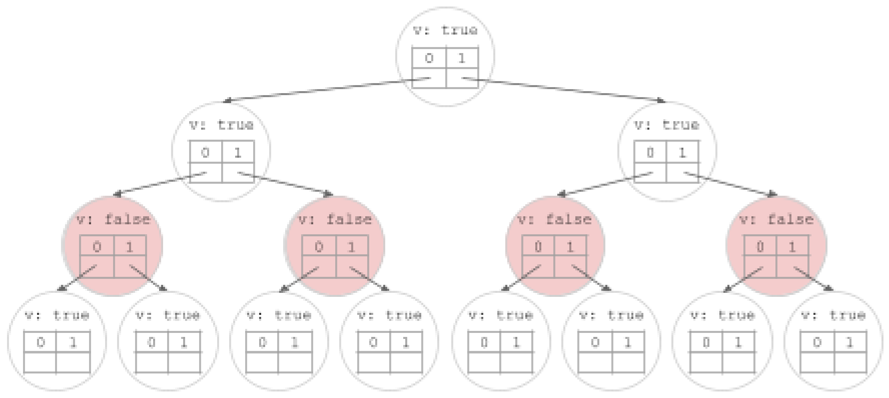
(a) Insert all possible binary strings of lengths 0–3 (i.e. ‘’, ‘0’, ‘1’, ..., ‘110’, ‘111’) into a trie.

Solution:



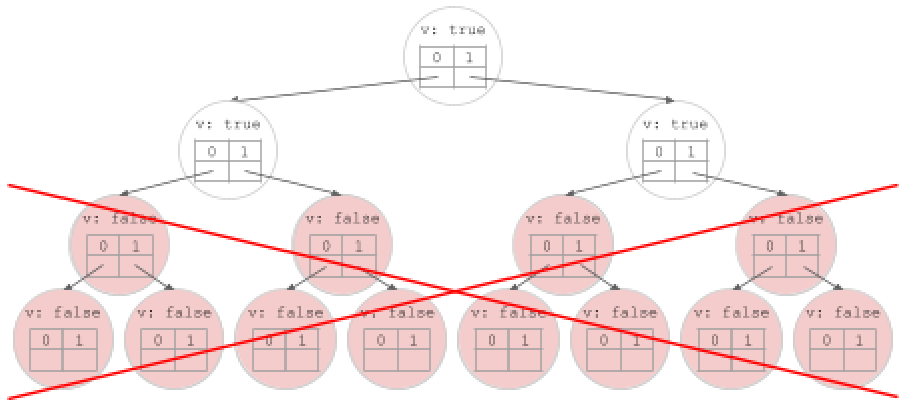
(b) From here, remove all binary strings of length 2. How many nodes would disappear? Why?

Solution:
0 nodes. We need to maintain pointers to nodes representing binary strings of length 3. Therefore, we only update the values in the nodes to null.



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

Solution:
12 nodes. We can delete all nodes representing binary strings of length 2–3 since these do not point to any relevant nodes.



2. Call Me Maybe

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.

(a) What data structure would you use and how would you implement it?

Solution:
There are multiple solutions to this problem, each with their own advantages and disadvantages.

One such solution is to use a HashMap where the keys are the area codes, and the values are a collection (e.g. List, Set, etc) of corresponding phone numbers.

- To add a phone number, we first extract the area code. Then we add the phone number to the collection of phone numbers corresponding to this area code.
- To find all the phone numbers for an area code, we simply retrieve the collection of phone numbers corresponding to the area code.

Another solution is to use a Trie, where phone numbers represent the path to the corresponding node, and nodes store *true*/*false* indicating a valid/invalid phone number.

- To add a phone number, traverse through the Trie and update the value in the corresponding node to *true*. Create nodes and edges along the path as needed.
- To find all the phone numbers for an area code, use the area code to partially travel down the trie, then visit all children nodes. If the value in a node is *true*, then it corresponds to a valid phone number.

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

Solution:
Let n be the total number of phone numbers, e be the number of phone numbers with a particular area code, and k be the length of a phone number.

HashMap:

- Inserting a phone number takes $\Theta(1)$ time. This means that inserting the entire phone book will take $\Theta(n)$ time.
- Given an area code, retrieving the corresponding collection of phone numbers takes $\Theta(1)$ time. Iterating through the collection takes $\Theta(e)$ time. This means that calling all the phone numbers for an area code takes $\Theta(e)$ time.

Trie:

- Inserting a phone number takes $\Theta(k)$ time. However, since the length of a phone number is a constant value, this is equivalent to $\Theta(1)$ time. This means that inserting the entire phone book will take $\Theta(n)$ time.
- Given an area code, it takes $\Theta(k) \equiv \Theta(1)$ time to visit and call each number. This means that calling all the phone numbers for an area code takes $\Theta(e)$ time.

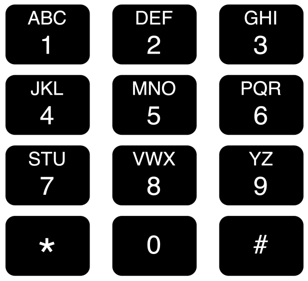
(c) What is the space complexity of your solution?

Solution:
Both the HashMap and Trie have $\Theta(n)$ space complexity, where n is the total number of phone numbers.

- The Trie is especially efficient if there are a lot of similar phone numbers. For example, if we have the phone numbers `123-456-7890`, `123-456-7891`, `123-456-7892`, then the HashMap must store each phone number individually, whereas the Trie is able to store them under the same path 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*, 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.