Week 6 Recitation Exercise

CS 261 – 10 points

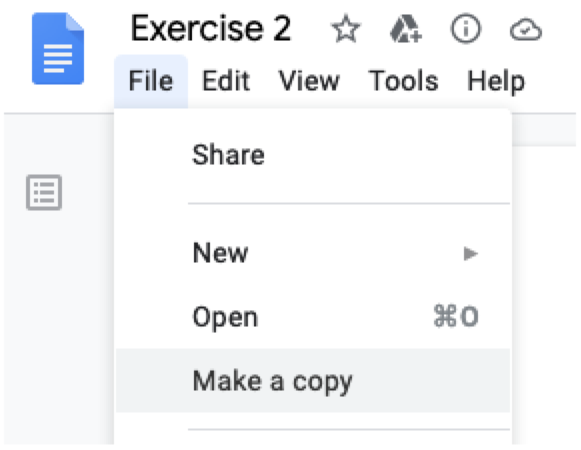
As we've talked about in lecture, data structures are the basis for many of the most popular interview questions for software development jobs. Typically, this kind of question asks the interviewee to use a specific data structure in a creative way to solve a given problem, and a number of these questions involve stacks and queues.

As developers who may need to answer this type of question in a real job interview some day, it will be useful to get some practice with these interview-style questions so you'll feel more confident about them when they come up. Even if you never interview for a software job, it is still useful to practice solving software job interview questions because they can challenge your thinking and help you learn to approach programming problems from new angles. Truly, these questions can be treated like brain teasers. For all these reasons, this recitation exercise involves thinking about and trying to solve a few common data structure-based job interview questions.

Follow the instructions below, and when you're finished, make sure to submit your completed exercise on Canvas. Please submit one copy per group (as long as all group members' names are listed below, it's fine if just one person submits). Remember that this exercise will be graded based on effort, not correctness, so don't worry if you don't get all the right answers. Do make sure you *try* to get the right answers though.

Step #1: Organize your group and create your submission document

Your TA will assign you to a small group of students. Among your group, select one group member to record your group's answers for this exercise. This group member should make a personal copy of this document so they are able to edit it:



Step #2: Add your names

Next, your group's recorder should add the names and OSU email addresses of all of your group's members in the table below:

Group member name	**OSU email address**
Hyun Taek, Oh	ohhyun@oregonstate.edu

Step #3: Practice some stack- and queue-based job interview questions

Below are four job interview-style questions that involve figuring out how to use stacks and queues in a clever way to accomplish various tasks. Read each question and discuss it with your group, then among your group, try to brainstorm potential solutions with your group.

One of the most important things to remember when you're answering these kinds of questions in a real job interview is to "think out loud." In other words, you want to talk through your reasoning process while you're figuring out how to approach and solve a problem. This is important because job interviewers are often more interested in figuring out *how* you think than they are in whether or not you get the right answer to a particular question, and thinking out loud can help the interviewer understand your thought process. Practice thinking out loud within your group as you approach these questions. In particular, don't just brainstorm solutions. Try to identify to the rest of your group the relevant parts of the problem, describe assumptions you might need to make, and explain *why* you make specific decisions in your proposed solution.

Once your group has a solution you're happy with for each question, describe that solution in the space provided under the question. Note that there's no need to write code to actually implement your solution. Instead, just describe your solution in plain English. Feel free to use pseudo code if it helps more clearly explain your solution.

Try to come up with solutions to as many of these questions as you can. If you're not able to come up with a specific solution for any of the questions, write down some "out loud" thoughts about the question.

Question #1: Bracket matching

Design a function that uses a single stack to determine whether the brackets in a given string are correctly matched. Specifically, the string can have three different kinds of brackets: curly braces ({}), square brackets ([]), and parentheses (()). A string contains correctly matched brackets if the following conditions are met:

- Every opening bracket in the string (i.e. every {, [, or () is eventually followed by a corresponding closing bracket (i.e. a },], or), respectively).
- The most recently opened bracket must be the first one closed. In other words, you can't close brackets out of order. For example, if brackets are opened like this {(, they must be closed like this)}, not like this }).
- No extra closing brackets exist in the string, i.e. there is no closing bracket that isn't preceded by a corresponding opening bracket.

Here are some examples of strings with correctly matched brackets:

- the quick brown fox jumped over the lazy dog (no brackets)
- the **{**quick**}** **[**brown**]** fox jumped **(**over**)** the lazy dog
- the **{{{**quick brown **(**fox**)** jumped**}}}** **[**over **{**the **(**lazy**)}]** dog

And, here are some examples of strings with *incorrectly* matched brackets:

- **{**the quick brown fox jumped over the lazy dog
- **[**the quick brown fox jumped over the lazy dog**)**
- **}**the quick brown fox jumped over the lazy dog**{**
- **[(**the quick brown fox jumped over the lazy dog**])** (wrong order)

[To design a function to determine the brackets in a given string are correctly matches by using a single stack, we have to understand and utilize characteristics of stack. Stack has a LIFO (Last- in, First-out) structure, which makes it convenient to check a pair of bracket. When a whole sentence with brackets in a stack, the first opening bracket in a sentence like the third one of examples in correctly matched brackets is in the bottom of the stack. Then, even whatever words or sentences in the middle of brackets, the first opening bracket will finally be met with closing one in the correctly matched case.

1. Define a single stack.
2. Store all the alphabets and brackets in a sentence from left to right into the stack.
3. Try to pop items in the stack and check it is an alphabet or bracket.
4. If the item popped is an alphabet, do not need to check.
5. However, if the item popped is a bracket, hold this item until it meets the opening one because opening bracket is in the deeper stack than closing one (by using hash table, it is much easier to match items you want to find.
6. When the brackets meet each other with same type and correct ordering, the brackets in the sentence is correctly matched.]

Question #2: Sorting the values in a stack

Design a function to sort the values in a stack so that the largest value is at the bottom of the stack and the smallest value is at the top. In addition to the stack being sorted, you may use a single additional stack as temporary storage, but you may not use any other data structure (i.e. no arrays, linked lists, queues, etc.).

[To sort the values from largest to smallest in a stack, when the unsorted stack popped, user check whether the value popped from a stack is higher than one popped from another stack. If the value from a stack is higher than another, the additional stack should make it empty for storing the higher value into the bottom of the stack. If not, just store it over the additional stack.

1. Define two stacks such as s1, s2.
2. Store all unsorted data into s1.
3. Pop the element in s1 and check it with the popped value from s2.
4. If the value of s1 is greater than one of s2, store higher value into s2 first, and then the value of s2 in s2.
5. If the value of s1 is smaller than one of s2, just store smaller value into s2.
6. Repeat 3-5 steps until finishing sort.]

Question #3: Stack from two queues

In assignment 2, you could earn extra credit by implementing a queue that used two stacks as its underlying data storage. In this question, do the opposite by designing a stack that uses two queues as its underlying data storage. In other words, your data structure will support the stack operations push() and pop(). When the user calls push(), the value being inserted must be placed into one of the two queues, as appropriate, and when the user calls pop(), your data structure will remove the correct element from one of the two queues, with reshuffling of the data between the two queues done as appropriate.

Hint: there are two good ways to implement a stack from two queues:

- *One way with an O(1) push() operation and an O(n) pop() operation.*
- *One way with an O(n) push() operation and an O(1) pop() operation.*

In either implementation, the O(n) cost is incurred by reshuffling values between queues to ensure they are popped in the correct order.

[With a characteristic of Queue (FIFO: First-In First-Out), a stack can be designed by two queues. A stack has a LIFO (Last-In First-Out) structure, which means a last element in a stack will always be popped first. To implement this, a last element in a queue should be dequeued first. It means the rest of them in a queue will enqueue in another queue to be entry of next stack pop. Then swap the roles of these queues until popping every element in both queues.

1. Define two queues such as q1, q2.
2. Enqueue all the elements into q1.
3. Dequeue some elements and enqueue these values into q2 until leaving only a last element in q1.
4. Then, the last element in q1 can be a top element of stack.
5. Dequeue this value to function like stack.
6. After that, swap the roles of both queues and do same steps 2-5 until popping all the values in both queues.]

Question #4: Converting decimal to binary

Write a function that uses a single stack and simple arithmetic operators (e.g. +, -, *, /, %) to convert a decimal (i.e. base 10) number into its binary representation. Specifically, your function should take a decimal number as an integer value and print out its corresponding binary string (i.e. it should print out a sequence of 1 and 0 characters).

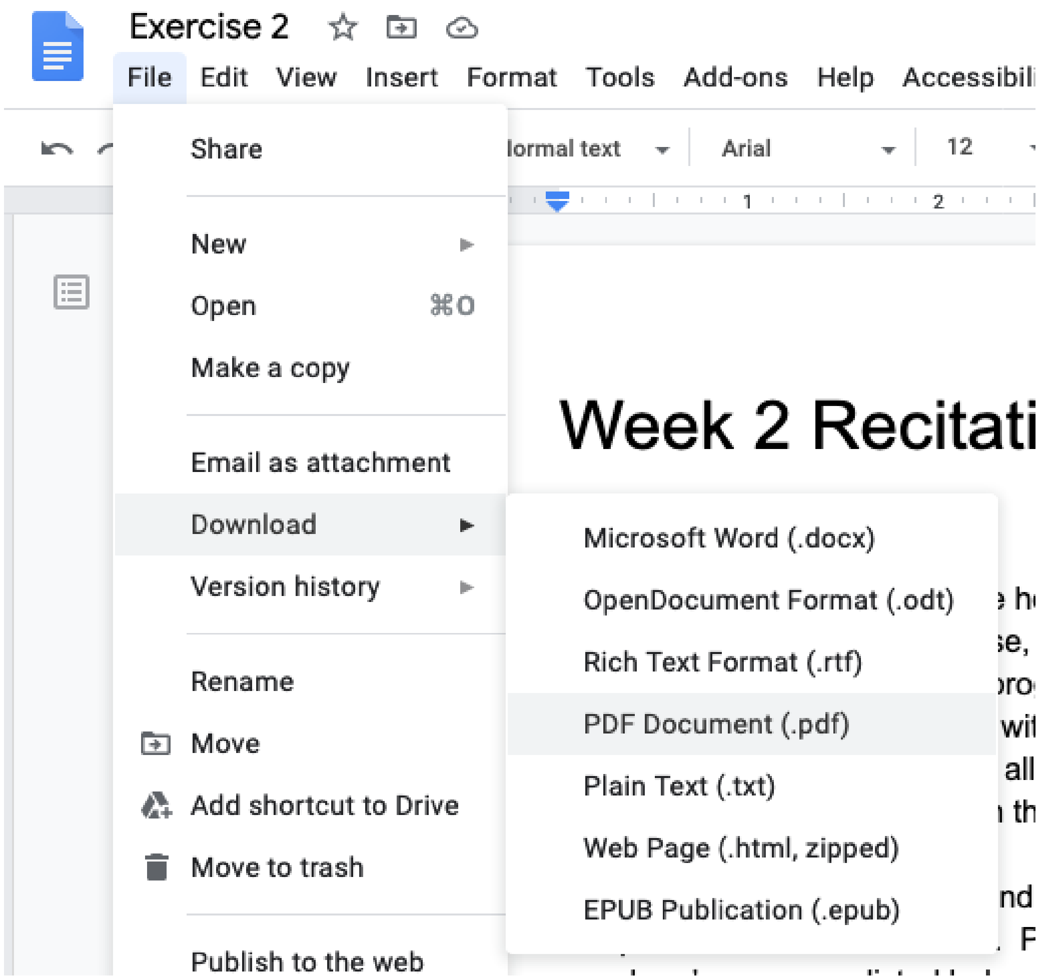
Hint: use the stack to store the individual bits (i.e. the individual 1's and 0's) of the binary representation. For an integer value n*, what does* $n \% 2$ *tell you? What does* $n / 2$ *do to the binary representation of* n*?*

[To store the individual bits into a stack, we have to understand the process of calculation, especially, division and mod operation. When a number is divided, a quotient can be a result of the division, and when a number is in mod operation, the result will be a remainder. Then, consider a characteristic of a stack (LIFO), which means the last number will be the leftmost digit. The result of mod operation should be first and then division calculation.

1. Define a stack such as s1.
2. Push a decimal number into s1.
3. Push '%' and '2' for mod operation and pop these for calculation.
4. After calculation, store that binary value into s1.
5. Then, push a number again with '/' and '2' for division calculation.
6. Push the Quotient that is the result of the division for next calculation.
7. Do same steps from 2-6 until the Quotient become zero.
8. Pop and print all the binary values in s1]

Submission and wrap-up

Once you've completed this exercise, download your completed version of this worksheet as a PDF and submit that PDF on Canvas (refer to the image below to help find how to download a PDF from Google Docs). Again, please submit one copy of your completed worksheet for your entire group. It's OK if just one member submits on Canvas. All group members will receive the same grade as long as their names and email addresses are included above.


Exercise 2
File
Edit
View
Insert
Format
Tools
Add-ons
Help
Accessibili
lormal text
Arial
12
Share
New
Open ⌘O
Make a copy
Email as attachment
Download
Version history
Rename
Move
Add shortcut to Drive
Move to trash
Publish to the web
Microsoft Word (.docx)
OpenDocument Format (.odt)
Rich Text Format (.rtf)
PDF Document (.pdf)
Plain Text (.txt)
Web Page (.html, zipped)
EPUB Publication (.epub)
Week 2 Recitati