CISC-235 Data Structures W22
Assignment 1

January 13, 2022

How to Submit: Important!

You need to prepare a folder that contains your code for each question. Name your folder as **your netid-A1**, zip the folder. To improve the readability of your answers, please separate the code for different questions. Avoid putting all code in one python file!

Write a README file (submitted in PDF format), specifying how to run the code for each question. TAs will read the README file to find the answers and code for grading. Question 1 requires a short description of your experiment design and how you determine the smallest value of k. Please include your answer to this question the README file as well.

To sum up, we ask for two files, a netid-A1.zip holding all your code, and a PDF file containing a short description of running your code for each question and experiment design description. **There will be a penalty of 10 points if your README file is missing or hard to read, you put all code in one file, or you do not correctly name your zip file.**

1 Binary Search or Linear Search? 50 points

Let us analyse the runtime complexity of two algorithms, i.e., linear search (Algorithm A) and binary search (Algorithm B) using experimental method.

Algorithm A: Store the list S in an array or list, in whatever order the values are given. Search the list from beginning to end. If $S[i] == x$ for any value of i, stop searching and return True (i.e., found), otherwise return False.

Algorithm B: Store the list S in an array or indexed list (such as the Python list). **Sort the list**. Use binary search to determine if x is in S. Return True or False as appropriate.

When using Algorithm A, searching for a value that is in the list will require an average of $n/2$ comparisons, while in the worst case, searching for a value that is not in the list will require n comparisons. When using Algorithm B, searching for any value will require an average of about $logn$ comparisons. However, sorting the list will take $O(nlogn)$ time.

If we are doing a very small number of searches, Algorithm A is preferable. However if we are doing many searches of the same list, Algorithm B is preferable since the time required to sort the list once is more than offset by the reduced time for the searches. This is what complexity theory tells us.

Your task is to conduct experiments to explore the relationship between the size of the list and the number of searches required to make Algorithm B preferable to Algorithm A. See the detailed requirement below:

1) Implement two algorithms using Python. When implementing Algorithm B, you must write your own sort function and your own binary search function. You may use any sort algorithm that has complexity in $O(nlogn)$.

2) For n = 1000, 5000, and 10000, conduct the following experiment:

   - Use a pseudo-random number generator to create a list S containing n integers.

   - For values of k ranging from 10 upwards:

   - Choose k target values, make sure half of which are in S and half are not in S, i.e., modeling the average case

   - Use Algorithm A and B separately to search the k target values in S.

   - Design and conduct experiments to determine the approximate smallest value of k for which Algorithm B becomes faster than Algorithm A. **Provide a short description on how you determine the smallest value of k and what is the smallest value of k you find.**

Other requirements:

- All data structures involved need to be implemented by yourself. Except for build-in data type, i.e., List in Python.
- Comment your code.
- You must write your test code in the main function.
- Coding style will be evaluated using PEP8[1].

Hints: To easily create a list of search values, half of which are in S and half of which are not:

- when generating the list S, use only even integer values. You can use step=2 in randrange function to control the interval of considered numbers. For instance

```
odd_rand_num = random.randrange(2,20,2)
```

[1] `https://realpython.com/python-pep8/`

will return any random number between 2 to 20, such as 2, 4, 6, ... 18. It will never select 20.

- randomly choose some elements of S as the first half of the search list
- randomly choose odd integer values as the second half of the search list

2 Implementing Stack: 15 points

Implement Stack using array-based list and linked list. Write a test code for validating the behavior of five functions mentoned below.

Your stack should support the following functions:

- **is_empty**, this function checks whether the current Stack instance is empty. It should return true if the Stack is empty.
- **push**, this function should takes a data item as input and add it into the Stack.
- **pop**, this function should return the data item on the top of the current Stack and remove it from the Stack.
- **top**, this function should return the data item on the top of the current Stack without modifying the Stack.
- **size**, this function should return the number of data items in the Stack.

Other requirements:

- All data structures involved need to be implemented by yourself. Except for build-in data type, i.e., List in Python.
- Comment your code.
- You must write your test code in the main function.
- Coding style will be evaluated using PEP8[2].

3 Implement a CircularQueue: 35 points

Implement a class named CircularQueue. CircularQueue implements a Queue ADT, but it stores elements in a circle. The below figure shows what happens if we add a sequence of elements (a, b, c) into a CircularQueue with an initial size of 5.

Your CircularQueue class should support the following functions, and you are free to add supporting attributes/classes/functions if needed. Note that you need to think about special cases. For instance, what if you want to enqueue an element to a full circular queue.

[2] `https://realpython.com/python-pep8/`

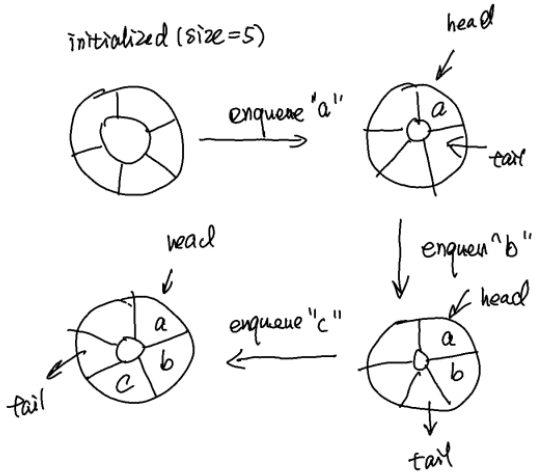

Figure 1: Sample CircularQueue

- An initialization function **__init__**. This function takes an int value as the input, and the input value determines the initialized capacity of the circular queue. You also need to define common attributes for supporting enqueue and dequeue function, e.g., head and tail. As shown in Figure 1, the tail is pointing to a position used to store the next coming element.
- An **enqueue** function. This function takes an element as input (string, int, etc.) and enqueues the item to the circular queue.
- An **dequeue** function. This function removes the earliest enqueued item from the circular queue and returns the item. For instance, if a, b, c are enqueued in sequence, dequeue will remove and return a because it is the earliest enqueued item.

Other requirements:

- All data structures involved need to be implemented by yourself. You can not import a queue class implemented by other libraries and directly call their functions. Except for build-in data type, i.e., List in Python.
- Comment test code.
- You must write your test code in the main function.
- Coding style will be evaluated using PEP8[3].

[3] `https://realpython.com/python-pep8/`