CSC 228 Data Structures and Algorithms, Spring 2020
Instructor: Dr. Natarajan Meghanathan

Exam 2 (Take Home)

Submission (in Canvas; See instructions in the last page) **Due: March 26th, by 11.59 PM**

Q1 - 30 pts) You will develop and implement algorithms to insert at an arbitrary index of a queue as well as delete the data at an arbitrary index of a queue. The index of an element (data node) in the queue is the position of the element (data node) starting from the front (first data node) of the queue.

For example, consider a queue as shown below:

Index	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
Data	34	21	90	45	87	22	43	55	81	98

where 34 is the data at the front of the queue (the data node for 34 is next to the head node) and 98 is the data at the end of the queue (the data node for 98 is previous to the tail node)

A call to the function to insert a new data (say, 40) at index 4 would result in the contents of the queue ordered as shown below:

Index	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
Data	34	21	90	45	**40**	87	22	43	55	81	98

A call to the function to delete data at index 7 on the above modified queue would result in the contents of the queue ordered as shown below (node that the data 43 at index 7 above is no longer available):

Index	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
Data	34	21	90	45	40	87	22	55	81	98

You are provided the code for the implementation of a Queue ADT using a doubly linked list. The code also has two auxiliary functions (*getQueueLength* and *Print*) to get the length of a queue and print the contents of the queue (starting from the data node next node the head node).

You are supposed to only use the enqueue and dequeue functions of the Queue class and the auxiliary function getQueueLength to implement the other two auxiliary functions *QueueInsertAtIndex*(Queue queue, int insertIndex, int insertData) and *QueueDeleteAtIndex*(Queue queue, int deleteIndex) that are currently blank in the code provided. Note that you could use some temporary variables inside these functions, but the space complexity should be $\Theta(1)$; i.e., the amount of additional space used should not grow with the queue size.

Note: If you insert or delete by simply traversing the doubly linked list (one node at a time) from the head node or the tail node and then inserting or deleting after an appropriate number of nodes have been traversed, then you will get a ZERO for this question. No Partial Credit will be given.

You need not modify the main function; it already has the code to test run your implementations. To demonstrate that your functions work as required, test run your code for a queue size of 10 with the range of values for the integer being (1... 100).
Run for the following test cases:

(i) Insert at index 4 and delete at index 7
(ii) Insert at index 0 and delete at index 9
(iii) Insert at index 9 and delete at index 0
(iv) Insert at index 0 and delete at index 10
(v) Insert at index 9 and delete at index 1

Q2 - 30 pts) For this question, you will explore the tradeoff between the average number of comparisons for a successful search of an element in a hash table vs. the load imbalance index. It is logical to expect that as the hash table size (the size of the array of linked lists representing the hash table) grows, the length of each of the linked lists reduces: as a result, the number of comparisons that would be needed to search for any element is likely to reduce. On the other hand, as we fix the number of elements to store in a hash table, the load imbalance index (the ratio of the sum of the maximum and minimum lengths of the linked lists and the difference between the maximum and minimum lengths of the linked lists; see the slides for the formulation and details) is expected to increase with increase in the hash table size. Thus, for a fixed number of elements to store in a hash table, as we increase the hash table size, the average number of comparisons for a successful search is expected to decrease and the load imbalance index is expected to increase. As part of your solution, you will explore/quantify the above tradeoff.

You are given the code featuring the implementation of a hash table as an array of singly linked lists. The main function is already coded to create an array of size 100,000 with the maximum value per element being 50,000. You will try 20 different values for the hash table size ranging from 11 to 2,287 as given in the code (note the array of hash table size is already created for you in the main function). For each hash table size, you will run 25 trials.

You are required to implement the functions FindAvgNumComparisonsSuccessfulSearch( ) and ComputeLoadImbalanceIndex( ) in the Hashtable class. The time complexity of each of these functions should be $\Theta(n)$ where n is the number of elements in the array. The main function is coded in such a way that these two functions are called for each of the trials for a certain hash table size and the average of the average number of comparisons for a successful search and the average load imbalance index are computed and printed for each value of the hash table size.

Take a screenshot of the output displaying the average number of comparisons for a successful search and the load imbalance index for different values of the hash table size.

Plot two Excel charts (X-Y plots) that features the values for the hash table size in X-axis and the values for the average number of comparisons for a successful search and the average load imbalance index in the Y-axes. Use the *trend line* option in Excel for each of these plots and determine a power function-based relation between the metric on the Y-axis and the hash table size in the X-axis. Display the power function-relation as well as the R^2 value for the fit in the Excel plots.

Q3 - 20 pts) You are given the code for storing integer elements of an array in a Hashtable. Modify the code in such a way that you could **print the number of occurrences of every unique element in the array in Θ(n) time**, where n is the number of elements in the array. You could modify any class (including the Node class) as well as add suitable member functions and/or modify existing member functions, if needed. Also, extend the main function to print the number of occurrences of every unique element in the array. Below is a sample screenshot of the output.

```
Enter the number of elements you want to store in the hash table: 25
Enter the maximum value for an element: 15
Enter the size of the hash table: 7
Elements generated: 0 6 12 1 9 8 4 2 7 11 6 5 7 5 14 1 8 7 9 14 6 0 2 10 8

Elements and their number of occurrences
0, 2
7, 3
14, 2
1, 2
8, 3
9, 2
2, 2
10, 1
4, 1
11, 1
12, 1
5, 2
6, 3
```

Q4 - 20 pts) A binary tree is a complete binary tree if all the internal nodes (including the root node) have exactly two child nodes and all the leaf nodes are at level 'h' corresponding to the height of the tree.

Consider the code for the binary tree given to you for this question. Add code in the blank space provided for the member function checkCompleteBinaryTree( ) in the BinaryTree class. This member function should check whether the binary tree input by the user (in the form of the edge information stored in a file) is a complete binary tree.

To test your code, come up with two binary trees of at least 12 vertices: one, a complete binary tree and another, a binary tree that is not a complete tree.

Prepare the input file for the two binary trees and input them to the code for this question. Capture the screenshots of the outputs.

Submission (in Canvas)

Submit separate C++ files for the codes as follows (so, a total of FOUR .cpp files as mentioned below):

Q1 - 25 pts) A separate .cpp file containing the completed version of the startup code provided, including the implementations of the *QueueInsertAtIndex* and *QueueDeleteAtIndex* functions.

Q2 - 25 pts) The entire code (that includes your implementations for the FindAvgNumComparisonsSuccessfulSearch( ) and ComputeLoadImbalanceIndex( ) functions in the Hashtable class).

Q3 - 18 pts) Complete C++ Code featuring the Node class, the List class (Singly Linked List), the Hashtable class and the main function

Q4 - 15 pts) The entire code (including the implementation of the function to check whether a binary tree is complete)

Submit a single PDF file that includes your screenshots and analysis for all the four questions put together as mentioned below (clearly label your screenshots/responses for each question):

Q1 - 5 pts) Test run your code for a queue size of 10 with the range of values for the integer being (1... 100). Submit screenshots of the output for the test cases (i)-(v) listed in the description of the question.

Q2 - 5 pts)

(i) Screenshot of the output displaying the hash table size, the average number of comparisons for a successful search and the load imbalance index.

(ii) Screenshots of the Excel plots (drawn as mentioned earlier in the question description) that also display the power function-based relation and the R^2 value for each.

Q3 - 2 pts) Screenshot of the output wherein the inputs are as follows: number of elements to store in the hash table is 30, the maximum value for an element is 20 and the size of the hash table is 11.

Q4 - 5 pts) Draw the two binary trees (along with their node ids) that you have come up with, include the contents of the text files (corresponding to these two binary trees) that are input to the program as well as screenshots of the outputs.

NOTE: If the instructor notices that you indulged in some cheating activity (like copying) for even one question, you will get ZERO for the entire exam. No Excuses!