**Homework 01 -- Questions 1 – 5**

1. What is a dynamic array? How does it work?

A dynamic array is a data structure which allows the programmer to store data in an array with alterable size. The dynamic array includes an array as instance data as well as variables that keep track of the size of the array (the actual length of the array) and the count of the array (how many elements are currently being stored in the array). When the program attempts to add an element to the array, it checks if the array is already full (that is, count == size). If it is not full, the program simply appends the element to the array. If it is full, the program calls another method of the dynamic array which grows the size of the array. To do this, it creates a new array with double the size of the old array and manually copies the data from the old array to the new array. This functionality allows the array to expand in size when necessary.

2. How is an Array different from a Linked List? Discuss in terms of the runtime and space for different operations.

The largest difference between arrays and linked lists comes from how the data is arranged and accessed in memory. With an array, the data is indexed, and each element of the array is stored consecutively in memory. With a linked list, however, the data is stored in nodes, each of which has a reference to the next node (and optionally a reference to the previous node in doubly-linked lists). These nodes do not need to appear consecutively in memory.

These differences create disparities in the efficiencies of arrays and linked lists for performing various operations. For example, the array is much more efficient at accessing elements than the linked list; accessing an element of an array simply requires the index of that element (giving us 0(1) runtime complexity), whereas the linked list must be traversed one node at a time until the desired node is found (giving us 0(n) runtime complexity). Inserting and deleting nodes, however, is relatively cheap for the linked list (0(1) runtime complexity), as it only requires updating the links of the list to either skip a node (in which case the node has no reference in the code its memory will be recycled) or include a new node. Inserting and deleting elements at the end of an array is similarly cheap, but inserting or deleting and the beginning or middle of an array requires the program to shift every element after the inserted or deleted element, which is very expensive (0(n) runtime complexity).

The linked list does require more memory space than the array for storing data, since the list must keep track of each node’s instance data as well as the reference to the next node. Despite this, adding elements to a linked list is relatively cheap in terms of memory space (0(1) space complexity), whereas adding a new element to a dynamic array when the array is full requires creating an entirely new array in memory (giving us 0(n) space complexity). The data of the original array has to be copied to the new array, which also makes this operation very costly in terms of runtime compared to adding a new node to a linked list.

3. Explain the scenarios where you can use linked lists and arrays.

Linked lists and arrays are useful for any situation where the program needs to store a set of similar data in a particular order. Linked lists are used for playlists of music or videos, for example. Arrays are particularly useful for storing numbered data since array elements are accessed by index. Arrays can also be multi-dimensional, which makes them useful for representing tables of data in a logical way. Since it is much cheaper to insert and delete elements from a linked list than it is for arrays, linked lists are more useful for data sets that require frequent insertion and deletion.

4. What is a Queue, and how is it different from a Stack? How can the Stack and Queue be implemented?

A queue is a data structure used to store and access data in a FIFO (first in, first out) order, which is different than a stack, which stores and accesses data in a LIFO (last in, first out) order. Adding items to a queue or stack is very similar; each item is simply appended to the queue or stack. However, when calling the dequeue() method for a queue, the front element of the queue is removed from the queue and returned by the program. Alternatively, when calling the pop() method for a stack, the element that was most recently added to the stack is popped off the stack and returned by the program.

The stack and queue can both be implemented using either an array or a linked list. Queues are best implemented using an array to make a circular queue. The queue keeps track of the front and rear positions in the array, and when the front or rear position reaches the end of the array and needs to be incremented, the index wraps around to the front of the array so that there is no unusable space at the front of the array. This kind of implementation can be done similarly with a linked list, but it requires more memory space. The stack can also be implemented using an array, although in this case there is no need for a circular array since the bottom of the stack never moves. Whenever at item needs to be popped off the stack, it is simply removed from the last used position in the array. Using a linked list is also possible, although the only real advantage gained from doing so is that the size of the stack becomes mutable.

5. Which data structures are used for implementing the LRU cache?

The LRU cache can be implemented using a linked list queue and a hash. The queue stores the list of recently used pages in order from least recently accessed to most recently accessed. When a page in the queue needs to be accessed, it is unlinked from its current position in the list and moved to the back of the queue. When the queue is full and a new item needs to be added, the front item in the queue is removed to make room for the new item. The hash is used to identify and access different pages inside of the queue efficiently when requested.

**Programming Problems (6 – 10)**

Question 6:

Discussion:

The program Matrix.java includes a constructor for generating a square matrix of size n x n using a 2D array as well as methods for rotating the matrix 90 degrees clockwise. The rotation is accomplished by treating the matrix as a set of “rings” and rotating four corresponding values of the matrix at a time. The first ring consists of all of the values on the outer edges of the matrix, and each additional ring consists of the values that would be on the outer edge, supposing that the previous ring was removed. Thus, in an even-dimensioned matrix, the innermost ring would be a 2 x 2 matrix, while in an odd-dimensioned matrix, the innermost ring would be a 1 x 1 matrix. To rotate the values in each ring, four values at a time are swapped starting with the corners of the ring and moving clockwise.

The iteration through the rings of the matrix is accomplished using a for loop. The rotation of the values in each ring uses another nested for loop that calls the function rotateValues() for each set of four values in the ring. The function takes as input the row and column indices for a specific cell in the matrix, and computes the values for the reflected row and reflected column indices (that is, the row and column obtained by reflecting the current row and column horizontally and vertically across the matrix respectively). These four indices are used to determine the locations to which each value will be rotated. The function first stores the first value in a temp variable, and then rotates the other three values, finally replacing the fourth value with the stored first value.

The runtime complexity of the rotation algorithm is 0(n2). The algorithm runs a for loop (n/2) times, inside of which it runs another loop (n – 1) times. Since we ignore constants in runtime analysis, this gives us 0(n2) runtime complexity. The space complexity of the algorithm is 0(1), however, since the algorithm only needs to create a few temporary variables in order to run.

Verification:

Question6.java is the driver class used to verify the correctness of the algorithm. The driver creates a square matrix using the constructor and then prints the input matrix using another method of the Matrix class. Then, it calls rotateMatrix() and prints the output matrix.

As test cases, we will use a 3 x 3 and a 4 x 4 matrix to test the algorithm for both even-dimensioned and odd-dimensioned matrices. The actual values within the matrix have no implications on the algorithm, so it is unnecessary to modify these.

3x3:

Input matrix: [[1,2,3][4,5,6][7,8,9]]

Output matrix: [[7,4,1][8,5,2][9,6,3]]

4x4:

Input matrix: [[1,2,3,4][5,6,7,8][9,10,11,12][13,14,15,16]]

Output matrix: [[13,9,5,1][14,10,6,2][15,11,7,3][16,12,8,4]]

Question 7:

Discussion:

The program Question7.java includes the code for this problem, which initializes and populates two linked lists with sorted values before merging the values from the second list into the first. To merge the lists while keeping the list sorted, the program runs a while loop that checks if the current index for the second list has surpassed the last element in the list. In the loop, the program compares the value at the current index in the first list to the value at the current index in the second list. If the item in the second list is less than or equal to the item in the first, it adds that item before the current element in the first list and then increments the indices of both lists. Otherwise, it increments only the index of the first list. This algorithm, however, is incapable of adding an item from the second list to the end of the first list. Therefore, we include a check before the main portion of the algorithm that simply adds the elements from the second list to the first list (in order) once the index of the first list becomes equal to its size.

The runtime complexity of the algorithm for this question is 0(n). The while loop runs a total of n times (where n is the length of the second list), and each operation within it has 0(1) runtime. The space complexity of the algorithm is also 0(n), since the algorithm needs space in memory for each element from the second list that it adds to the first list. Were we to access the actual nodes of the list rather than use Java’s built-in LinkedList class, we could change the links of the nodes in order to reuse them, in which case the algorithm would require no additional memory space.

Verification:

To verify this algorithm, we test several positive and negative values in the sorted lists, along with lists of different sizes as follows:

Input lists: [1, 2, 4], [-3, 3, 6]

Output list: [-3, 1, 2, 3, 4, 6]

Input lists: [-10, 7, 12, 27, 30, 54], [-3, 7, 40, 60]

Output list: [-10, -3, 7, 7, 12, 27, 30, 40, 54, 60]

Question 8:

Discussion:

The program Question8.java includes the code for this problem, which initializes the two linked lists and includes a method for converting the lists to integers using a stack as well as an algorithm for storing the sum as another linked list. To convert the lists to integers, the method listToInt() first initializes a stack and an empty string. It then pushes the items from the head of the list to the tail onto the stack. Afterwards, it pops the items off the stack and stores them in the string, which now holds the integer written forwards instead of backwards, which is returned by the method as an integer using the parseInt() method of the Integer class.

Once both lists are converted to integers, they can simply be added to obtain their sum. Then, the program stores each digit of the sum in a new linked list through a do-while loop. The loop parses the digit to be added using the modulo operator, dividing by 10 to take just the last digit of the sum. It then divides the list by 10, which effectively removes the last digit from the sum since both values are integers (not floating point values). The loop must be a do-while loop since the condition for the loop is that (sum != 0). If the sum of the two initial values is equal to 0, then we must ensure the loop runs one time even though this condition fails so that we can store 0 in the final list.

The runtime complexity for this algorithm is 0(n). The listToInt method runs two loops that are not nested, each of which performs n operations. Then, storing the sum of the two values in the final list requires another loop which performs n operations. Since none of these loops are nested and we ignore constants in runtime analysis, this gives us the complexity 0(n). The space complexity is also 0(n), since the algorithm needs to put each value from each list onto a stack and put the final value into a new list. Each of these memory usages would be additive rather than multiplicative in complexity, giving us 0(n) space complexity.

Verification:

To verify this algorithm, we test several values of different numbers of digits in the initial lists, as well as the case where both lists are initially zero.

Input lists: [2, 4, 3], [5, 6, 4]

Output list: [7, 0, 8]

Input lists: [9, 8, 3, 5, 9], [2, 1, 6, 8]

Output list: [1, 0, 0, 4, 0, 1]

Input lists: [0], [0]

Output list: [0]

Question 9:

Discussion:

MyStack.java is the class that includes the methods for a stack implementation using two queues. Since the Queue is only supported natively as an interface in java, we instantiate the two queues as linked lists, as the LinkedList class implements the Queue interface and therefore gives us access to the necessary methods. One of the two queues acts as the stack, while the other queue acts as an auxiliary queue used to ensure the values in the stack are in the right order. When pushing an element onto the stack, we first enqueue the element into the auxiliary queue. Then, we dequeue each element from the stack queue and enqueue it onto the auxiliary queue. The elements are now in the right order with the new element included, so we simply dequeue and enqueue each element back onto the stack queue.

The other three methods of the stack are relatively straightforward. To pop and element, we simply remove it from the front of the stack queue and return it. To peek at the top element, we use the peek() method of the queue. Lastly, the queue has its own implementation of the isEmpty() method, so we only need to have the stack’s empty() method return the same value.

The runtime for the pop, top, and empty methods of the MyStack class are each 0(1) since they perform one operation, and each requires no extra memory. The runtime complexity for the push operation is 0(n), however, since we need to use two separate loops using the two queues to perform the operation, each of which runs n times. The space complexity for this operation would be 0(n) as well, since the auxiliary queue must be updated with every element on the stack each time we push a new element.

Verification:

We use a driver class (Question9.java) to test the functionality of each of the stack’s methods:

Input:

myStack.push(5);

System.out.println(myStack.top());

myStack.push(10);

myStack.push(11);

System.out.println(myStack.top());

System.out.println(myStack.empty());

System.out.println(myStack.pop());

System.out.println(myStack.top());

System.out.println(myStack.pop());

System.out.println(myStack.pop());

System.out.println(myStack.empty());

Output:

5

11

false

11

10

10

5

true

Question 10:

Discussion:

StringDecoder.java contains the code for the decoding algorithm. The algorithm acts recursively, first separating the string into three pieces: the initial string (substring of characters before a multiplier k), the bracketed string (substring to be multiplied by k), and the remainder string (substring occurring after the bracketed section). The initial string will always come before a multiplier, so it does not need to be decoded and can be immediately added to the final string. The bracketed string may need to be decoded, so it is passed recursively to the decode() method again. If it does not need to be decoded, the base case will simply return the string. Once the bracketed string has been decoded, it is multiplied by its multiplier and added to the final string. Lastly, the remainder string may need to be decoded, so it is also passed recursively to the method. The decoded remainder string is then added on to the final string, giving us the final decoded string.

The runtime complexity of the decode algorithm would be 0(n\*log(n)). The decode function calls itself twice, and each recursive call of the decode function for an input size n has an input size proportional to (n/2). The base case and other functions in the algorithm result in total 0(n) runtime, which is smaller than 0(n\*log(n)), so it is unnecessary to include this. The space complexity is 0(n), since each call of the function only uses 0(1) space for its operations.

Verification:

Question10.java contains the driver code for this problem. We verify the algorithm using a few test cases including nested encoded strings, encoded bracketed strings and remainder strings, and unencoded initial strings and remainder strings. This diversity ensures the algorithm acts accordingly for all program paths.

Input: 3[a]2[bc]

Output: aaabcbc

Input: 3[a2[c]]

Output: accaccacc

Input: 2[abc]3[cd]ef

Output: abcabccdcdcdef