Quentin Creese
Jess
CS 260
DUE: 01/22/24

Assignment 2 (Design)

1. Based on what we know about linked lists, stacks, and queues, design a queue data structure:
    a. What functions are we likely to need for a queue to function like the one discussed in class?

Some functions we would need for a queue are:
enqueue(value): A function that adds an element to the rear end of the queue.
dequeue(): A function to remove and return the element from the front of the queue.
peek(): A function that returns the element at the front of the queue without removing it.
isEmpty(): A function to check whether a queue is empty.
size(): A function that returns the number of elements currently in the queue.
clear(): A function that removes all elements from the queue, leaving it empty.

Case-sensitive functions:
display(): A function displays all elements in the queue.
front(): A function returning a reference to the element at the front of the queue w/o removing it.
rear(): A function that returns a reference to the element at the end of the queue.
copy(): A function that creates a copy of the queue.
merge(queue): A function that merges another queue into the current queue
iterator(): A function that returns an iterator object for iterating through the elements of a queue.

    b. What values will we need to know about the structure for our queue to function properly?

Capacity: Define the maximum number of elements that a queue can hold. This allows us to manage memory and avoid some overflow conditions.
Front and Rear Pointers: This shows the positions of the queue's front and rear elements, which are essential for enqueue() and dequeue().
Size/Length: Used to represent the current number of elements in the queue. This allows us to determine whether the queue is empty or full.
Data Type: Specifying the queue's data type ensures type safety and proper memory allocation.
Underlying Data Structures: Underlying data structures used for queues can include arrays, linked lists, or dynamic arrays.
Overflow and Underflow Conditions: Define the conditions under which the queue is considered full (overflow) and empty (underflow). This will ensure the correctness of my queue operations.
Error Handling: Define the expected errors and conditions to handle these errors.

2. Based on what we know about linked lists, design a list data structure that allows us to add (insert) or remove (delete) values at a given location in the list (instead of the top of a stack or the front or back of a queue):

    a. What functions are we likely to need for a list to function like this?

insert(value, position): A function that allows us to insert a new node with the given value at the specified position in the list.
remove(position): A function that removes the node at the specified position from the list.
get(position): A function that retrieves the node's value at the specified position in the list.
size(): A function that returns the list's number of elements (nodes).
isEmpty(): A function that checks whether the list is empty.
clear(): A function that removes all elements from the list, leaving it empty.

Special Cases:
append(value): A function that appends a new node with a given value to the end of the list.
contains(value): A function that checks whether the list contains a node with the specified value.
index(value): A function that returns the index (position) of the first occurrence of the specified value in the list.
reverse(): A function that reverses the order of elements within the list.
slice(start, end): A function that returns a sublist containing elements from the specified start to end positions
iterator(): A function that returns an iterator object for iterating through the elements of the list.

    b. What values will we need to know about the structure for our list to function properly?

Node Structure: Define the structure of a node in the list, including the value it holds and a pointer to the next node.
Head Pointer: Keep track of the head (first node) of the list.
Tail Pointer (if using a singly linked list): Optionally, keep track of the tail (last node) of the list for efficient appending.
Size/Length: Maintain the current number of nodes in the list to implement size() and isEmpty() functions.
Positioning: Implements logic to handle inserting and deleting nodes at specific positions in the list, including edge cases such as inserting at the beginning or end of the list.
Error Handling: Define how errors or exceptional conditions will be handled, such as through exceptions, error codes, or assertions.