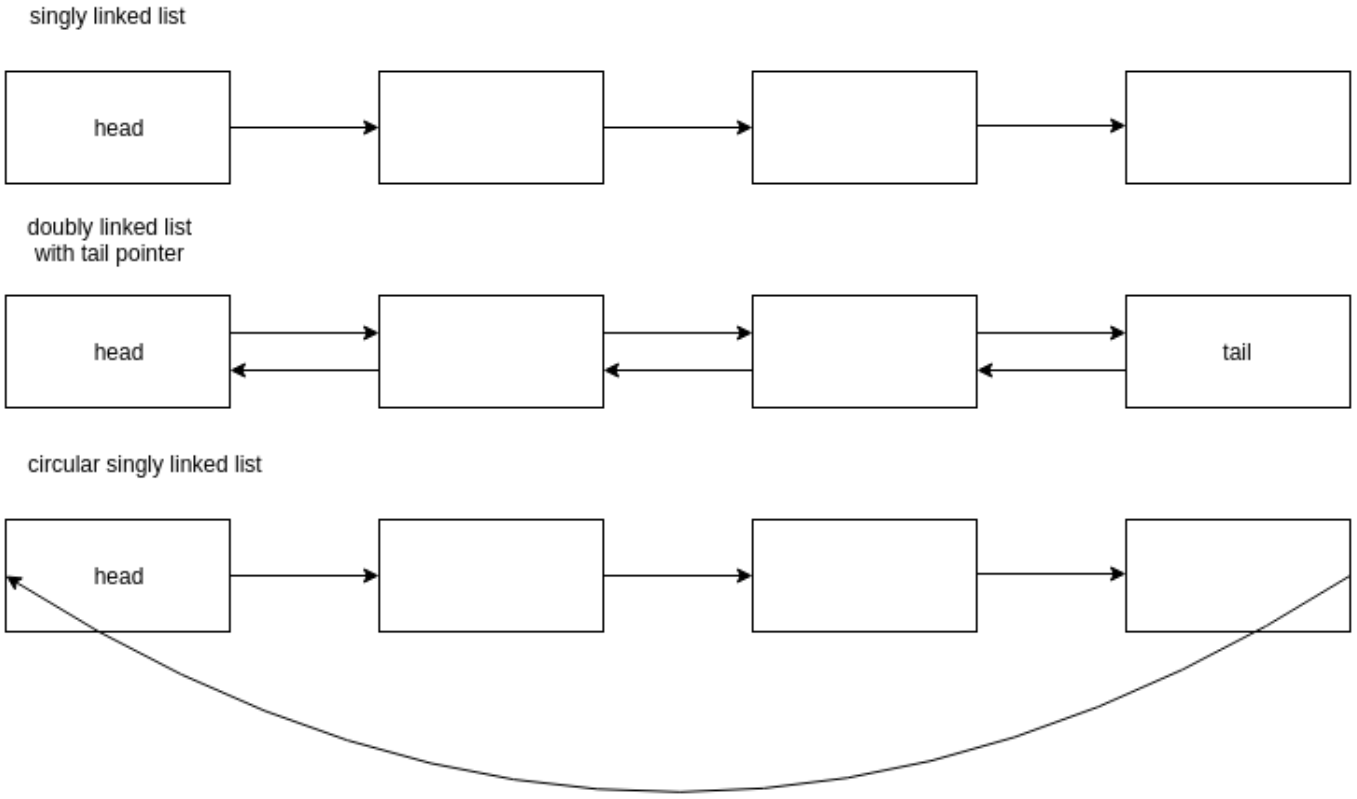
Overview

Due: October 7, 11:59:59PM

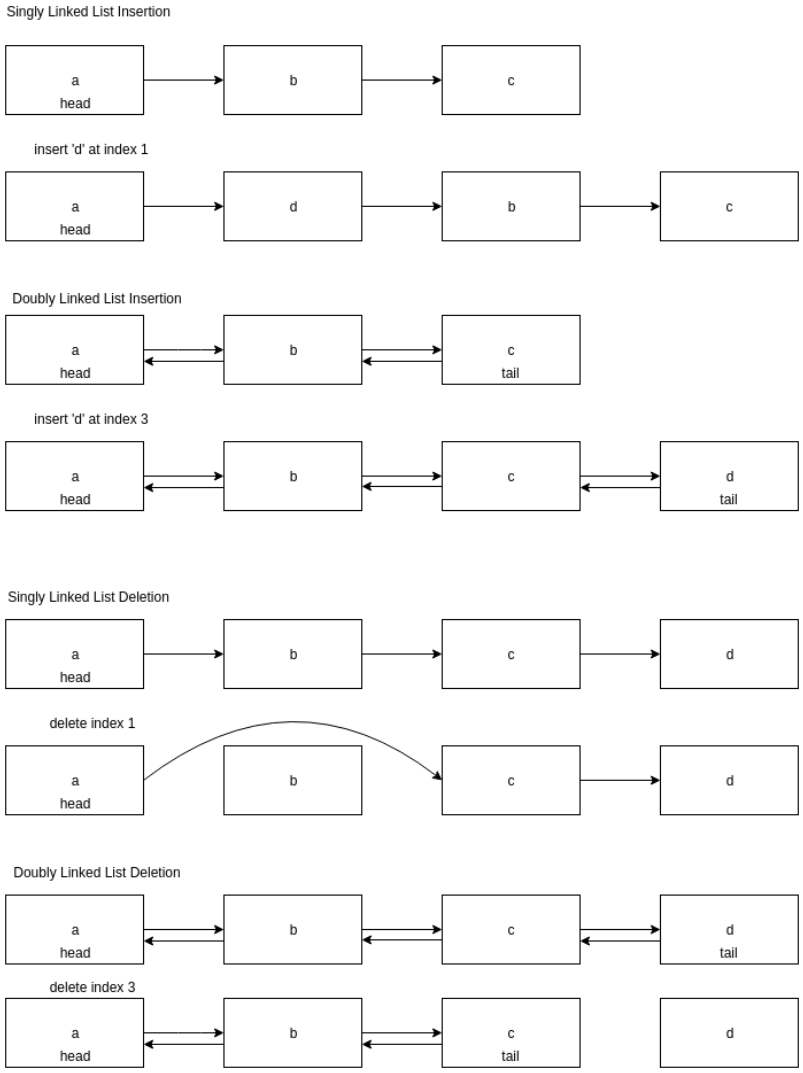
This assignment is to implement simple data structures that operate efficiently using the provided skeleton code and test cases. In this assignment, you will implement the following:

- Singly Linked List
- Doubly Linked List
- **Optional Bonus:** Circular Buffer (built on top of a Circular Singly Linked List)



Insertion and Deletion Rules

As you can see from the example figure below, insertion and deletion for Singly and Doubly Linked Lists work similarly. We do not actually 'delete' the node, but instead remove all references to it. Once every reference to a node has been removed, it is no longer possible to reach the node and it will be cleaned up by the Java Garbage Collector.
Circular Buffers behave the same as Circularly Singly Linked Lists with a *tail* pointer (except for the last element's *next* reference) until a certain number of elements is reached. This is explained further in the Circular Buffer section.



Part 1 - *Singly Linked List*

A Singly Linked List is a well-known data structure consisting of a series of nodes that each point to another node in a linear fashion. One node is designated as the *head* and is the start of the linked list. The last element in the list will not point to anything (implemented as a null value in Java). Note that this version of the Singly Linked List may differ from the version discussed in the lectures and textbook.

You have been provided with a SinglyLinkedList.java file that you will edit and use to implement all of the required features. Directions:

- Write your name and student ID number in the comment at the top of the file. Please write your name in English letters, not 한글.
- Implement all of the required functions in the SinglyLinkedListNode class. You will need it to complete the rest of this part.
- Implement all of the required functions in the SinglyLinkedList class.

Part 2 - *Doubly Linked List*

A Doubly Linked List is similar to a Singly Linked List except that each node points to the next and previous nodes in the linked list. Additionally, we will be using a variant that has both a *head* and a *tail* pointer. This allows for $O(1)$ insertion and deletion from the end of the list, instead of $O(n)$ as in our singly linked list. Since there is nothing before the *head* node, its *prev* pointer is null. Similarly, the *tail*'s *next* pointer is null. Note that this version of the Doubly Linked List may differ from the version discussed in the lectures and textbook.

You have been provided with a DoublyLinkedList.java file that you will edit and use to implement all of the required features. Directions:

- Write your name and student ID number in the comment at the top of the file. Please write your name in English letters, not 한글.
- Implement all of the required functions in the DoublyLinkedListNode class. You will need it to complete the rest of this part.
- Implement all of the required functions in the DoublyLinkedList class.

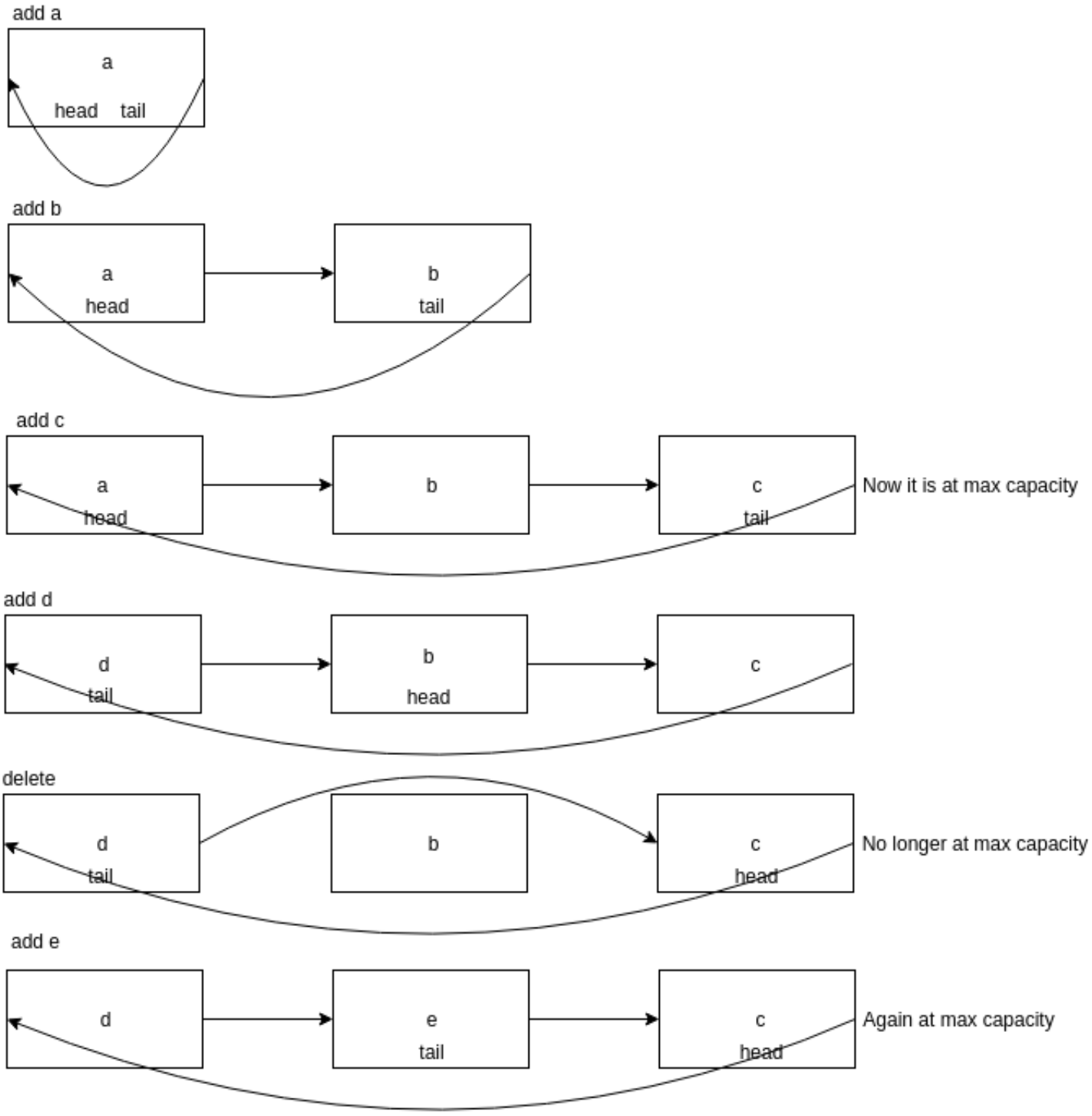
Part 3 - *Circular Buffer* (Optional Bonus)

A Circular Buffer is similar to a Circular Singly Linked List with a tail pointer. Circular referring to the fact that the *tail* a pointer to the *head*, unlike the Singly Linked List where the *head* had nothing pointing to it. Another difference is that a circular buffer has a limit to the amount of elements it can hold, referred to as the capacity. Once the capacity is reached, if a new element is added, it will overwrite from the *head* of the buffer. The *head* will then move to the next element in the circular buffer. If elements are deleted so that the buffer is not at full capacity, it reverts to its usual insertion and deletion rules. For simplicity, this implementation will only allow addition to the end of the Circular Buffer and deletion at the beginning of the Circular Buffer. When searching for an element in the Circular Buffer, the *head* is still considered to be index 0. The figure on the next page shows an example when the capacity is 3. Pay close attention to the *head* and *tail* pointers.

You will also have to implement a check for overflowing, which is when there is an addition to the buffer while it is at full capacity. Any time you call a function (not just the ones that alter the buffer) on a Circular Buffer, you should determine if it caused an overflow and update the overflow flag accordingly.

You have been provided with a CircularBuffer.java file that you will edit and use to implement all of the required features. Directions:

- Write your name and student ID number in the comment at the top of the file. Please write your name in English letters, not 한글.
- Implement all of the required functions in the CircularBufferNode class. You will need it to complete the rest of this part. Other than the name, it is exactly the same as the SinglyLinkedListNode class.
- Implement all of the required functions in the CircularBuffer class.


add a
a
head tail
add b
a
head
b
tail
add c
a
head
b
c
tail
Now it is at max capacity
add d
d
tail
b
head
c
delete
d
tail
b
c
head
No longer at max capacity
add e
d
e
tail
c
head
Again at max capacity

General Notes

- Do not change any method or class signatures. You should only edit inside of the functions. If your code changes any class or method names or signatures, you will receive an automatic 0. You should not need to implement any other functions besides the ones that are provided, however, doing so is acceptable.
- You should carefully consider all edge and boundary cases when implementing these data structures.
- All of our data structures are assumed to be indexed from 0. Meaning the *head* is located at index 0. When inserting into position k, it means placing the item in between the items at positions $k-1$ and k. If $k = 0$, it means placing the item immediately before the *head*.
- Make sure your code compiles. Non-compiling code will receive a 0. If you have a problem that is causing you to not be able to compile, it may be better to just comment out the incorrect code and return a dummy value (something like null or -1) so the rest can compile. We *may* award some partial credit if your commented out code is extremely close to the correct solution.
- When adding, deleting, or looking up an index, the only valid indices are $0...n-1$ or $0...n$, where n is the size of the list or buffer. Do not use any sort of modulo to 'wrap around' the list. The only time the indices $0...n$ are valid is when adding to the end of a linked list. For example, if we have a linked list of size 5, add('b',5) should add to the end of the list. However, getNode(5) should not work since only indices $0, 1, 2, 3, 4$ are valid.

Testing

You have been provided with a small test suite for each of the required classes. This will test the most basic procedures so you know if you are on the right track or not. They will not cover all (or even most) edge and boundary cases, so you should implement your own testing to check for these. The test suite that we will use to grade your homework will be much more comprehensive, but you will not be able to see it until after the submission deadline. You may change these test files in any way you wish, as they will not be submitted.
You can test your code by compiling your code and the test suite corresponding to the class you want to test. Then you can execute the test suite to view how you scored. For example:
javac SinglyLinkedList.java
javac SinglyLinkedListTest.java
java SinglyLinkedListTest

Deliverables

You should submit **only** the following items in a zip-file called "PHW1":

- SinglyLinkedList.java
- DoublyLinkedList.java
- CircularBuffer.java (optional)

Having an incorrect zip-file name will result in points being deducted. If your java files are named incorrectly, they automatically receive a 0.