CS 1332 Exam 1 PLUS Session - Spring 2023

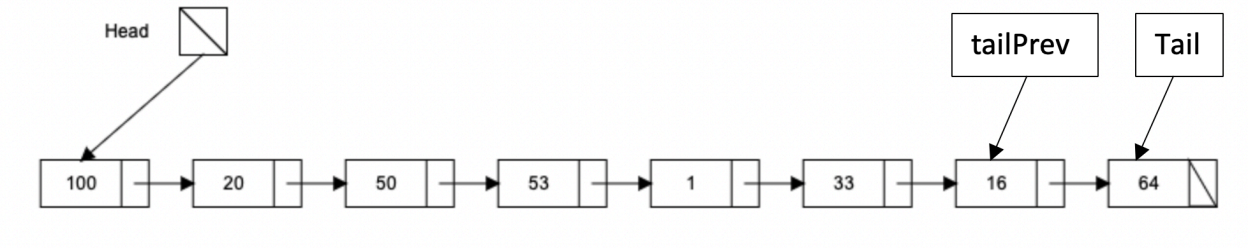
CS 1332 EXAM 1 TOPIC LIST

- Arrays
- Lists
    - ArrayLists (With coding)
    - LinkedLists
        - Singly
        - Doubly
        - Circular Singly (With Coding)
        - Circular Doubly
- Linear Data Structures
    - Stacks, both linked and array-backed
    - Queues, both linked and circular array-backed
    - Deques, both linked and circular array-backed
- Binary Trees
    - Shape Properties (full, complete, degenerate)
    - Traversals (preorder, inorder, postorder, levelorder)
- Binary Search Trees
    - Operations: search, add, remove

Big O Questions

- (Easy) What is the worst case cost of adding to the back of an ArrayList?
- (Easy) What is the average case cost of removing the middle element of a doubly linked list with size n?
- (Medium) What is the worst case time complexity of adding to a SLL-backed stack without a tail pointer?
- (Medium) What is the worst case cost of calling addToBack(data) to a CSLL? The CSLL does NOT have a tail pointer.
- (Medium) Suppose you add n elements to an array-backed stack? What is the best case time complexity of retrieving the first element added to the stack?
- (Medium) What is the worst case time complexity of removing an element from a linked-list backed queue and adding to an array-backed deque?
- (Medium) What is the worst case cost of searching for data in a complete BST?
- (Medium) What is the average case cost of performing a level-order traversal?
- (Hard) What is the worst case cost of clearing a SLL of size n with a tail pointer by calling removeFromBack() n times?

- (Hard) Assume you have a SLL with a tail pointer and a tailPrev pointer that points to the node directly before the tail (see image below). What is the average case efficiency of removeFromBack() from this modified SLL?



- (Hard) What is the worst case time complexity of creating a list of data of all the elements in a BST from largest to smallest element?

Diagramming

1. ArrayList

Assume you have an ArrayList backed by an array of capacity 5. If a resize is necessary, double the array's current capacity. Perform the list of instructions below and show the resulting ArrayList

a.

addToFront(1)

addToBack(5)

addToFront(3)

addAtIndex(data: 7, index: 1)

addAtIndex(data: 9, index: 3)

b.

addToFront(2)

removeFromFront()

removeFromBack()

2. Queues (Array-Backed)

Assume you have an empty array with capacity 7. If a resize is necessary, double the array's current capacity. Perform the list of instructions below on the empty initial array. Be sure to indicate where front points to and what size is after each set of instructions.

a.

Enqueue 4

Enqueue 7

Enqueue 3

Dequeue

b.

Enqueue 1

Enqueue 9

Enqueue 8

Enqueue 5

Enqueue 6

c.

Enqueue 10

Dequeue

Dequeue

Dequeue

3. Queues (SLL-Backed)

Assume you have an empty SLL with a head and tail pointer. Perform the following operations below and determine the resulting SLL.

Enqueue 10

Enqueue 8

Dequeue

Enqueue 3

Enqueue 4

Dequeue

4. Stacks (Array-Backed)

Assume you have an empty array with capacity 5. If a resize is necessary, double the array's current capacity. Perform the list of instructions below on the empty initial array. Be sure to indicate what size is after each set of instructions.

a.

Push 2

Push 9

Pop

Push 3

b.

Push 10

Push 0

Push 7

Push 4

Pop

Pop

5. Stacks (SLL-Backed)

Assume you have an empty SLL with a head pointer. Perform the following operations below and determine the resulting SLL.

Push 5

Push 10

Push 1

Pop

Push 6

Pop

Pop

6. Deques (Array-Backed)

Given the following deque, perform the following operations. The resizing policy is to double the current backing array's capacity.

			3 [front]	

- addFront(5)
- addBack(8)
- addBack(9)1
- removeFront()
- addFront(1)
- addBack(2)
- addBack(10)

7. Deques (Array-Backed)

Assume you have an empty backing array with capacity 7. Perform the operations below and diagram the populated array. Be sure to indicate the location of the front pointer.

a.

addFront(3)

addBack(2)

addFront(1)

addBack(5)

b.

addFront(6)

addBack(3)

addBack(9)

c.

addFront(4)

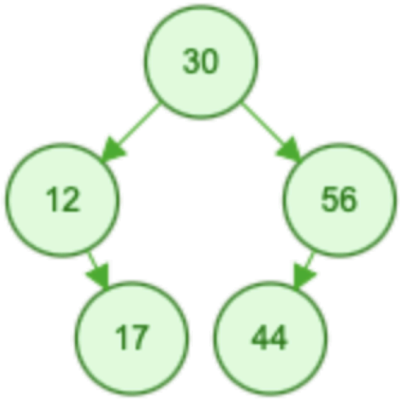
removeFront()

removeBack()

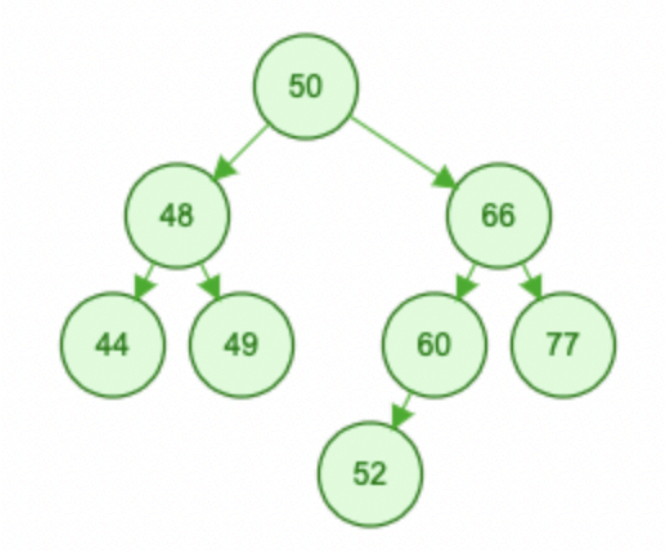
removeFront()

8. BST Operations

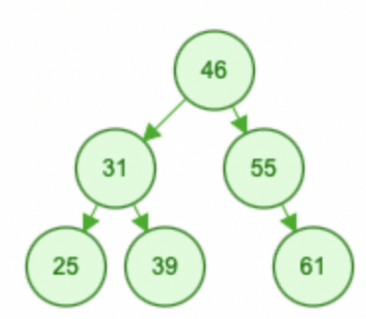
a. Add 25 to the tree below



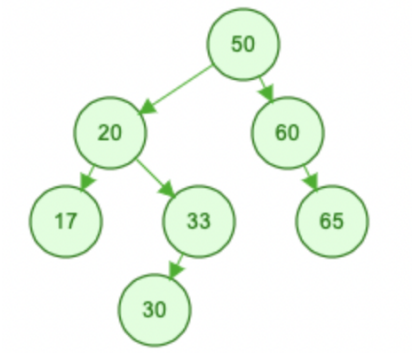
b. Add 51 to tree below



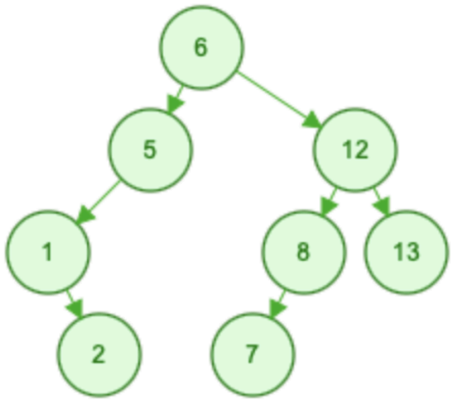
b. Remove 46 from the tree below (using predecessor if needed)



c. Remove 33 from the tree below (use successor if needed)



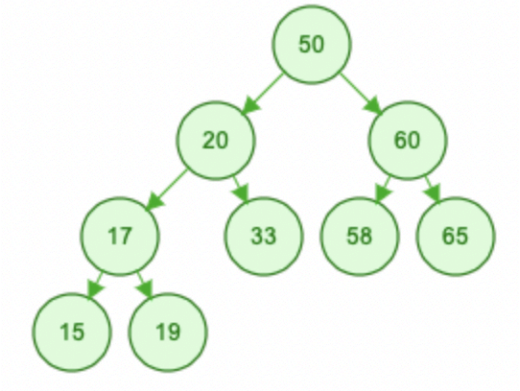
9. BST Traversals:
Given the following BST, perform the traversals below and write the data in the order they appear for each traversal.



Pre Order:
In Order:
Post Order:
Level Order:

10. BST Shape Properties:

a. (Multi-Select) Select all of the following that apply to the BST below. If you select none of the above, you should only select none of the above.



Full
Complete
Degenerate
None of the above

b. (Multi-Select) Select all of the following that apply to the BST below. If you select none of the above, you should only select none of the above.

Full
Complete
Degenerate
None of the above

Coding

1. ArrayList - removeFirstOccurrence()

Goal: Given the following ArrayList class with private variables backingArray and size, implement the removeFirstOccurrence(T data) method. This method will remove the first occurrence of *data* in the backingArray if it exists. Recall that an ArrayList must be contiguous, meaning elements must be shifted when removing an element.

Requirements:

- Your code must be as efficient as possible
- If *data* does not exist in the backingArray, you must throw a NoSuchElementException()
- You may assume that *data*, *size*, and *backingArray* are nonnull and valid
- You should not use any other methods in the ArrayList class
- The data may occur in the ArrayList more than once, but you must remove the first occurrence, i.e. the data closest to index 0
- The *size* variable must reflect the new size of the *backingArray* after the removal
- You must return the first occurrence of the data in the *backingArray*, not the *data* parameter passed into the method

Examples:
Initial *backingArray*: [10, 5, 2, 10, 6, null, null]
removeFirstOccurrence(10) is called
Final *backingArray*: [5, 2, 10, 6, null, null, null]

Initial *backingArray*: [6, 0, 5, 4, null]
removeFirstOccurrence(1) is called
A NoSuchElementException() is thrown as 1 does not exist in *backingArray*

```java
import java.util.NoSuchElementException;

public class ArrayList<T> {

    private T[] backingArray;
    private int size;

    //implementation omitted

    /**
     * Removes and returns the first occurrence of the inputted data
     *
     * If the data is not found, throw a NoSuchElementException
     * Hint: Take advantage of your size variable and make sure you
     * terminate as soon as possible to be efficient.
     *
     * @param data the data to be removed from the list (assume data is
     * not null)
     * @return the data that was removed (NOT the passed in data)
     * @throws java.util.NoSuchElementException if data is not found
     **/
     public T removeFirstOccurrence(T data) {
       //YOUR CODE HERE
     }
}
```

2. Circular Singly Linked List - calculateSize()

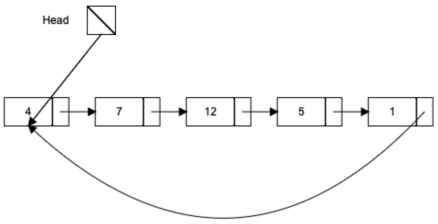
Goal: Given the following Circular Singly-Linked List class (CSLL), implement the calculateSize() method.

Requirements:

- Your code must be as efficient as possible
- You cannot assume that the input to the method is nonnull
- You must calculate the size of the CSLL (size is not given to you)

Example:

Input CSLL:



calculateSize() returns 5 since there are 5 nodes in the CSLL

```
public class CSLL<T> {

    private class CSLLNode<T> {
        public T data;
        public CSLLNode<T> next;
    }

    //implementation of other methods omitted

    /**
     * Calculates the size of the CSLL (size is not given!). You must handle
     * any edge cases and return the appropriate integer value representing
     * the size of the CSLL. Consider what a CSLL is if size = 0 and size = 1.
     *
     * @param head the head of the CSLL
     * @return int that denotes the number of nodes in the CSLL
     *
     **/
    public int calculateSize(CSLL<T> head) {
        //YOUR CODE HERE
    }

}
```

3. ArrayLists - addBefore()

Goal: Given the following ArrayList class which has private variables *backingArray* and *size*, implement the addBefore(T data, T arrayData) method. This method will add the inputted data *data* to the *backingArray* at the index directly before the index where *arrayData* appears. This method must not delete any elements that were initially in the *backingArray*, meaning elements must be shifted to account for *data* being added to the *backingArray*.

Requirements:

- Your code must be as efficient as possible
- You may assume there are no duplicate elements and that *arrayData* exists at a valid index in the *backingArray*
- You may assume that all inputs are non-null and valid
- You may assume that *backingArray* is large enough to accommodate at least one more element, so **no resize is required**
- The *size* variable must be updated to reflect the new size of the *backingArray* after the addition
- You cannot use any other methods of the ArrayList class

Example:
Initial *backingArray*: [4, 1, 7, 20, 2, null]
addBefore(10, 7) is called
Final *backingArray*: [4, 1, 10, 7, 20, 2]

```java
public class ArrayList<T> {
    private T[] backingArray;
    private int size;

    //implementation omitted

    /**
    * Adds 'data' to the index directly before the index of 'arrayData'.
    *
    * @param data the data to add to the backingArray
    * @param arrayData the data currently in the backingArray (data is
     * added before this element)
    */
    public void addBefore(T data, T arrayData) {
        //YOUR CODE HERE
    }
}
```