Midterm 1 Notes

Q1. Difference Between ArrayList and LinkedList?

ArrayList:

- Implemented using an array with a fixed size.
- When capacity is reached, doubles its capacity and copies elements to new array.
- Fast random access to elements due to contiguous memory locations.
- Ideal for situations where random access to elements is required, such as when implementing a database or search algorithm.

LinkedList:

- Implemented using nodes that point to each other.
- Each node stores a reference to the next node, allowing for quick insertion and removal of elements.
- Accessing a particular element can be slower than with an ArrayList since nodes are not stored in contiguous memory locations.
- Ideal for situations where frequent additions and removals are required, such as when implementing a queue or stack.

In summary, you would use an ArrayList when you need fast random access to elements and do not need to add or remove elements frequently, while you would use a LinkedList when you need to frequently add or remove elements from the List but do not need fast random access to elements.

Operations	ArrayList	LinkedList
	Complexity	Complexity
get(int index)	O(1)	O(n/4)
add(E element)	O(1)->O(n)	O(1)
remove(int index)	O(n/2)	O(n/4)
add(int index, E element)	O(n/2)	O(n/4)
Iterator.remove()	O(n/2)	O(1)
ListIterator.add(E element)	O(n/2)	O(1)

Q2. Suppose you have some List of Integers. Write a method that finds the minimum and maximum values stored in the list and returns their sums, regardless of implementation of the List.

```
import java.util.List;`
public class MinMaxSum {
public static int minPlusMax(List<Integer> list){
int min = Integer.MAX_VALUE;
int max = Integer.MIN_VALUE;`


    for (int i = 0; i < list.size(); i++) {
        int current = list.get(i);
        if (current < min) {
            min = current;
```

```
        }
        if (current > max) {
            max = current;
        }
    }

    return min + max;
}
```

Common ArrayList Methods

```
import java.util.ArrayList;`
public class ArrayListExample {
public static void main(String[] args) {
// Create a new ArrayList of Strings
ArrayList<String> list = new ArrayList<String>();`

    // Add some elements to the list
    list.add("apple");
    list.add("banana");
    list.add("cherry");

    // Access elements in the list
    System.out.println("Element at index 1: " + list.get(1));

    // Modify elements in the list
    list.set(0, "orange");

    // Remove an element from the list
    list.remove("banana");

    // Check if the list contains an element
    System.out.println("List contains 'cherry': " + list.contains("cherry"));

    // Get the size of the list
    System.out.println("List size: " + list.size());

    // Print out the entire list
    System.out.println("List contents: " + list);

    // Convert the list to an array
    String[] array = list.toArray(new String[list.size()]);

    // Print out the array
    for (String element : array) {
```

```
        System.out.println(element);
    }
}
```

}

Sort an Array of integers

```
import java.util.ArrayList;
import java.util.Collections;  // Import the Collections class

public class Main {
  public static void main(String[] args) {
    ArrayList<Integer> myNumbers = new ArrayList<Integer>();
    myNumbers.add(33);
    myNumbers.add(15);
    myNumbers.add(20);
    myNumbers.add(34);
    myNumbers.add(8);
    myNumbers.add(12);

    Collections.sort(myNumbers);  // Sort myNumbers

    for (int i : myNumbers) {
      System.out.println(i);
    }
  }
}
```

ArrayList Lab

```
public static <E> boolean Uniquness(List<E> A) {
    for (int i = 0; i < A.size(); i++ )
        for (int j = i +1; j < A.size(); j++)
            if (A.get(i).equals(A.get(j)))
                return false;
    return true;
}
public static ArrayList<Integer>  AllMultiples(List<Integer> A, int x) {
    ArrayList <Integer> B = new ArrayList<>();
    for (int i = 0; i < A.size(); i++)
       if (A.get(i) % x == 0){
```

```
            B.add(A.get(i));
        }
    return B;
}
public static ArrayList<String> allStringsofsizes(ArrayList<String> A, int x) {
    ArrayList<String> C = new ArrayList<>();
    for (int i = 0; i < A.size(); i++)
        if (A.get(i).length()== x){
            C.add(A.get(i));

        }
    return C;
}
public static <E extends Comparable<E>> boolean isPermutation(ArrayList<E> A, ArrayLis
t<E> B) {
    if (A.size() != B.size()) {
        return false;
    }
    for (E item: A) {
        int countA = 0;
        int countB = 0;
        for (int i = 0; i < A.size(); i++) {
            if(A.get(i).equals(item)){
                countA++;
            }

        }
        for (int i = 0; i < A.size(); i++) {
            if (B.get(i).equals(item)) {
                countB++;
            }
        }
        if (countA != countB) {
            return false;
        }
    }
    return A.equals(B);


}
public static ArrayList<String> StringToList(String A) {
    ArrayList<String> split_string = new ArrayList<>();
    for (String _string: A.split("\\s+")) {
        split_string.add(_string);
    }
    return split_string;

}
public static List<Integer> removeAllInstances(List <Integer> list ,int x){
    for (int i= list.size()-1; i >= 0; i--) {
        if(list.get(i)==x){
            list.remove(i);
        }
    }
    return list;
}
```

Linked List

```
public class LinkedList {
private Node head;



private static class Node {
    private int data;
    private Node next;

    public Node(int data) {
        this.data = data;
        next = null;
    }
}
//Adding a node
public void add(int data) {
    Node newNode = new Node(data);

    if (head == null) {
        head = newNode;
    } else {
        Node current = head;
        while (current.next != null) {
            current = current.next;
        }
        current.next = newNode;
    }
}
//Removing a node
public void remove(int data) {
    if (head == null) {
        return;
    }

    if (head.data == data) {
        head = head.next;
        return;
    }

    Node current = head;
    while (current.next != null) {
        if (current.next.data == data) {
            current.next = current.next.next;
            return;
        }
        current = current.next;
    }
}
```

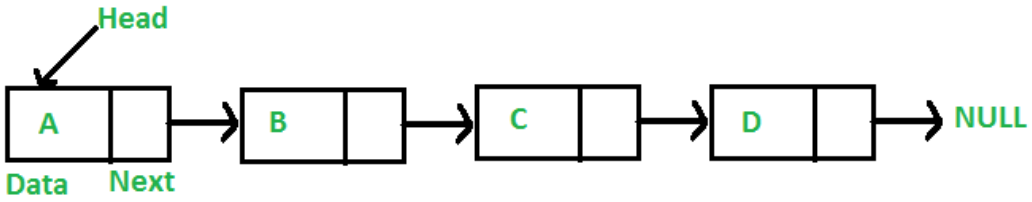
```
public void printList() {
    Node current = head;
    while (current != null) {
        System.out.print(current.data + " ");
        current = current.next;
    }
    System.out.println();
}
```

This implementation defines a LinkedList class with a private Node class that contains an integer data value and a reference to the next node in the list. The LinkedList class has an instance variable head that points to the first node in the list.

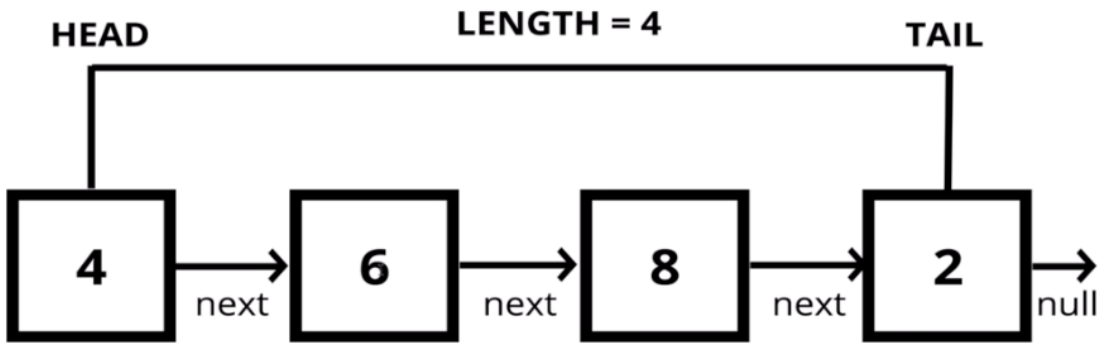
The add method adds a new node with the given data value to the end of the list. If the list is empty, the new node becomes the head. Otherwise, the method traverses the list until it reaches the last node and then adds the new node as the next node.

The remove method removes the first node in the list with the given data value. If the node to be removed is the head, the method updates the head to point to the next node. Otherwise, the method traverses the list until it finds the node to be removed and then updates the next pointer of the previous node to skip over the removed node.

The printList method traverses the list and prints out the data values of each node.



Singly Linked Lists



SINGLY LINKED LIST OPERATION	REAL TIME COMPLEXITY	ASSUMED TIME COMPLEXITY
Access i-th element	O(√N * N)	O(N)
Traverse all elements	O(√N * N)	O(N)
Insert element E at current point	O(1)	O(1)
Delete current element	O(1)	O(1)
Insert element E at front	O(1)	O(1)
Insert element E at end	O(√N * N)	O(N)

Reverses a String

```
public List<String> reverseWords(List<String> list) {
List<String> reversedList = new ArrayList<String>();
for (String s : list) {
String reversedString = new StringBuilder(s).reverse().toString();
reversedList.add(reversedString);
}
return reversedList;
}
```

Deletes all the items in LinkedList

```
public void deleteList() {
Node<E> current = head;
while (current != null) {
Node<E> next = current.next;
current.prev = null; // remove reference to previous node
current.next = null; // remove reference to next node
current = next;
}
head = null; // set head to null
tail = null; // set tail to null (if tail is present)
}
```

Different types of linked lists

1. Singly-linked list: Each node in a singly-linked list contains a reference to the next node in the list. This type of linked list is simple and efficient, but can be limiting because each node can only be traversed in one direction (forward).
2. Doubly-linked list: Each node in a doubly-linked list contains references to both the previous and next nodes in the list. This type of linked list allows for efficient traversal in both directions, but requires more memory than a singly-linked list because each node needs to store an extra reference.
3. Circular linked list: In a circular linked list, the last node in the list points back to the first node, forming a circle. This type of linked list is useful for applications where the list needs to be traversed repeatedly in a loop, but can be more difficult to implement and reason about than a linear linked list.
4. Sorted linked list: A sorted linked list is a linked list where the elements are kept in sorted order. This can be useful for certain applications, such as maintaining a priority queue, but can be slower to modify than an unsorted linked list.

Write an instance method called count. Count iterates over a LinkedList and returns
the number of times item occurs in the list.

// This is an instance method, so you can access the head, tail, and the Node class
public int count(E item){

```
public int count(E item) {
int count = 0;
Node<E> current = head;
while (current != null) {
if (current.data.equals(item)) {
count++;
}
current = current.next;
}
return count;
}
```

Write a static method that takes in two sorted linked lists of integers and merges them into
one sorted linked list and returns it. Since this a static method outside of LinkedList, you cannot
access the Nodes. You are allowed to remove items from listA and listB
public static LinkedList<Integer> merge(List<Integer> listA, List<Integer> listB){
(2 points) What is the time complexity of this algorithm?

```
public static LinkedList<Integer> merge(List<Integer> listA, List<Integer> listB) {
LinkedList<Integer> mergedList = new LinkedList<Integer>();
while (!listA.isEmpty() && !listB.isEmpty()) {
if (listA.get(0) < listB.get(0)) {
mergedList.addLast(listA.remove(0));
} else {
mergedList.addLast(listB.remove(0));
}
}
mergedList.addAll(listA);
mergedList.addAll(listB);
return mergedList;
}
```

Write a method that reverses a doubly-linked LinkedList. This method will be an in-
stance method for LinkedList, so you can use the

Node head and tail references and refer to the LinkedList object using the this keyword.
// You have access to the Node class
public void reverse(){

```
public void reverse() {
if (head == null) {
return; // empty list, nothing to reverse
}
Node<E> current = head;
Node<E> temp = null;
while (current != null) {
// swap next and prev pointers for current node
temp = current.prev;
current.prev = current.next;
current.next = temp;
// move current to next node
current = current.prev;
}
// update head and tail references
temp = head;
head = tail;
tail = temp;
}
```

Here's an implementation of the `add` method for the `SortedLinkedList` class:

```
public class SortedLinkedList<E extends Comparable<E>> {private Node<E> head;
private Node<E> tail;
private int size;

private static class Node<E> {
    private E data;
    private Node<E> next;
```

```
    public Node(E data) {
        this.data = data;
    }
}

public void add(E data) {
    Node<E> newNode = new Node<>(data);
    if (head == null) {
        head = tail = newNode;
    } else {
        Node<E> curr = head;
        Node<E> prev = null;
        while (curr != null && data.compareTo(curr.data) > 0) {
            prev = curr;
            curr = curr.next;
        }
        if (prev == null) {
            newNode.next = head;
            head = newNode;
        } else if (curr == null) {
            prev.next = tail = newNode;
        } else {
            newNode.next = curr;
            prev.next = newNode;
        }
    }
    size++;
}

// other methods...
```

}

Change a singular linked list to a doubly linked list in Java by modifying the node structure of the linked list.

public class DoublyLinkedListNode<E> extends LinkedListNode<E> {
private DoublyLinkedListNode<E> prev;

```
public DoublyLinkedListNode(E data) {
    super(data);
    prev = null;
}
```

```
public DoublyLinkedListNode<E> getPrev() {
    return prev;
}

public void setPrev(DoublyLinkedListNode<E> prev) {
    this.prev = prev;
}
```

}

Write a Java method to remove all nodes in a linked list that have duplicate values.

```
public void removeDuplicates(Node head) {
if (head == null) {
return;
}
Node current = head;
while (current != null) {
Node runner = current;
while (runner.next != null) {
if (runner.next.data == current.data) {
runner.next = runner.next.next;
} else {
runner = runner.next;
}
}
current = current.next;
}
}
```

Sure, here's an example Java method that detects if a linked list is a palindrome:

```
public boolean isPalindrome(Node head) {
if (head == null || head.next == null) {
// A list with 0 or 1 node is always a palindrome
return true;
}
// Find the midpoint of the list
Node slow = head, fast = head;
while (fast != null && fast.next != null) {
slow = slow.next;
```

```
fast = fast.next.next;
}
// Reverse the second half of the list
Node prev = null, current = slow, next;
while (current != null) {
next = current.next;
current.next = prev;
prev = current;
current = next;
}
// Compare the first and second halves of the list
Node p1 = head, p2 = prev;
while (p2 != null) {
if (p1.data != p2.data) {
return false;
}
p1 = p1.next;
p2 = p2.next;
}
// Restore the original list by reversing the second half again
prev = null;
current = prev;
while (current != null) {
next = current.next;
current.next = prev;
prev = current;
current = next;
}
return true;
}
```