CS 310 - Program 1 Report

Name: Trevor Cargile

RED ID: 813542789

rohan#: masc0328

Analysis of methods:

LinkedListDS-

**public void addFirst(E obj);**

The addFirst method is O(1) and this is because there is no traversing through the linked list. It sets the next variable of the new node being added to the current head of the list, and then changes the head node to the new node being added. There are also some checks to avoid null pointers.

**public void addLast(E o);**

Just like addFirst, addLast is O(1). Because we have a tail pointer for the linked list we can go straight to that pointer, change the current tail of the linked list to point at a new node being added, then change the tail pointer to point to the new node. There is no traversing the list in this method.

**public E removeFirst();**

My removeFirst method is also O(1). We don't care about the previous node in this case because there are no nodes before the beginning of the linked list. This allows us to just set the head pointer to point to the node after the first node in the linked list, then the garbage collector will destroy the unused node.

**public E removeLast();**

The removeLast method is O(n). It isn't O(1) because we need to know the previous node to the last node in the list. Since we don't have a doubly linked list, we need to traverse the whole list and create a temp node to store the next to last node in the list. This way, we can set the tail pointer to the next to last node, and unlink the last node from the list for garbage collecting.

**public E peekLast();**

This simple methos is O(1). All that this method does is look to the tail pointer and return what it points to. There is no traversing the list.

**public boolean contains(E obj);**

The contains method is O(n). This is because the worst case scenario for the method is O(n), the best case is O(1), so the average is n/2, which is O(n). The method starts at the beginning of the list and go until it finds the list or hits null, which indicates that the list doesn't contain the object that was sent to it. The compareTo method is called, but this doesn't change the complexity because whether the list has 10 or 100 nodes, it will always compare only 2 objects.

**public E find(E obj);**

I chose to have the find method use the contains method because both methods do the same thing. If the method returns true, then I return the obj sent to the find method, otherwise I'll return null. This mean that the find method is O(n), because the contains method is also O(n).

**public boolean remove(E obj);**

The remove method is also O(n). The method needs to search for the object in the list, which uses the same formula for the find/contain methods. Then if it finds it, it sends it to one of three cases. If it is in the front of the list, removeFirst() is called, which is O(1) so it doesn't affect speed. If it is at the end, removeLast() is called, which is also O(1) so the speed is affected either. If it isn't in either of those and still exists in the list, then no method is called, and the previous node is made to point to the node's next variable. My search function stores the previous node for this purpose.

**public void makeEmpty();**

This simple method is O(1). This is because the method only changes two variable and doesn't traverse through any list. It simply makes the head and tail pointers equal to null, which unlinks all nodes for garbage collecting. Then it resets the currentsize.

**public boolean isEmpty();**

The isEmpty method is also O(1). This is because it only looks at one variable regardless of size - the head node. If the head node is null then that is an indication that the list is empty. You don't need to look at the tail node.

**public boolean isFull();**

Since linked lists can never become full (for the purpose of this assignment, since space isn't infinite), this method will always return false, which makes this method O(1).

**public int size();**

The size method is O(1) because it looks for one variable and returns its value regardless of how big the list is.

**public Iterator<E> iterator();**

The iterator method overwrites the original iterator method is order to help implement search functions for the linked list. There is no method contained within the Iteratorhelper() method that traverses the list completely, so this method should be O(1). There is an iterator pointer that will store the position of the current node. The next() method is O(1) because it goes to the next node in the list, no traversing needed. The hasNext method only checks to see if the current node is equal to null, which is signal the end of the list, making it O(1) as well. The remove method has no function and will always return an exception, making this O(1) too.

Stack-

**public void push(E obj);**

The push method just uses the inherited addFirst method so this method is O(1) just like its parent method.

**public E pop();**

Pop uses the removeFirst method to pop the last thing added to the stack. Since all this method does is call removeFirst, it has the same complexity, which is O(1).

**public int size();**

The size method calls the parent size method to get the currentSize of the list. The parent size method is O(1), so this method is O(1) as well.

**public boolean isEmpty();**

Returns the call to the parent isEmpty method. The parent isEmpty method only checks one variable and has a complexity of O(1), so this method is O(1).

**public E peek();**

Looks at the top of the stack and returns it. This method calls the peekFirst method from the LinkedListDS class and returns the object. Since the peekFirst method is O(1), so is this peek method.

**public boolean contains(E obj);**

Calls the parent contains method, which has a complexity of O(n). Since this method does nothing more, the complexity of this contains method is the same at O(n).

**public void makeEmpty();**

Calls the parent makeEmpty method. Since the makeEmpty method only sets the head and tail to null, independent of list size, the complexity is O(1).

**public boolean remove(E obj);**

This remove method calls the parent remove method. The parent remove method is O(n), so this is also O(n).

**public Iterator<E> iterator();**

Calls the parent iterator method and the class involved with it. This iterator method is O(1) because all of the methods contained within are O(1), which means this iterator method is O(1).

Queue-

**public void enqueue(E obj);**

Calls the addLast method to add a node to the end of the Queue. This allows me to use the removeFirst method instead of calling the addFirst/removeLast methods in order to avoid the O(n) of the removeLast. Since addLast is O(1), this enqueue method is also O(1).

**public E dequeue();**

The dequeue method returns, and removes the object at the front of the queue, which is done by calling the parent removeFirst method. Since the removeFirst method is O(1), this method is also O(1).

**public int size();**

Same as the other size methods. This returns the value of currentSize, which makes this method O(1).

**public boolean isEmpty();**

Same as the other isEmpty methods. Just checks one variable regardless of size, which is O(1).

**public E peek();**

Calls the peekFirst method, which returns the front of the queue. This method only checks one variable, making it O(1).

**public boolean contains(E obj);**

Same as the other contains methods. Calls the parent contains method which is O(n), making this one O(n) because it has to traverse the list.

**public void makeEmpty();**

Same as the other makeEmpty methods. This method changes two variables regardless of size, which makes it O(1).

**public boolean remove(E obj);**

Calls the parent remove method. This method is O(n), which makes this method O(n) as well.

**public Iterator<E> iterator();**

Calls the parent iterator method, which is O(1).