What is the correct Java for the following:

A WindowList is a linked list of JFrames or any subclass of JFrames:

Option 1: public class WindowList<T extends JFrame> extends LinkedList<T> {

Option 2: public class WindowList extends LinkedList<JFrame> {

What are the benefits and disadvantages of each option?

Option 1 lets us create lists that we can limit to specific subclasses of JFrame, if we wish. However, we have to specify the generic for WindowList everywhere we use it.

Option 2 does not let us restrict the list to specific subclasses of JFrame. On the other hand, we do not need to specify a generic when we use it.

Sorting

We want to create a method that sorts a linked list. To do so, we must restrict the generic on the linked list to be a type that implements to Comparable interface.

<T extends Comparable<? super T>>

- we already have the OrderedLinkedList that makes this restriction.

OrderedLinkedList has an insertInOrder method. We can use that to sort by inserting each element, in order, into the list.

This technique is known as "Insertion Sort".

How many steps does it take to sort a list of n elements using Insertion Sort?

The worst case for each step is that each element goes at the end of the list.

At first, the list will have nothing in it, but each time we increase the length of the list by 1. So the worst case will be:

1 + 2 + 3 + ... + n = n(n+1)/2 = (approximately) n^2 / 2.

List size # steps needed

1,000 500,000

1,000,000 500,000,000,000

1,000,000,000 500,000,000,000,000,000

Even with a billion operations a second, this will take a long time!

How about average case? On average, the element should end up somewhere between the 1/4 and 3/4 location in the list. Let us use the middle for average,

but that still gives us approximately n^2/4, and so on average we only expect it to be twice as fast than the worst case.

Unlike with searching, most of the obvious sorting algorithms are really slow! It takes careful analysis to come up with faster algorithms.

Don't fall into the bad habit of using "easy-to-code" algorithms instead of faster algorithms.

In lecture, we ran an insertion sort on 1,000,000 items. It never completed during the lecture hour.

In that time, we coded and ran the merge sort algorithm below.

Here is a better algorithm: Merge sort

1. Split the list into two equal sized halves.

2. Sort each half

3. Merge the sorted halves together.

How many steps does it take?

1. split the list in half: n steps because we need to traverse the entire list

2. sort each half: ???? we don't know how many steps it takes to sort the whole, how can we know how many steps it takes to sort a half?

3. merge the sorted halves together: n steps if we use a careful algorithm that does not do needless looping

To determine the total number of steps needed, let T(n) be the number of steps for a list of n elements.

T(n) = n + 2 T(n/2) + n = 2 T(n/2) + 2n

A trick to determine T(n). Consider the first iteration of the algorithm, it will do 2n steps to split and merge the lists of size n.

At the second iteration, it will do 2(n/2) + 2(n/2) steps to split and merge each of the two lists of size n/2.

At the third iteration, it will do 2(n/4) + 2(n/4) + 2(n/4) + 2(n/4) steps to split and merge each of the four lists of size n/4.

Etc.

So, at each iteration, there is a total of 2n steps to split and merge the lists. The total number of steps will be 2n \* (number of iteration layers)

At each iteration, we deal with a list of half the size. So, at iteration k, the list has size n/(2^{k-1})

The number of iterations is k where n/(2^{k-1}) = 1, or k = log\_2 n + 1

So, the total number of steps needed by merge sort is T(n) = 2n(log\_2 n + 1) = 2n log\_2 n + 2n

(Remember that log\_2 1000 is approximately 3).

List size # steps needed by merge sort

1,000 2000 \* 10 + 2000 = 22,000

1,000,000 2,000,000 \* 20 + 2,000,000 = 42,000,000

1,000,000,000 2,000,000,000 \* 30 + 2,000,000,000 = 62,000,000,000

Much, much faster than the "obvious" sorting method!!!

Implementing the algorithm:

We are going to work with the nodes of the list because we need to do a lot of moving around of data.

So we will write a merge sort method that takes an unsorted list of nodes and returns a sorted list of nodes.

We then need a "wrapper" method so this works on the OrderedLinkedList.

private LLNode<T> mergeSortNodes(LLNode<T> list) {

...

}

public void mergeSort() {

setFront(mergeSortNodes(getFront()));

}

Of the three steps of the algorithm:

1. split the list into two

2. sort each half

3. merge the sorted halves together

which is the easiest to do?

Step 2! We just use the mergeSort algorithm we are writing!

private LLNode<T> mergeSortNodes(LLNode<T> list) {

LLNode<T> list1; // one half of the list

LLNode<T> list2; // the other half of the list

// 1. split each list into two

// ---- code to split the lists ----

// 2. sort each half

list1 = mergeSort(list1);

list2 = mergeSort(list2);

// 3. merge the sorted lists together

// ---- code to merge the lists ----

Now, how to split the list in two?

One idea is to think about dealing out a deck of cards.

What we will do is to go through the list and set each next pointer to skip the next node and point to the one after it.

Once this "unbraiding" is done, we will have two separate lists, one containing all the nodes at odd locations, and one containing all the nodes at even locations.

See the split method in OrderedLinkedList.

How to merge two sorted lists back into one list?

Start at the top of the each list. The smallest element will be the first element of either of these lists.

Take the node with the smaller element, remove the node from its list and add it to the new list that will be returned.

Repeat the process until one of the two lists runs out.

See the merge method in OrderedLinkedList.

Testing code:

We tested insertion sort and merge sort by trying to sort 1,000,000 random numbers:

Insertion sort:

int numToSort = 1000000;

int printRate = 1000; // used to print a status message since sort is so slow

OrderedLinkedList<Integer> list = new OrderedLinkedList<Integer>();

for (int i = 0; i < numToSort; i++) {

list.insertInOrder((int)(Math.random() \* numToSort));

if ((i + 1) % printRate == 0)

System.out.println((i+1) + " numbers sorted");

}

Merge sort:

int numToSort = 1000000;

OrderedLinkedList<Integer> list = new OrderedLinkedList<Integer>();

for (int i = 0; i < numToSort; i++) {

list.addToFront((int)(Math.random() \* numToSort));

}

list.mergeSort();

Sorting and Arrays:

Note that merge sort will not produce a fast sort if you are trying to sort on a single array. (Can you see why?)

There are other fast sort techniques that work with arrays, and you will learn about them in EECS 233.