Lab 12: Merge and Quick Sort

Introduction

In this week's lab, you'll implement two of the sorting algorithms that we learned about in lecture this week. In lecture, we focused on sorting arrays. In this lab, you'll instead focus on sorting linked lists, which requires some cleverness and a good understanding of how merge sort and quick sort operate.

All of the functions that you write will operate on the Princeton Queue Implementation, which many of you used in last week's lab and which implements a queue using a linked list. You should implement sorting using the public methods in the `Queue` class.

Merge Sort

Test driven development

In this week's lab, you'll practice test-driven-development by writing a test before writing any code. With test-driven-development, you start by writing a test that fails (because you haven't written any code yet!). After writing the relevant code, you re-run the test to make sure that it passes.

Today, you'll write a lightweight test by writing a `main` method in `MergeSort` . Your main method should create a `Queue` of unsorted objects and print that queue. Next, call `MergeSort.mergeSort()` on that queue, and print both the returned, sorted queue.

You can put any kind of object you like in your test queue, as long as the object implements the `Comparable` interface. It may work well to create a `Queue` of Strings, as in the code below:

```
Queue<String> students = new Queue<String>();
students.enqueue("Alice");
students.enqueue("Vanessa");
students.enqueue("Ethan");
```

Try running your main method. Is the output what you expect, based on the implementation we provided of `mergeSort()` ?

Sorting

If you need to review how mergesort works, you may find the Merge sort demo from lecture or this Merge sort demo to be useful.

To help you implement merge sort, start by implementing two helper methods:

- Implement `makeSingleItemQueues` . This method takes in a `Queue` called `items` , and should return a `Queue` of `Queues` that each contain one item from `items` . For example, if you called `makeSingleItemQueues` on the `Queue` `"(Alice" -> "Vanessa" -> "Ethan")` , it should return `(("Alice") -> ("Vanessa") -> ("Ethan"))` .
- Implement `mergeSortedQueues` . This method takes two sorted queues `q1` and `q2` as parameters, and returns a new queue that has all of the items in `q1` and `q2` in sorted order. For example, `mergeSortedQueues(("Alice" -> "Vanessa"), ("Ethan"))` should return `("Alice" -> "Ethan" -> "Vanessa")` . The provided `getMin` heper method may be helpful in implementing `mergeSortedQueues` . Your implementation should take time linear in the total

Θ(q1.size() + q2.size()).

Once you've finished implementing these helper methods, use them to implement `mergeSort` . With the help of the two methods above, your `mergeSort` method should be short (fewer than 15 lines of code). Run the main method you wrote above to test whether your `mergeSort` implementation works!

Quick Sort

Test driven development

As you did for merge sort, begin by writing a main method in `QuickSort.java` that creates an unsorted `Queue` , prints it, sorts it, and then prints the result.

Sorting

If you need to review how quick sort works, take a look at slides 6 through 10 from lecture. You'll be using the 3-way merge partitioning process described on slide 10. This partitioning approach, unfortunately, has no Hungarian dance demo (the dancers chose to partition based on the first item in the array, rather than on a random element.).

Begin by implementing the helper function `partition()` . The `partition()` method takes an unsorted queue called `unsorted` and an item to pivot on, and three empty queues called `less` , `equal` , and `greater` . When it returns, `less` should contain all items from `unsorted` that were less than the pivot, `equal` should contain all items from `unsorted` that were equal to the pivot, and `greater` should contain all items that were greater than the pivot.

Once you've implemented `partition()` , use it to implement the `quickSort` function. You may fund the `getRandomItem()` and `catenate()` methods that we've quickSort method should be short (fewer than 15 lines of code).

FAQ

What does the `<Item extends Comparable>` syntax mean?

In this week's lab, many of the functions have syntax that looks something like:

```
public static <Item extends Comparable> Queue<Item>
        Queue<Item> items) {
    ...
}
```

Recall from lecture 13 that if a method operates on generic types, the generic type should be defined before the return type of the method. In the example above, the part of the function declaration that says `<Item extends Comparable>` means that the `mergeSort` function operates on generic type `Item`, which must extend `Comparable` (we need `Item` to extend `Comparable` so that we can use the `compareTo` method to compare items). In other words, you can interpret the declaration above as saying "the `mergeSort` function takes a `Queue` of things that implement the `Comparable` interface, and returns a `Queue` of those things in sorted order." If you're unsure how to write code in functions like this, take a look at the helper functions that we provided, which may be helpful examples.

My code works fine but the autograder fails with some sort of JSON error.

The issue is probably that your code is quadratic time instead of linearithmic. Your code should be able to easily handle collections of 10,000 items, even if there are lots of duplicates and/or the collection is in sorted order already.

Lab 12 Navigation