Project 1a: Data Structures, version 1.0

Unlike the previous project, this project is broken into three parts. This is to keep you on the right track. Each part will be due separately. You must have the same partner (if applicable) for all three parts.

This project is brand new. Please let us know on Piazza if you spot any bugs or issues.

Introduction

In project 1, we will build implementations of a "Double Ended Queue" using both lists and arrays. We will later learn how to write our own tests for those data structures, and finally we will use those data structures to solve a small real world problem.

Project 1a is the implementation of the data structures. In this part of the project you will create exactly two Java files: `LinkedListDeque.java` and `ArrayDeque.java` , with public methods listed below.

ls, we'll say what

For this project, you are allowed to work with a partner. If you'd like to work with a partner sign up here. You may not work with the same person that you worked with on project 0. While we will not absolutely require that you have the same level of experience for this project, we still recommend it.

Getting the Skeleton Files

As with project 0, you should start by downloading the skeleton files. The directions are repeated below.

To do this, head to the folder containing your copy of your repository. For example, if your login is 'agz', then head to the 'agz' folder (or any subdirectory). If you're working with a partner, you should instead clone your partner repository, e.g. `git clone https://github.com/Berkeley-CS61B/proj1-bqd-aba`

If you're working solo, you should now be in your personal repo folder, e.g. `agz` . If you're working with a partner, your computers should both be in the `proj1-bqd-aba` folder that was created when you cloned the repo.

Now we'll make sure you have the latest copy of the skeleton files with by using `git pull skeleton master` . If you're using your partner repo, you'll also need to set the remote just like we did in lab1 using the `git remote add skeleton https://github.com/Berkeley-CS61B/skeleton-sp16.git` command.

If you find yourself faced with a strange text editor or a merge conflict, see the project 0 directions for how to proceed.

Once you've successfully merged, you should see a proj1 directory appear with files that match the skeleton repostiory.

If you get some sort of error, STOP and either figure it out by carefully reading the the git guide or seek help at OH or Piazza. You'll potentially save yourself a lot of trouble vs. guess-and-check with git commands. If you find yourself trying to use commands you Google like `force push` , don't.

The only provided file in the skeleton is LinkedListDequeTest.java. This file provides examples of how you might write tests to verify the correctness of your code. To encourage self-sufficiency in testing, we will not be putting up the autograder until 2/1. We strongly encourage you try out the given tests, as well as to write your own.

The provided tests involve a lot of custom logic (e.g. printing error messages, keeping track of whether all tests have passed, printing tests names, etc.). In part B of this project, we will write so-called JUnit tests, which will do most of this work for us. Writing tests for part A will therefore make part B easier, since you will have already thought about testing.

To use the sample test, you must uncomment the lines in the sample tests. Only uncomment a test once you have implemented all of the methods used by that test (otherwise it won't compile).

The Deque API

The double ended queue is very similar to the SList and AList classes that we've discussed in class. Specifically, any Deque implementation must have exactly the following operations:

- `public void addFirst(Item)` : Adds an item to the front of the Deque.
- `public void addLast(Item)` : Adds an item to the back of the Deque.
- `public boolean isEmpty()` : Returns true if deque is empty, false otherwise.
- `public int size()` : Returns the number of items in the Deque.
- `public void printDeque()` : Prints the items in the Deque from first to last, separated by a space.
- `public Item removeFirst()` : Removes and returns the item at the front of the Deque. If no such item exists, returns null.
- `public Item removeLast()` : Removes and returns the item at the back of the Deque. If no such item exists, returns null.
- `public Item get(int index)` : Gets the item at the given index, where 0 is the front, 1 is the next item, and so forth. If no such item exists, returns null. Must not alter the deque!

Your code must for both implementations (`LinkedListDeque.java` and `ArrayDeque.java`) must include all of these public methods, in addition to any listed below in the section for the respective implementations.

Your class should accept any generic type (not just integers). For information on creating and using generic data structures, see lecture 5. Make sure to pay close attention to the rules of thumb on the last slide about generics.

Linked List Deque

Note: We covered everything needed in lecture to do this part on Jan 27 and Jan 29.

As your first of two Deque implementations, you'll build the LinkedListDeque class, which will be linked list based. Your operations are subject to the following rules:

- add and remove operations must not involve any looping or recursion. A single such operation must take "constant time", i.e. execution time should not depend on the size of the Deque.
- get must use iteration, not recursion.
- size must take constant time.
- The amount of memory that your program uses at any given time must be proportional to the number of items. For example, if you add 10,000 items to the Deque, and then remove 9,999 items, the resulting size should be more like a deque with 1 item than 10,000. Do not maintain references to items that are no longer in the Deque.

In addition to the methods listed above, you should also include:

- `public LinkedListDeque()` : Creates an empty linked list deque.
- `public Item getRecursive(int index)` : Same as get, but uses recursion.

While this may sound simple, there are many design issues to consider, and you may find the implementation more challenging than you'd expect. Make sure to consult the lecture on doubly linked lists, particularly the slides on sentinel nodes: two sentinel topology circular sentinel topology. I prefer the circular approach. **You are not allowed to use Java's LinkedList data structure (or any data structure from java.util) in your implementation**.

Array Deque

Note: We'll have covered everything needed in lecture to do this part by Feb 1st (lecture 6)

As your second of two Deque implementations, you'll build the ArrayDeque class. This Deque must use arrays as the core data structure.

For this implementation, your operations are subject to the following rules:

- add and remove must take constant time, except during resizing operations.
- get and size must take constant time.
- The starting size of your array should be 8.
- The amount of memory that your program uses at any given time must be proportional to the number of items. For example, if you add 10,000 items to the Deque, and then remove 9,999 items, you shouldn't still be using an array of length 10,000ish. For arrays of length 16 or more, your usage factor should always be at least 25%. For smaller arrays, your usage factor can be arbitrarily low.

We strongly recommend that you treat your array as circular for this exercise. In other words, if your front pointer is at position zero, and you `addFirst` , the front pointer should loop back around to the end of the array (so the new front item in the deque will be the last item in the underlying array). This will result in far fewer headaches than non-circular approaches. See the project 1 demo slides for more details.

The signature of the constructor should be `public ArrayDeque()` . That is, you need only worry about initializing empty ArrayDeques.

Tips

- Check out the project 1 slides for some additional visually oriented tips.
- Stuck on LinkedListDeque and/or ArrayDeque? One great first step is implementing SList and/or AList. Starter code for SList and AList. For maximum efficiency, work with a friend or two or three. Solutions also available on github.
- Take things a little at a time. Writing tons of code all at once is going to lead to

misery and only misery. If you wrote too much stuff and feel overwhelmed, comment out whatever is unnecessary.

- If your first try goes badly, don’t be afraid to scrap your code and start over. The amount of code for each class isn’t actually that much (my solution is about 130 lines for each .java file, including all comments and whitespace).
- For ArrayDeque, consider not doing resizing at all until you know your code works without it. Resizing is a performance optimization (and is required for full credit).
- Work out what your data structures will look like on paper before you try implementing them in code! If you have a partner, have one partner give commands, and have the other partner draw everything out. Make sure you agree on what's happening. Try to come up with operations that might reveal problems with your implementation.
- Make sure you think carefully about what happens if the data structure goes from empty, to some non-zero size (e.g. 4 items) back down to zero again, and then back to some non-zero size. This is a common oversight.
- Sentinel nodes make life easier.
- Circular data structures make life easier for both implementations (but especially the ArrayDeque).
- Consider a helper function to do little tasks like compute array indices. For example, in my implementation of `ArrayDeque` , I wrote a function called `int minusOne(int index)` that computed the index immediately before a given index.

Frequently Asked Questions

How should I print the items in my deque when I don't know their type?

It's fine to use the default String that will be printed (this string comes from an Object's implementation of `toString()` , which we'll talk more about later this semester). For example, if you called the generic type in your class `Jumanji` , to print `Jumanji j` , you can call `System.out.print(j)`.

I can't get Java to create an array of generic objects!

Use the strange syntax we saw in February 1st's lecture, i.e. `Item[] a = (Item[]) new Object[1000];`

I tried that but I'm getting a compiler warning.

Sorry, this is something they messed up when they introduced generics into Java. There's no nice way around it. Enjoy your compiler warning. We'll talk more about this in a few weeks.

How do I make my arrows point to particular fields of a data structure. In your diagram from lecture it looked like the arrows were able to point to the middle of an array or at specific fields of a node.

Any time I drew an arrow in class that pointed at an object, the pointer was to the ENTIRE object, not a particular field of an object. In fact it is impossible for a reference to point to the fields of an object in Java.