



# Project 1A:

# LinkedListDeque

[FAQ](#)[Introduction](#)[Style](#)[Getting the Skeleton Files](#)[Deque: ADT and API](#)[LinkedListDeque](#)[Assignment Philosophy](#)[Creating the File](#)[JUnit Tests](#)[Writing and Verifying the Constructor](#)[Writing and Verifying addFirst and addLast](#)[Writing and Verifying toList](#)[The Testing Component](#)[Writing Tests](#)[Truth Assertions](#)[Example Test](#)[The Remaining Methods](#)[isEmpty and size](#)[get](#)[getRecursive](#)[removeFirst and removeLast](#)[Submit to the Autograder](#)[Scoring](#)

---

# Due: February 6th

---

## FAQ

Each assignment will have an FAQ linked at the top. You can also access it by adding “/faq” to the end of the URL. The FAQ for Project 1A is located [here](#).

---

## Introduction

In Project 0, you were a *client* of a variety of abstract data types in `java.util`. In Project 1A and 1B (and some upcoming labs), you will implement your own versions of these data structures! In Project 1, you'll begin by building your own versions of the list-like structure: implementations of a new abstract data type called a Double Ended Queue (deque, pronounced “deck”).

By the end of Project 1A, you will...

- Gain an understanding of the usage of a backing linked list in datastructures.
- Have experience with using testing and test-driven development to evaluate the correctness of your own datastructures.

For Project 1A, we will provide a significant amount of scaffolding by giving explicit instructions. In Project 1B, you'll be doing a similar task, but with much less scaffolding.

! This section assumes you have watched and fully digested the lectures up till the DLList lecture, Lecture 5.

! For this project, you must work alone! Please carefully read the **Policy on Collaboration and Cheating** to see what this means exactly.

In particular, do not look for solutions online.

⚠ It should go without saying that you may not use any of the built-in `java.util` data structures in your implementation! The whole point is to build your own versions! There are a few places where you may use specific data structures outside of tests, and we will clearly say where.

---

## Style

Starting with this project, **we will be enforcing style**. You must follow the [style guide](#), or you will be penalized on the autograder.

You can and should check your style locally with the CS 61B plugin. **We will not remove the velocity limit for failing to check style.**

---

## Getting the Skeleton Files

Follow the instructions in the [Assignment Workflow guide](#) to get the skeleton code and open it in IntelliJ. For this project, we will be working in the `proj1a` directory.

You see a `proj1a` directory appear in your repo with the following structure:

```
proj1a
├── src
│   └── Deque.java
└── tests
    ├── LinkedListDequeTest.java
    └── NodeChecker.java
```

If you get some sort of error, STOP and either figure it out by carefully reading the [git WTFs](#) or seek help at OH or Ed. You'll potentially save yourself a lot of trouble vs. guess-and-check with git commands. If you find yourself trying to use commands recommended by Google like `force push`, **don't. Don't use force push, even if a post you found on Stack Overflow says to do it!**

You can also watch Professor Hug's [demo](#) about how to get started and this [video](#) if you encounter some git issues.


---

## Deque: ADT and API


The double ended queue is very similar to the SLList and AList classes that we've discussed in class. Here is a definition from the [Java standard library](#).

A linear collection that supports element insertion and removal at both ends. The name *deque* is short for “double ended queue” and is usually pronounced “deck”. Most `Deque` implementations place no fixed limits on the number of elements they may contain, but this interface supports capacity-restricted deques as well as those with no fixed size limit.

We don't need all the methods defined in Java's `Deque`, and have defined our own interface, which can be found in `src/Deque.java`.

 **Task:** Begin by opening the `Deque.java` file and **reading** the documentation in it. We **will not** repeat information that is in the interface file in the specification – so, it is *on you* to make sure that you are reading it as you complete the project.

**You should not edit** `Deque.java`.

 **Seriously. Do not skip this. You will spend **hours** confused if you skip this step. Please save yourself the time and stress!**

 **Please.**

---

## LinkedListDeque

---

### Assignment Philosophy

A common beginner mistake is to write a large amount of code and hope that it all works once you're finished. This makes life very difficult for a programmer. Imagine implementing all the methods above, submitting to the autograder, and getting back a message that says something like "call to `get` returned 9, expected 7". You have no idea if the problem is the `get` method itself, or if some other necessary methods are broken.

To help encourage better programming habits, in Project 1A, we're going to hold your hands through the development process. You are not strictly required to follow the recommended steps, i.e. if you pass the autograder, then you get all the points, but we strongly encourage you to follow the steps outlined in this spec.

**⚠ For the intended experience, follow these steps in order. If you do something else and ask us for help, we will refer you back to these steps.**

---

## Creating the File

Start by creating a file called `LinkedListDeque`. This file should be created in the `proj1a/src` directory. To do this, right-click on the `src` directory, navigate to "New -> Java Class", and give it the name `LinkedListDeque`.

Create a `main` method in the class and add exactly this code:

```
public static void main(String[] args) {  
    Deque<Integer> lld = new LinkedListDeque<>();  
}
```

You should see a squiggly red line underneath the `<>`. To fix this, you should edit the declaration of your class so that it reads:

```
public class LinkedListDeque<T>
```

Recall from lecture that it doesn't actually matter if we use `T` or some other string like `LinkedListDeque<Glerp>`. However, we recommend using `<T>` for consistency with other Java code.

Now that we've fixed this problem, you'll see that the entire line declaring your new Deque is underlined. Press `Alt+Enter` (Windows / Linux) or `⌘ / ⌥ + Enter` (Mac) and you'll see a list of options. Select "Make 'LinkedListDeque' implement 'Deque'" and press enter. You'll see a bunch of methods appear, all of which need filling in.

If you look carefully, you'll also see that our class declaration now ends with `implements Deque<Integer>`. This is a slight mistake on IntelliJ's part. It assumed that a `LinkedListDeque` always represents a `Deque` of integers. To fix this, right-click on the word "Integer", then click "Refactor", then "Type Migration". Replace `java.lang.Integer` with `T`, then click "Refactor", and all of these erroneous usages of `Integer` will be replaced by `T`. If you picked a string other than `T` you'll need to use that instead.

Lastly, you should create an empty constructor. To do this, add the following code to your file, leaving the constructor blank for now.

```
public LinkedListDeque() {  
}
```

You can also watch [this](#) video that goes through the steps mentioned in the spec.

Note: You can also generate the constructor by clicking "Code", then "Generate" then "Constructor", though I prefer the typing the code yourself approach.

Now you're ready to get started!

---

## JUnit Tests

Now open the `LinkedListDequeTest.java` file. You'll see that every line has a `//` preceding it. Let's remove all of the `//` comments except last line. To do this, highlight all the lines of the file that start with `//`. Then click "Code" in the top menu bar, then "Comment with Line Comment". All the lines should now be uncommented. You can also use `Ctrl+/` (Windows / Linux) or `⌘ / ⌥ + /` (Mac).

Now click and run all the tests. You should fail nearly all the tests since you haven't implemented any methods yet.

Before you can pass these tests, there's a lot of work you'll need to do, so we're going to set aside the tests for now and come back to them much later.

---

## Writing and Verifying the Constructor

**i** This section assumes you have watched and fully digested the lectures up to **and including** the `DLList` lecture, Lecture 5.

Start by selecting a “topology,” or structure that you'd like to represent the empty list. The possible choices discussed in lecture are:


- The empty list is represented by a null value. [See this slide.](#)
- The empty list is represented by two sentinel nodes that point at each other. The first sentinel node is pointed to by a variable called `first`, and the last sentinel node is pointed to by a variable called `last`. [See this slide.](#)
- **STRONGLY RECOMMENDED:** The empty list is represented by a single sentinel node that points at itself. There is a single instance variable called `sentinel` that points at this sentinel. [See this slide.](#)


As mentioned in lecture, though this last approach seems the most complicated at first, it will ultimately lead to the simplest implementation.

Implement the constructor for `LinkedListDeque` to match your chosen topology. Along the way you'll need to create a `Node` class and introduce one or more instance variables. This may take you some time to understand fully. Your `LinkedListDeque` constructor **must** take 0 arguments.

You're welcome to pick whichever choice you'd like, but it **must** “look like” a `DLList`. That is, the nodes should be doubly linked, and have exactly the necessary fields for a doubly linked node. Additionally, you should only have one node class, and this node class **must** be an inner, or nested class inside `LinkedListDeque`.

When you're done, set a breakpoint on the line where you create a `LinkedListDeque` in the `main` method. Run your program in debug mode,

and use the Step Over () feature. Use the Java Visualizer to verify that your created object matches the topology you chose.

 **Task:** Pick a doubly-linked list topology, and implement the constructor.

---

If the test `noNonTrivialFields` fails, your `Node` class is **insufficient** in some way:

- It might be defined in a separate file.
- It might be using an incorrect type to store data. Remember that `Deque` is *generic*.
- It might have a constructor that takes additional arguments.
- It might have too few or too many fields for a doubly-linked node.

---

The tests will not work until you complete `toList`.

---

## Writing and Verifying `addFirst` and `addLast`

`addFirst` and `addLast` **may not** use looping or recursion. A single add operation must take "constant time," that is, adding an element should take approximately the same amount of time no matter how large the deque is. This means that you cannot use loops that iterate through all / most elements of the deque.

Now fill in the `addFirst` method. Add some `addFirst` calls to your `main` method and use the debugger and visualizer to verify that your code is working correctly.

Fill in the `addLast` method, and again add some `addLast` calls and use the visualizer to verify correctness.

 **Task:** Implement `addFirst` and `addLast`, and verify that they are correct using `main` and the Java visualizer.

---

The tests will not work until you complete the next section, `toList`.



## Writing and Verifying toList

You may have found it somewhat tedious and unpleasant to use the debugger and visualizer to verify the correctness of your `addFirst` and `addLast` methods. There is also the problem that such manual verification becomes stale as soon as you change your code. Imagine that you made some minor but uncertain change to `addLast`. To verify that you didn't break anything you'd have to go back and do that whole process again. Yuck.

(Also, we have just under 1700 students! No way we're doing that to grade everyone's work.)


What we really want are some automated tests. But unfortunately there's no easy way to verify correctness of `addFirst` and `addLast` if those are the only two methods we've implemented. That is, there's currently no way to iterate over our list and get back its values and see that they are correct.

That's where the `toList` method comes in. When called, this method returns a `List` representation of the `Deque`. For example, if the `Deque` has had `addLast(5)`, `addLast(9)`, `addLast(10)`, then `addFirst(3)` called on it, then the result of `toList()` should be a `List` with 3 at the front, then 5, then 9, then 10. If printed in Java, it'd show up as `[3, 5, 9, 10]`.

Write the `toList` method. The first line of the method should be something like `List<T> returnList = new ArrayList<>()`. **This is one location where you are allowed to use a Java data structure.**

To verify that your `toList` method is working correctly, you can run the tests in `LinkedListDequeTest`. If you pass all the tests, you've established a firm foundation upon which to continue working on the project. Woo! If not, use the debugger and carefully investigate to see what's wrong with your `toList` method. If you get really stuck, go back and verify that your `addFirst` and `addLast` methods are working.

❗ Some later methods might seem easy if you use `toList`. **You may not call `toList` inside `LinkedListDeque`**; there is a test that checks for this.

 **Task:** Implement `toList`, and verify that it is correct with the tests in `LinkedListDequeTest`.

---

## The Testing Component

In Project 0, we gave you a full suite of unit tests that you could use to test your code locally. In this project, you'll be required to write *your own* unit test suite that provides similar coverage. To give a bit of insight about how this works, we will essentially be taking your test file

(`LinkedListDequeTest.java`) and using it to “test” our staff solution of `LinkedListDeque`. Using some autograder magic, we're able to determine which edge cases your tests are able to hit, thus telling us the “coverage” of your test suite. So, in order to get a full score on this component, you should try to think of any and all corner cases for each of the methods!

Our staff solution uses a circular sentinel topology. Our staff solution also only has a constructor that takes 0 arguments, which means that your tests should only use a constructor that takes 0 arguments.

---

## Writing Tests

To write tests, we will use Google's [Truth](#) assertions library. We use this library over JUnit assertions for the following reasons:


- Better failure messages for lists.
- Easier to read and write tests.
- Larger assertions library out of the box.


We often write tests using the Arrange-Act-Assert pattern:

1. **Arrange** the test case, such as instantiating the data structure or filling it with elements.
2. **Act** by performing the behavior you want to test.
3. **Assert** the result of the action in (2).

We will often have multiple “act” and “assert” steps in a single test method to reduce the amount of boilerplate (repeated) code.

You should write your tests in `LinkedListDequeTest.java`.

 **Note:** The tests that you write in this project will be checked for the different “scenarios” they cover. You will need to cover sufficiently many scenarios, including a few edge cases.

 **Warning:** While the coverage checker can check how much you *do* to the deque, it doesn’t check what you *assert* about the deque. If you find yourself failing autograder tests that you think you have coverage for, a good next step is to add additional assertions to your own tests. Examples include verifying the result of every call, checking the entire deque between every call, or checking the results of other deque methods.

---

## Truth Assertions

A Truth assertion takes the following format:

```
assertThat(actual).isEqualTo(expected);
```

To add a message to the assertion, we can instead use:

```
assertWithMessage("actual is not expected")
    .that(actual)
    .isEqualTo(expected);
```

We can use things other than `isEqualTo`, depending on the type of `actual`. For example, if `actual` is a `List`, we could do the following to check its contents without constructing a new `List`:

```
assertThat(actualList)
    .containsExactly(0, 1, 2, 3)
    .inOrder();
```

If we had a `List` or other reference object, we could use:

```
assertThat(actualList)
    .containsExactlyElementsIn(expected) // `expected` is a List
```

```
.inOrder();
```

Truth has many assertions, including `isNull` and `isNotNull`; and `isTrue` and `isFalse` for `boolean`s. IntelliJ's autocomplete will often give you suggestions for which assertion you can use.

---

## Example Test

Let's break down the provided `addLastTestBasic`:

```
@Test
```

```
/** In this test, we use only one assertThat statement. IMO this te
 * In other words, the tedious work of adding the extra assertThat
public void addLastTestBasic() {
    Deque<String> lld1 = new LinkedListDeque<>();

    lld1.addLast("front"); // after this call we expect: ["front"]
    lld1.addLast("middle"); // after this call we expect: ["front",
    lld1.addLast("back"); // after this call we expect: ["front", "
    assertThat(lld1.toList()).containsExactly("front", "middle", "k
}
```

- `@Test` tells Java that this method is a *test*, and should be run when we run tests.
  - **Arrange:** We construct a new `Deque`, and add 3 elements to it using `addLast`.
  - **Act:** We call `toList` on `Deque`. This implicitly depends on the earlier `addLast` calls.
  - **Assert:** We use a Truth assertion to check that the `toList` contains specific elements in a specific order.
- 

## The Remaining Methods

All that's left is to test and implement all the remaining methods. For the rest of this project, we'll describe our suggested steps at a high level. We

**strongly encourage** you to follow the remaining steps in the order given. In particular, **write tests before you implement**. This is called “test-driven development,” and helps ensure that you know what your methods are supposed to do before you do them.

---


## isEmpty and size

These two methods must take **constant time**. That is, the time it takes for either method to finish execution should not depend on how many elements are in the deque.

Write one or more tests for `isEmpty` and `size`. Run them and verify that they fail. **Your test(s) should verify more than one interesting case**, such as checking both an empty and a nonempty list, or checking that the size changes.

For these tests, you can use the `isTrue` or `isFalse` methods on your `assertThat` statements.

Your tests can range from very fine-grained, e.g. `testIsEmpty`, `testSizeZero`, `testSizeOne` to very coarse grained, e.g. `testSizeAndIsEmpty`. It's up to you to explore and find what granularity you prefer.


 **Task: Write tests** for the `isEmpty` and `size` methods, and check that they fail. Then, implement the methods.

---

## get

Write a test for the `get` method. Make sure to test the cases where `get` receives an invalid argument, e.g. `get(28723)` when the `Deque` only has 1 item, or a negative index. In these cases `get` should return `null`. You should disregard the skeleton code comments for `Deque.java` and take spec as your primary point.

`get` must use iteration.

 **Task: After you've written tests and verified that they fail,** implement `get`.

## getRecursive

Since we're working with a linked list, it is interesting to write a recursive get method, `getRecursive`.

Copy and paste your tests for the `get` method so that they are the same except they call `getRecursive`. (While there is a way to avoid having copy and pasted tests, though the syntax is not worth introducing – passing around functions in Java is a bit messy.)

 **Task: After you've copy-pasted tests and verified that they fail, implement `getRecursive`.**


---

## removeFirst and removeLast

Lastly, write some tests that test the behavior of `removeFirst` and `removeLast`, and again ensure that the tests fail. **For these tests you'll want to use `toList`!** Use `addFirstAndAddLastTest` as a guide.

Do not maintain references to items that are no longer in the deque. 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 memory usage should amount to a deque with 1 item, and not 10,000. Remember that the Java garbage collector will “delete” things for us if and only if there are no pointers to that object.

If `Deque` is empty, removing should return `null`.

 **Task: After you've written tests and verified that they fail, implement `removeFirst` and `removeLast`.**

---

## Submit to the Autograder

Once you've written local tests and passed them, try submitting to the autograder. You may or may not pass everything.

- If you fail any of the coverage tests, it means that there is a case that your local tests did not cover. The autograder test name and the test coverage component will give you hints towards the missing case.
  - If you fail any of the timing tests, it means that your implementation does not meet the timing constraints described above.
- 

## Scoring

This project, similar to Project 0, is divided into individual components, each of which you must implement *completely correctly* to receive credit.

1. **Empty list (5%)**: Define a valid `Node` class and correctly implement the constructor.
2. **Adding (25%)**: Correctly implement `addFirst`, `addLast`, and `toList`.
3. **`isEmpty`, `size` (5%)**: Correctly implement `isEmpty` and `size` with add methods working.
4. **`get` (10%)**: Correctly implement `get`.
5. **`getRecursive` (5%)**: Correctly implement `getRecursive`.
6. **Removing (40%)**: Correctly implement `removeFirst` and `removeLast`.

Additionally, there is a **test coverage (10%)** component. We will run your tests against a staff solution, and check how many scenarios and edge cases are tested. You can receive partial credit for this component.

*Last built: 2023-10-26 18:40 UTC*