

This lab has **five (5)** checkpoints.

Learning Outcomes

Upon completing this lab, students should be able to

- **Trace** the *heapsort* algorithm.
- **Translate** a heap between a tree view and an *array* view.
- Implement to a description
- Analyze debug output to fix code

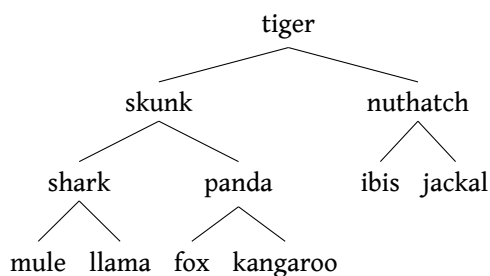
Introduction

1. Consider a *max-heap* data structure containing `String` objects ordered lexicographically but stored in a `String[]`.

```
private void bubbleUp(int index) {
    if (index > 0) {
        if (storage[index].compareTo(storage[parent(index)]) > 0) {
            swap(index, parent(index));
            bubbleUp(parent(index));
        }
    }
}

private void bubbleDown(int index) {
    if (left(index) != NULL) { // at least one
        int iMaximum = left(index);
        if (right(index) != NULL) {
            int iRight = right(index);
            iMaximum = (storage[iMaximum].compareTo(storage[iRight]) > 0)? iMaximum : iRight;
        }
        if (storage[iMaximum].compareTo(storage[index]) > 0) {
            swap(index, iMaximum);
            bubbleDown(iMaximum);
        }
    }
}
```

- (a) Translate the following *tree* view of a heap into its corresponding *array* view:



- (b) After two steps of *heapsort* where the root of the heap is swapped with the element in the furthest node still in the heap, heap size is reduced, and the root is bubbled-down to reestablish the heap property, the *array* view is (shaded cells are already in sorted order):

0	1	2	3	4	5	6	7	8	9	10
shark	panda	nuthatch	mule	fox	ibis	jackal	kangaroo	llama	skunk	tiger

Convert the *heap* at this point into a tree view.

- (c) **After** swapping the correct element into array index 8 but **before** running `bubbleDown`, what is the *array* view of the heap **and** sorted elements. Be sure to indicate the line between the heap and the sorted portions of the array.
- (d) `bubbleDown` will be called with the **index** number of the node that *might* be out of order; it will also have access to the array holding the heap/sorted entries. The method will be called *recursively* until the heap property is restored.

Give the **index** parameter and the *tree* representation of the heap portion of the array as `bubbleDown` starts while restoring the heap property from the point you captured in the previous question. Notice that the first heap value is just what you answered there.

✓ Show the lab instructor your drawings of the data structures.

2. Using `println` for debugging is a crude way to check how your program is running *but* it can be very effective at identifying problems. One cost of debugging this way is the cost of adding/removing (or commenting out) debug information. This can be most annoying when you *think* a bug is fixed and strip the debugging, only to find the problem again.

Another issue is when the bug happens on the client's machine. You could recompile, with the debugging data in place, ship the instrumented executable, and have them capture the output. Necessary when you cannot recreate the error locally, perhaps, but how do you know whether you added in the correct debugging output?

Another way to deal with this is to build a logging object: an object that is created at the beginning of the program that supports an interface like `println` but also tracks a `boolean` on/off switch or some sort of "debug level". If the level can be set from the command-line, then setting it to `NONE` by default means the logger produces no output. Until you run the program with the appropriate command-line switch.

This lab will have you build `logger.Logger` with the ability to `print/println` Strings as *errors*, *warnings*, or *information*. If the importance of the message meets or exceeds the level of messages to be logged, the value is printed; if not, no input is generated.

Create a new `gradle` project:

- application
- Java
- No multiple subprojects
- Groovy
- JUnit Jupiter
- Project: logger
- Java v19
- No new APIs

Create two *packages*: `driver` for the main program and `logger` where a debug logger will be created.

Modify `build.gradle`:

Test-logger Add test-logger plugin:

```
plugins {
    id 'application'
    id 'com.adarshr.test-logger' version '4.0.0'
}
```

```

Print during testing tasks.named('test') {
    useJUnitPlatform()
    testLogging.showStandardStreams = true
}

```

Rename executable Use driver.LoggerDriver

Obviously: move your application code to driver/LoggerDriver.java.

Go into src/test/main/java. The following directions are for saving the shell of logger/AppTest.java (the import lines). You can handle this however you want.

Rename AppTest.java to LoggerTest.java. Open the file and update class name and (if necessary) package name. Rename the function to TestGetInstance and get rid of the body of the function (it will be empty for the moment).

While you probably should test the *driver* as well as the new class, this lab will only have you test logger.Logger.

Build the project, fixing typos until it compiles.

Show your newly working (but doing nothing) code to the lab instructor.

✓ Show the lab instructor your restructured code.

- How to make **global variables** in Java. Sharing a single instance of an object across a whole project leads to a global object like System.out, or constant *dependency injection* where the object becomes a parameter across every level of the program, like the Random object passed into almost every function in pMeleeI.

Generally, globally-shared objects are frowned upon but sometimes the price (all of your program depends on a single object so changes in it can require changes across the whole thing) must be paid.

The object logger.Logger will be an example of the **singleton** design pattern: there will be a static instance of Logger but no public constructors. To get **the** logger, call the static function Logger.getInstance() (it returns the one Logger).

Logger has the static field, instance:

```
private static Logger instance = null;
```

When getInstance is called, it constructs a new Logger (w/ a private default constructor) if the field is null, setting instance. It then returns instance to its caller. All future calls will just return instance because it is now not null.

Write getInstance and write a test for it:

```

@Test
void testGetInstance() {
    Logger logger = Logger.getInstance();
    assertNotNull(logger);
}

```

Make the test pass. Notice that Logger does not yet have any fields.

✓ Show the lab instructor your working code.

- Logger will have a couple of print/println functions that take a message (String) and a *debug level* (int) and print out the string if the debug level is high enough.

Stop for a minute and consider what **state** (fields) Logger needs to keep track of for handling this.

A Logger needs a *current debug level* and someplace to print the message. We will use PrintStream (the type of System.out) to permit the use of println.

```

private PrintStream out;
private int debugLevel;

public PrintStream getOut() {
    return out;
}
public void setOut(PrintStream out) {
    this.out = out;
}
public int getDebugLevel() {
    return debugLevel;
}
public void setDebugLevel(int debugLevel) {
    this.debugLevel = debugLevel;
}

```

Add tests for getting/setting the fields. Try making a new output stream that writes to buffer of characters:

```
PrintStream expectedNewOutputStream = new PrintStream(new ByteArrayOutputStream());
```

Now a private, **primary** constructor must be written. It is **private** so that `getInstance` is the only way to get a `Logger` from outside the class.

The constructor will take two parameters, `out` and `debugLevel`. `getInstance` should call this constructor with `System.err` and 0. A level of 0 is effectively “none”.

✓ Show the lab instructor your working code.

5. Finally, the print routines. Nothing is printed if `debugLevel` is less than 1.

```

// if messageLevel >= debugLevel, println the message
public void println(int messageLevel, String message);
// if messageLevel >= debugLevel, println (start new line)
public void println(int messageLevel);
// if messageLevel >= debugLevel, print the message
public void print(int messageLevel, String message);

// println message with "Info : " prepended, level 1
public void info(String message);
// println message with "Warning : " prepended, level 2
public void warning(String message);
// println message with "Error : " prepended, level 3
public void error(String message);

```

To test these, you can use the `ByteArrayOutputStream` type like this:

```

@Test
void testPrintlnLevel2() {
    Logger logger = Logger.getInstance();
    ByteArrayOutputStream debugOut = new ByteArrayOutputStream();
    PrintStream otherOut = new PrintStream(debugOut);
    logger.setOut(otherOut);
    logger.setDebugLevel(2);
    logger.println(1, "One Fish");
    logger.println(2, "Two Fish");
    logger.println(3, "Three Fish");
}

```

```
String expected = ""
    Two Fish
    Three Fish
    "";
String actual = debugOut.toString();
assertEquals(expected, actual);
}
```

Notice that `debugOut` is named so that we can use the `toString` function to get whatever has been printed to it returned in a `String`.

You can use this method as a model for testing different print routines with different debug levels. Make sure to test a zero debug level and have separate test routines for `info`, `warning`, and `error`.

✓ Show the lab instructor your working code testing `debugLevel` zero and the three named print routines with multiple debug levels.

6. [Not a separate checkpoint but necessary] Add and commit your full `gradle` project (if you `git add` just the root directory, `..`, `git` will add all subdirectories recursively).

Create a repository in your Gitea turn-in organization, associate it with your lab work, and push the repo. This will be done at the end of every lab.