**Task Scheduler Application**

Zackary Finer

CS 146 Section 07, Prof. Wu

March 29, 2018

Table of Contents

[Application Use 2](#_Toc510083759)

[Application Functionality and Design 6](#_Toc510083760)

[Overview 6](#_Toc510083761)

[Class and Function Descriptions 7](#_Toc510083762)

[Class TaskHeap 7](#_Toc510083763)

[Class TaskScheduler 9](#_Toc510083764)

[Class Task 9](#_Toc510083765)

[Class SchedulerGuiMain 9](#_Toc510083766)

[Class HeapTreeFrame/HeapTreeDrawer 12](#_Toc510083767)

[Class ArrayDisplayDialog 15](#_Toc510083768)

[Challenges and Solutions 16](#_Toc510083769)

[Indexing 16](#_Toc510083770)

[Dynamic Management of the Task List 17](#_Toc510083771)

[Displaying the Heap Tree 18](#_Toc510083772)

[Testing 19](#_Toc510083773)

[Heap 19](#_Toc510083774)

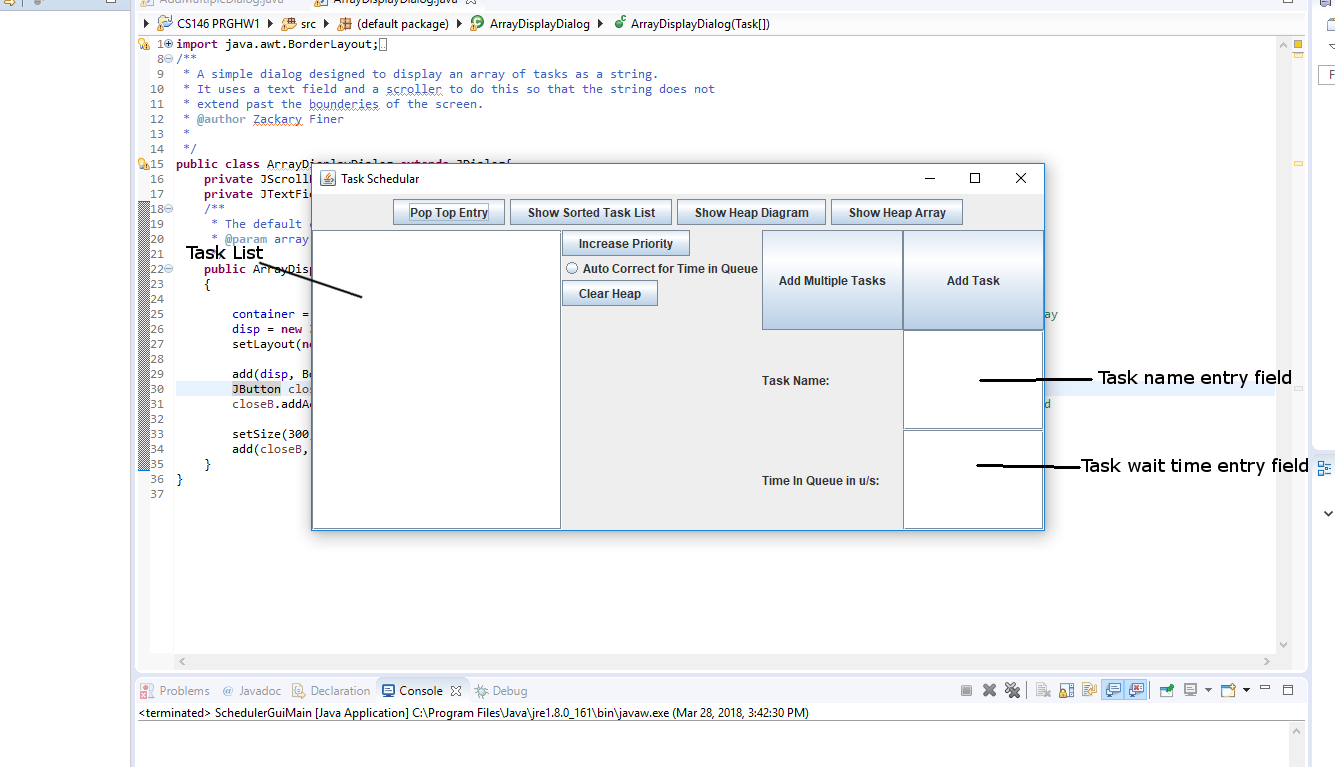
[GUI Tests 27](#_Toc510083775)

[Conclusion – Lessons Learned 33](#_Toc510083776)

Application Use

Installation

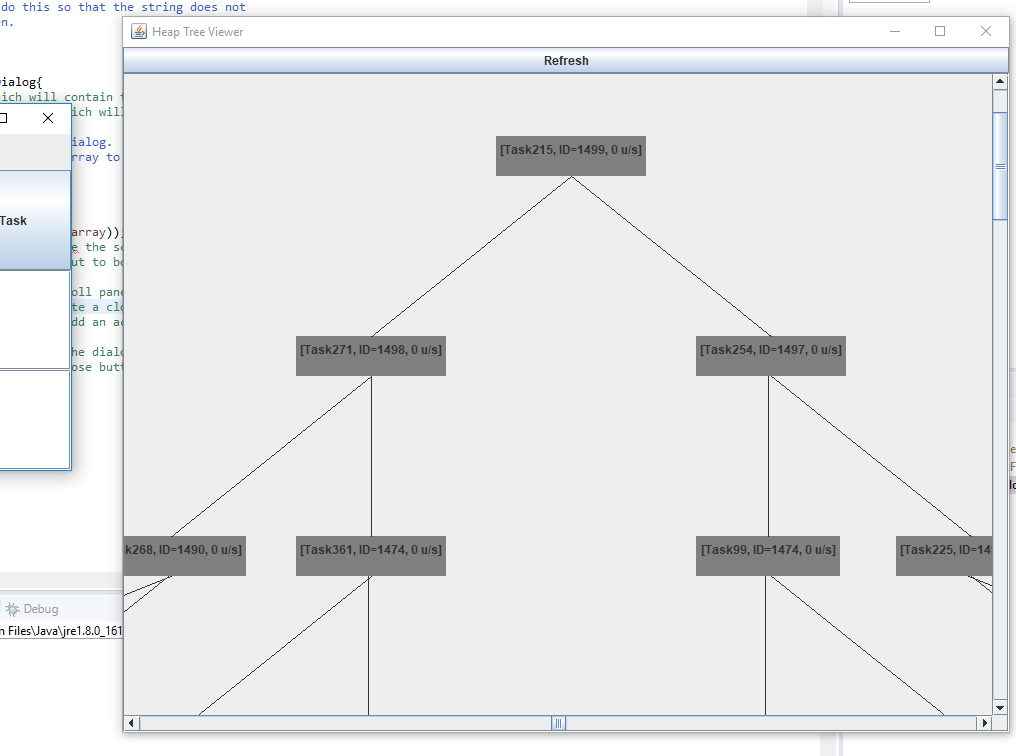
1. To run the application, simply double click the TaskSchedular,jar file located in the same directory you found this report document in.
2. To view and/or run the source code, navigate to the src folder located within the same directory you found this report document in. This directory contains all source files (\*.java) which were used to compile the executable jar. These files also contain java doc and comments for all methods used.
   1. SchedulerGuiMain is the main entry point of the application
   2. All files except HeapAndApplicationUnitTests.java are crucial to the execution of the application, if you are using an IDE like eclipse or net beans, you will need to import all other java files specified in order to compile and run.

Operation

Adding Tasks:

1. Enter a name into the task name entry field for the task which you would like to add.
2. (Optional) Enter the amount of time the task you would like to insert has been in queue. This must be an integer. If it is not formatted as such, an error message will show.
3. Press the Add Task Button directly above the Task name entry field.

Showing the Heap Tree Diagram



(Above): An image of the heap tree viewer window

1. Press the Show Heap Diagram window button.
2. A window should open with a heap diagram of the priority queue

Showing the Heap Array

1. Press the Show Heap Array button.
2. A dialog will open with a text field which can be scrolled containing the array of the heap.

Clearing the Heap

1. Press the Clear button located to the right of the task list.
2. All entries in the heap will be removed, the Task List should reflect this. If the Heap Tree View window is open, it will close.

Removing the Top Entry

1. Press the Pop Top Entry button located directly above the Task List.
2. A dialog will open showing the entry which was removed, the task list and heap tree view will redraw to reflect the changes.

Showing Sorted Task List

1. Press the Show Sorted Task List button located above and to the right of the task list.
2. A dialog will open which contains a scrollable text field of all the tasks in order of priority from least to greatest

Increasing the Priority of a Task

1. Select the task which you would like to increase the priority of in the task list.
2. Press the Increase Priority button to the right of the task list. This will increase the priority ID of the selected task by 100. Repeat this until the task has reached the desired priority.

Adding Multiple Entries

1. Press the add multiple entries button located to the left of the Add Task button.
2. In the dialog, enter the desired number of tasks to add, the name assigned to each task (i.e., If the name is DoStuff, each task will be DoStuff1,DoStuff2, … DoStuff#)
3. In the dialog, select the delay setting you would like when generating the tasks.
   1. If you want each task to be assigned a random wait time, select Random.
   2. If you want every task to be assigned a wait time of your choosing, select Custom.
   3. If you want every task to be assigned 0 for the wait time, select none.
4. Once you are done entering this information, press the Ok button to complete the operation.
5. The task list and Heap Viewer window will redraw to reflect the changes.

Automatically Sifting Entries Base on Wait Time

1. Press the Auto Correct for Time in Queue Radio Button located between the increase priority and clear buttons.
2. Select Yes in the dialog which appears.
3. You can look at the Heap Tree Viewer to see the changes.
4. Click the Auto Correct for Time in Queue Radio Button again to add more entries to the heap.

Application Functionality and Design

Overview

At the core of the application is a heap data structure. This heap data structure is responsible for organizing all tasks in the queue such that the maximum task is easily accessible. The class TaskHeap contains the implementation of this heap structure. Implementing this heap was done using pseudo code provided in the book Introduction to Algorithms 3rd edition, specifically in chapter 6. However, certain modifications were necessary in order to successfully implement these methods in java (see Challenges and Solutions section).

It should also be noted that this TaskHeap structure operates on a buffer of Tasks as opposed to a buffer of integers. The class Task was designed to hold state information on the Tasks entered into the priority queue. Task objects hold a String representing the title of the task, an integer representing the amount of time the task has been in queue, and an integer representing the Task’s priority ID. The priority ID is the key by which tasks are prioritized in the queue: a task with a higher priority ID will be removed from the queue before tasks with a lower priority ID are. Most methods in the TaskHeap class only access this priority ID member variable in tasks.

This heap structure is then surrounded by a TaskScheduler class which serves as a wrapper for the main application. TaskScheduler contains a TaskHeap object, as well as a more limited set of methods than TaskHeap for editing the heap. This was done to simplify the interface of the heap and reduce the chances of data corruption. TaskScheduler also contains more specialized methods, such as one which sifts up entries based on their wait time in queue.

The rest of the classes are dedicated primarily to GUI. The GUI was implemented using Java’s Swing library. There are 2 main windows of the GUI: Task-Scheduler window (SchedulerGuiMain) which is used for editing the priority queue of tasks, and the Heap-Tree Diagram window (HeapTreeFrame) which displays a visual representation of the binary tree structure of the TaskHeap. There are also two JDialog classes. One (AddMultipleFrame) is used for gathering information from the user during the add multiple tasks process, and the other (ArrayDisplayDialog) is used for displaying the raw text form of the Task Queue and sorted Task list.

The Task-Scheduler window is ultimately the entry point of the application. It contains is responsible for managing a buffered priority queue, and for providing a UI to access the priority queue. There are two storage components in the Task-Scheduler window class. One is a DefaultListModel which contains copy of all elements currently allocated in the heap and is used by a JList to display entries, so the user can edit them. The other is the task buffer, which is a task array with a maximum size of 512 elements. The task buffer holds all elements in the priority queue and is the array which all heap methods operate on. As such, the allocated contents of the task buffer are the array representation of the priorities queue’s binary task heap. The task buffer is also what the Heap-Tree Diagram window reads from in order to display the tree.

Class and Function Descriptions

Class TaskHeap

The TaskHeap class contains all the methods used for creating and editing a binary heap data structure. The implementations of these methods are based on pseudocode provided in the book Introduction to Algorithms 3rd edition.

Member Variables:

**int heapSize**: This variable holds the size of the heap, or the number of elements currently allocated in the heap data structure.

Methods:

**exchange(HeapBufferArray, firstIndex, secondIndex)**: This method is used for exchange two entries in the heap with each other. Its implementation is relatively simple; it creates a temporary variable to hold the first element, assigns the first element to the second, and then assigns the second element to the temporary variable.

**int getHeapSize()**: A gettter for the heapSize variable

**int parent(index)**: This method is used to return the index of the parent node of a given entry in the heap at the specified index. It returns the index of the parent node, and if the node has no parent then it returns -1 for debug purposes.

**int left(index)**: Similar to the parent method above, this method returns the index of the left child of a given node at the specified index.

**int right(index)**: Similar to parent and left, this method returns the index of the right child of a given node at the specified index.

**Task heapMax(HeapBufferArray)**: Returns the element with the greatest priority in the heap but does not remove it from the heap; it is similar to the peak method on a stack.

**maxHeapify(HeapBufferArray, index)**: Performs the max-heapify operation for a subheap with the root specified at the index; it will sift down the entry at that index if the subheap does not obey the heap property, and repeat this operation for each subheap contained in in the initial subheap.

**buildMaxHeap(HeapBufferArray)**: Builds a max heap in the specified array. This is accomplished by repeatedly calling the max-heapify operation on all subheaps contained in the array: it starts from the leaves of the tree and works its way up.

**buildMaxHeap(HeapBufferArray, heapSize)**: Same as the method above, except it is designed to operate on a buffer, or an array which contains empty “cells”. Because of this, heapSize must be specified to avoid null pointer exceptions.

**heapExtractMax(HeapBufferArray)**: Returns the maximum or highest task in the heap, and then removes that entry from the heap and replaces it with second highest element in the heap. This is accomplished through the use of the max-heapify method. This method is similar to the pop method for a stack.

**heapIncreaseKey(HeapBufferArray, index, Task)**: Increases the priority of a task in the heap at the specified index to that of the passed Task argument. After doing this, it sifts that entry up in the heap until it has reached the correct position. If the passed Task has a priority ID less than the Task currently at the specified index, then the method will display an error and do nothing.

**maxHeapInsert(HeapBufferArray, Task)**: Inserts a new Task into the heap. This is done by inserting the new task at the bottom of the heap with the minimum priority id possible (to ensure that the key is increased). It then uses the heapIncreaseKey method using the passed task to sift the new entry up to the correct position.

**heapSort(HeapBufferArray)**: Sorts the entries in the specified heap buffer from least to greatest using heap Sort.

**heapSort(HeapBufferArray, heapSize)**: Same as above, except it is designed to operate on a buffer instead of a fully allocated array.

**boolean testHeapProperty(HeapBufferArray, heapSize)**: Used for testing whether a given heap is in proper order: all child entries are less than their parent entity. I decided to use recursion to do this instead of a while loop, because it seemed more intuitive to me at the time.

**boolean testSub(HeapBufferArray, heapSize, index)**: This is the recursive method used by the testHeapProperty to ensure the heap is in proper order. It recursively access all entries in the heap and checks whether they are less than their parent entry. If at any point it finds that this isn’t true, it exits and returns false.

Class TaskScheduler

The TaskScheduler class was designed to surround the TaskHeap class with an interface which is more simplified and specific to the priority queue application. It contains a more limited set of methods for modifying the heap, and hides methods like parent, left, and exchange from the user.

Constants:

**int TASK\_INCRIMENT\_WAITTIME**: This is a constant which represents the amount of time a task can be in the queue before it’s priority is increased; it is only ever used when increaseForWaitTime is true.

**int PRIORITY\_INCREASE\_AMOUNT**: The amount which is added to a tasks priority ID if it has been in queue for the specified amount of time.

Member Variables:

**TaskHeap priorityQueue**: The TaskHeap object that this TaskSchedular will use to build and maintain the heap of the priority queue.

**boolean increaseForWaitTime**: A boolean which represents whether or not this TaskScheduler should compensate for entries with a higher wait time when it is building the queue, it is set to false by default.

Methods:

**buildTaskScheduler(HeapBufferArray)**: Builds a priority queue in the passed heap buffer. It uses the buildMaxHeap method in TaskHeap to do this. If increaseForWaitTime is true, it also calls increase for wait time on the specified array.

**buildTaskScheduler(HeapBufferArray, heapSize)**: Same as above, except it is designed to operate on a buffer. It uses the corresponding buildMaxHeap method in TaskHeap.

**addTask(HeapBufferArray, Task)**: Adds a task to the priority queue, uses the TaskHeap addTask method. if increaseForWaitTime is true then it performs the increase for wait time operation.

**increaseTaskPriority(HeapBufferArray, index, Task)**: Increases the priority of a task at the specified index to that of the passed Task. It does this using the TaskHeap heapIncreaseKey method.

**Task popTopTask(HeapBufferArray)**: returns and removes the top entry in the priority queue. Uses the heapExtractMax method in TaskHeap.

**increaseForWaitTime(boolean)**: Not to be confused with the variable increaseForWaitTime, this method is responsible for sifting up any entries which have been in the queue for an extended period of time. It does this by increasing key’s priority ID by the amount specified in the constant PRIORITY\_INCREASE\_AMMOUNT\*(the time it’s been in queue)/TASK\_INCRIMENT\_WAITTIME.

Class Task

The Task class was designed to hold state information on Tasks currently being held in the Task priority que. It’s methods primarily just return the values of it’s member variables, aside from toString and equals.

Member Variables:

**int priorityID**: Holds the priority ID of this task, which will be used in the heap methods for comparisons.

**int timeInQueue**: Holds the amount of time this task has been in the queue. This entry will be processed by the increaseForWaitTime method in TaskScheduler.

**String title**: Holds the title or name of this task, this is to provide the user with some way to track an entries existence through the heap.

Methods:

**Task(priorityID, timeInQueue, title)**: The constructor for a Task object. It takes the arguments specified and assigns them to their corresponding member variable in the objects state.

**int getPriority()**: Simple getter, returns priorityID.

**int getTimeInQueue()**: Simple getter, returns timeInQueue.

**String getTitle()**: Simple getter, returns title.

**boolean equals(Other)**: Returns whether or not this Task is equal to another. This is accomplished by comparing the state information of this task to that of the other – all member variables are taken into account.

**String toString()**: Since this method is used a lot by the GUI and test methods, it differs from the getTitle() method. It returns a string representation of this task in the form [title, ID=priorityID, timeInQueue u/s].

Class SchedulerGuiMain

SchedularGuiMain is one of the largest classes in the project: it is the entry point of the application and the UI. There will only ever be one instance of this class active during the execution of the application. For the sake of readability, I’ve omitted some of the member variables from this description which are not crucial to the understanding of this class. For more detailed descriptions of these member variables, please refer to the comments in the source code.

Constants:

**int BUFFER\_SIZE:** The size of the buffer to be allocated, I’ve set this to 512 tasks.

**int MAX\_PRIORITY**: The highest priority a task in the tree can have, this is used primarily by the random priority ID generation system.

Member Variables:

**<Various Swing components>**: These are member variables which hold references to all the UI elements (such as buttons and textfields) to be displayed in the frame of the window, so that they may be accessed and modified throughout the execution of the application.

**DefaultListModel<Task> taskListModel**: A list model which will contain a copy of all tasks currently entered into the array. It is used by other methods in the class as well as a JList element which will display entries to the user for editing.

**Task[] buffer**: A buffer of tasks which will hold the priorityQue which the user is editing. This is the main storage area of the application and will hold all tasks which the user enters.

**int allocated**: This variable stores the total number of entries allocated in the buffer, it is incremented and decremented whenever an entry is added or removed and is used as primarily by the Show Tree Diagram class to determine how many elements to show the user.

**TaskScheduler taskPriorityQueue**: The TaskScheduler object which will maintain and create the priority queue which the user will be able to view and edit.

**Random rng**: A random number generator which will be used to generate random priority IDs for Tasks which the user enters.

**HeapTreeFrame treeDisplay**: A reference to a HeapTreeFrame object that will be used to display the visual representation of the

Methods:

**ShchedulerGuiMain()**: The default constructor for the class. It initializes all member variables, including UI elements and the buffer for the tasks. It also assigns the appropriate action listeners to buttons.

**hideOptions(boolean)**: Hides GUI buttons associated with editing the heap. This is a necessary operation if autoSift is currently enabled.

**int getAllocated()**: Returns the number of entries allocated in the buffer. This is used by the HeapTreeFrame object when determining what elements in the buffer should be displayed.

**addEntries(Task[])**: Adds multiple entries to the priority queue. Increments allocated by the appropriate amount. This method was primarily intended to be used by the addMultipleEntries action listener.

**int getAvailableID()**: Returns a random and unique ID for a task. This is necessary for add and addMultiple task action listeners. It uses the containsID method to accomplish uniqueness.

**boolean containsID(HeapBufferArray, heapSize, ID)**: Returns whether or not the passed heap of the specified size contains the ID specified. It accomplishes this through a linear search of all entries. Linear search was selected because the heap will not be sorted; Binary Search is not applicable.

**clear()**: Removes all entries from the priority queue. Intended for use in the clearQueue action listener.

**updateTree()**: A method which updates the tree display if it is currently open. This is done by calling the HeapTreeFrame object’s updateHeap method and passing it the buffer new number of elements allocated.

**int getIndexOf(Task)**: Returns the index of the specified Task in the buffer. If the task is not allocated in the buffer, then it returns -1. This is used by the increaseTaskPriority action listener to determine which element in the heap corresponds to the one in the JList.

ActionListners:

**addTask**: Called when the Add Task button is pressed. If the user entered invalid information for the wait time or task name, it displays an error message to the user. Otherwise, it will determine whether there is enough space in the buffer to add another entry, and then add the entry if this is true using the addTask method in TaskSchedular. It also adds the entry to the taskListModel for display.

**popTop**: Called when the Pop Top Task button is pressed. If there is nothing in the buffer, it displays an error message to the user informing them of this fact. If there is an entry in the buffer, it removes the maximum entry using the popTopTask method defined in TaskScheduler. Also removes the entry from the taskListModel.

**showSorted**: Called when the Show Sorted button is pressed. It copies the allocated elements in the main buffer to a temporary array for sorting, then uses the heapSort method defined in TaskHeap to sort the entries from least to greatest. It then creates an ArrayDisplayDialog to show the sorted task list to the user.

**showRawHeap**: Called when the Show Heap Array button is pressed. It copies all allocated elements from the main buffer into a temporary array, and then passes that temporary array to an ArrayDisplayDialog object to display them to the user. This is intended as a back up to the Heap Diagram window.

**showHeapDialog**: Called when the Show Heap Diagram button is pressed. If the treeDisplay is null or has been disposed, it creates a new instance of HeapTreeFrame of the buffer to display the diagram to the user and then assigns that to the treeDisplay. If treeDisplay is already open, it brings it to the front.

**increasePriority**: Called when the user hits the Increase Priority button. If the user has selected an entry in the Task List, it finds that Task in the buffer using the indexOf method and uses the increaseTaskPriority method defined in taskSchedular to increase its priority value by 100. If no entry is selected, it shows a dialog asking the user to select an entry.

**addMultiple**: Called when the Add Multiple Tasks button is pressed. If there is any space in the buffer remaining, it creates a AddMultipleDialog to prompt the user for information on how many entries to add, what they should be named, and how much time they should have been in queue. The dialog then calls then uses the addEntries method to add the entries to the buffer.

**autoSift**: Called when the Auto Sift radio button is toggled or un toggled. If the user is toggling the button, it displays a warning to the user that doing so will re-arrange the heap. If the user chooses to continue, it sets increaseForWaitTime in taskPriorityQueue and then rebuilds the heap in the buffer and displays it. If the user is un-toggling Auto Sift, then it it sets increaseForWaitTime to false and rebuilds the heap. When toggled, hideOptions will be set to true to prevent the user from corrupting the data in the heap, and when un-toggled hideOptions will be set to false to allow the user to edit the heap.

Class HeapTreeFrame/HeapTreeDrawer

HeapTreeFrame and HeapTreeDrawer are both dedicated to creating a visual representation of the binary tree structure contained in a heap.

The reason this class has been labeled HeapTreeFrame/HeapTreeDrawer is because in file HeapTreeFrame has a private class called HeapTreeDrawer. HeapTreeDrawer is responsible for drawing the elements on a canvas. HeapTreeFrame contains the HeapTreeDrawer in a scroll pane which allows the user to view the tree diagram drawn in it. For the sake of simplicity and space I will display the most important shared Methods and Member Variables for these classes.

Member Variables:

**Task[] heap**: The heap of tasks which this object will display on screen. This is used to keep track of what Tasks are currently being displayed.

**HeapTreeNode[] nodes**: An array of HeapTreeNode components, which serve as the visual representation of nodes in the binary heap tree being displayed. These are used by the HeapTreeDrawer to display the all nodes in the tree.

**SchedulerGuiMain p**: A reference to the parent object of this tree drawer. This reference allows the objects to pull various pieces of data from the main application, as well as perform certain operations on the parent window.

Methods:

**int getHeapHeight(size)**: Returns an Integer representing the height of the heap tree structure using lg(size). This method will be used by the buildFromHeap method to determine proper placement of HeapTreeNode objects in the canvas.

**int getHeapWidthOfBase(size)**: Returns an Integer representing the number of leaf nodes, or elements without children, that there could possibly be at the bottom of the binary heap tree. This method is used for determining proper placement of HeapTreeNode during the buildFromHeap process.

**int getOffSetFromCenter(index)**: Returns an integer representing the offset from the center line of the tree a particular node at a given index in the tree should have. This is also used in the buildFromHeap method to determine proper placement of elements.

**int getHeapIndexRow(index)**: Returns an integer representing the row a particular node at the specified index should fall on in the binary heap tree. This method is used to determine proper node placement during the buildFromHeap process.

**buildFromHeap(HeapBufferArray, heapSize)**: Creates a graphical representation of a specified heap using heapTreeNodes to represent each task in the heap. heapSize must be specified, as the method will use the buffer specified in SchedulerGuiMain. The method uses two for loops: one which places heapTreeNodes in the correct position, and another which sets up the relationships between each node.

**paint(Graphics g)**: It was necessary to modify the paint method for the canvas in order to show the relationships between the nodes. The only addition is a single for loop which iterates over every entry in nodes and draws a line from a given node to it’s parent if it has one.

For a more in depth look at how the HeapTreeFrame class is implemented, please refer to the comments in the source code.

**Class AddMultipleDialog**

The class AddMultipleDialog extends JDialog and serves to gather information from the user during the add multiple task operation. Its modularity is configured such that it prevents the user from performing any other operation in the application while it is active. This prevents potential errors from arising.

Member Variables:

**SchedulerGuiMain parent**: A reference to the parent SchedulerGuiMain object of this AddMultipleDialog. This reference is necessary in order to allow this Dialog to add the entries using the addEntries method in SchedulerGuiMain.

**JTextField taskNamer**: A text field which will hold the task name input from the user.

**JButton cofirmB**: A button which allows the user to finish the add multiple tasks operation.

**JButton cancelB**: A button which allows the user to cancel the add multiple tasks operation.

**JSpinner amountS**: A spinner where the user can enter the number of nodes they want to add. When the dialog is created, it automatically sets the upper limit of this spinner to be the same as the max number of nodes which could possibly be added.

**JSpinner delayS**: A spinner where the user can enter the amount of time all the nodes to be added have been in the priority queue. This spinner will be enabled/disabled depending on the setting of the delayTypeCB.

**JComboBox<String> delayTypeCB**: A combo box with options for how entries should be assigned wait time in queue. It contains Random, Custom, and None. None means all entries will be given a wait time of zero, Random means all entries will be given a random wait time between 0 and 1000, and Custom allows the user to enter a desired wait time in the delayS spinner.

Methods:

**AddMultipleDialog(Scheduler)**: Constructor for the AddMultipleDialog class. It initializes all of the JComponents stated above and positions them using a GridLayout.

Action Listeners:

**delayTypeSelcted**: Called whenever the user selects the desired delay type. If they selected Custom, then the spinner delayS is enabled so the user may enter the value they want, otherwise it is disabled.

**confirmOperation**: Called when the button confirmB is pressed by the user. This action listener pulls the relevant information from JComponents and creates a Task[] of all the entries to be added. It then uses the reference to the parent ShedulerGuiMain object to call the addEntries method and add the entries to the heap. The dialog will close after this is complete.

**cancel**: Called when the button cancelB is pressed by the user. It disposes this instance of AddMultipleDialog, closing the dialog.

Class ArrayDisplayDialog

The class ArrayDisplayDialog extends JDialog and serves to display an array of Tasks to the user. It is relatively simple in its implementation and is only about 50 lines. It creates a text field in a scroller, so the user can view really long strings of entries. In early implementations, a simple showMessage call was used with a string representation of the Array, but this proved inconvenient for large data sets as the text would blur and the window would extend passed the boundaries of the screen. This fixes those problems with a scroller and a text field.

Member Variables:

**JScrollPane disp**: The scroll pane which will hold the text field. This scroll pane prevents the text field from extending past the boundaries of the screen.

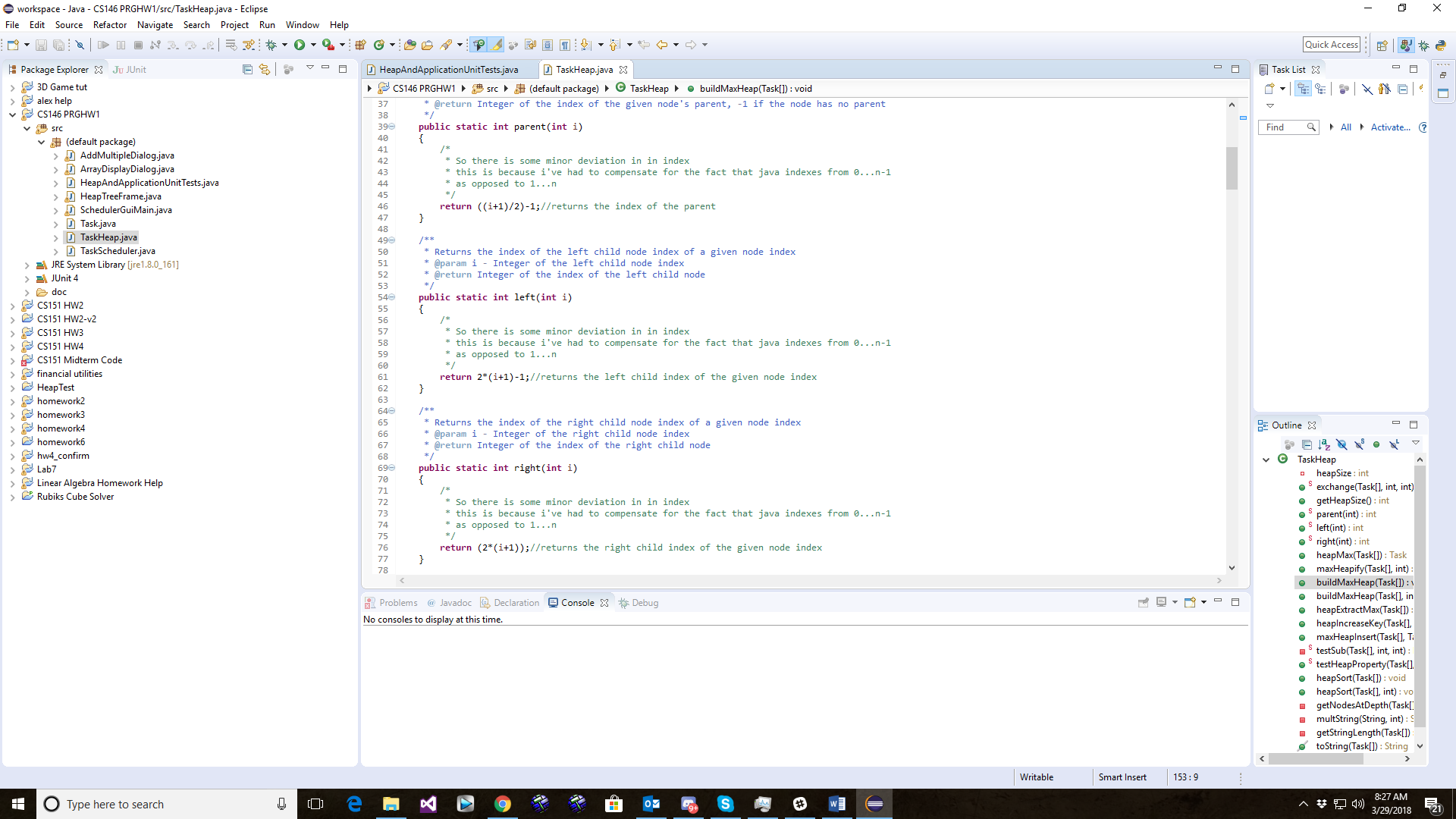
**JTextField container**: A text field which will contain the String representation of the task array being displayed.

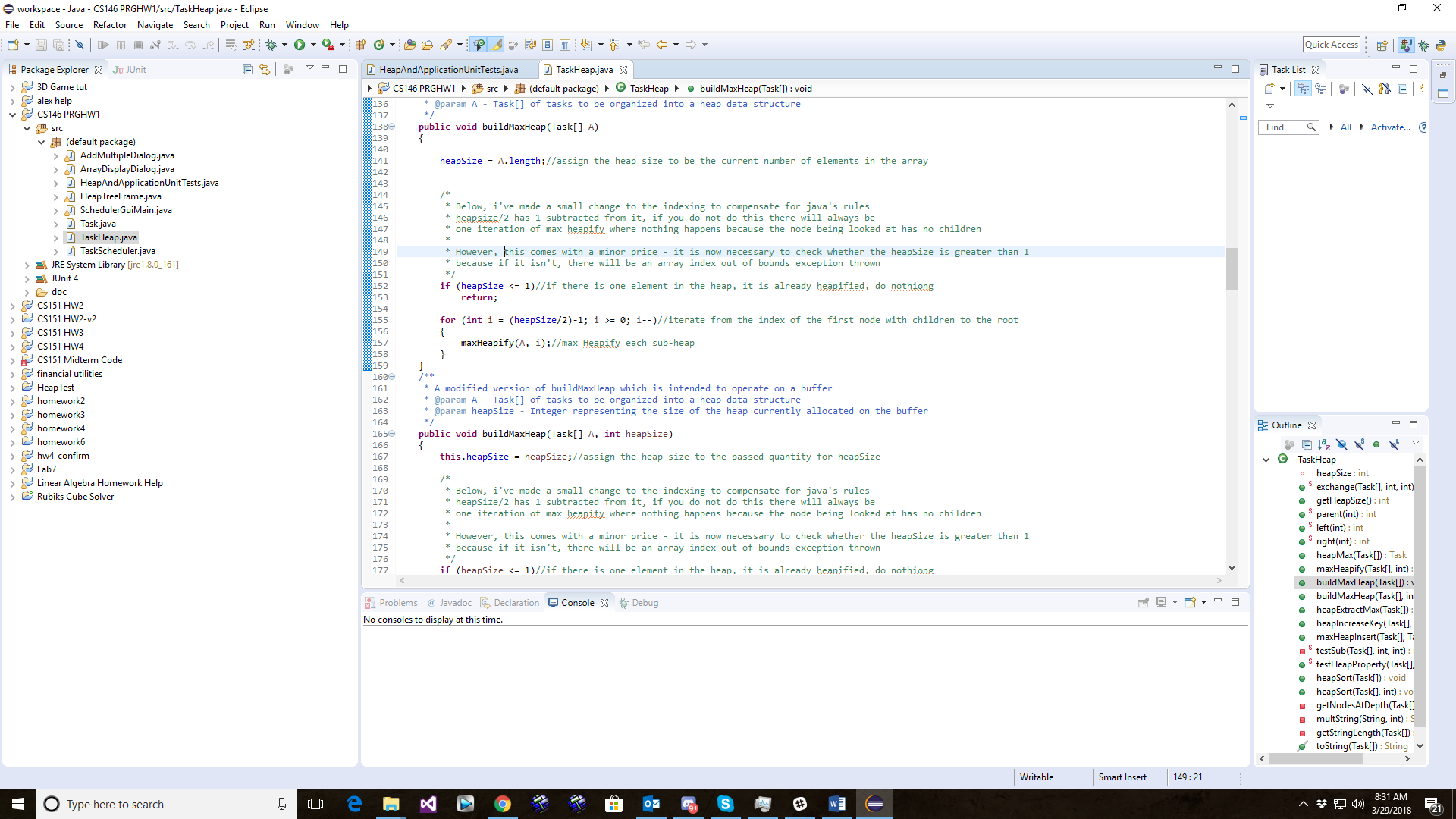
Methods:

**ArrayDisplayDialog(Task[] array)**: The constructor for the ArrayDisplayDialog class. It simply initializes the scroller and text field component to contain the string representation of the passed array using Arrays.toString. There is also a close button.

Challenges and Solutions

Indexing

 One challenge encountered during the development of this application was determining how to properly calculate the index of element in the heap. The pseudo code in the book assumes that indexes go from 1 to n. However, in java, indexes go from 0 to n-1. If the functions were written exactly as they were in the pseudo code, the application simply would not work. In order to correct this issue, the solution I came up with was to add 1 to each index passed into the methods for indexing, perform the calculation as written in the pseudo code, and then subtract 1 from the result to get the proper index in java.



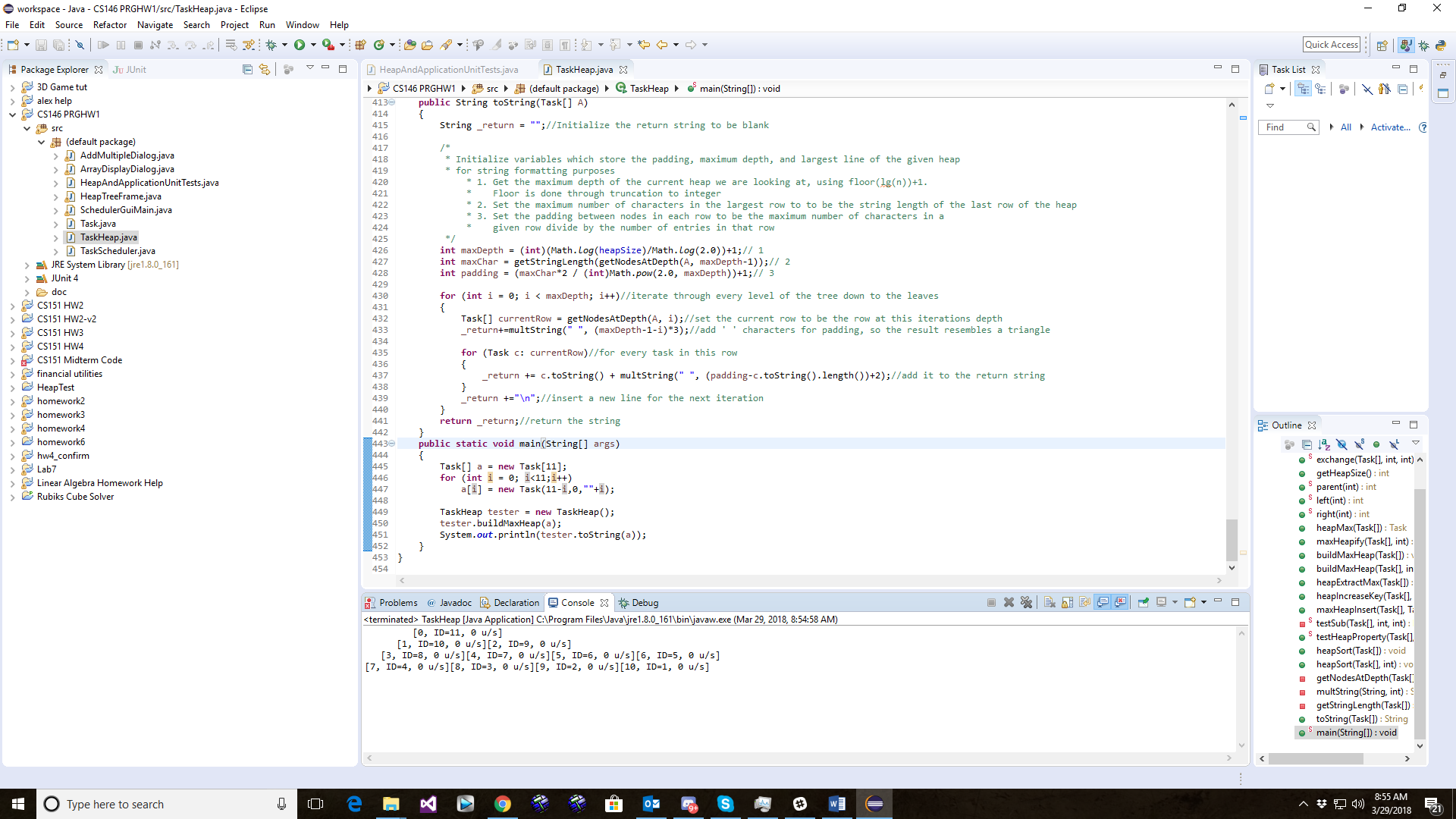
There was another instance where this issue arose; during the implementation of the build max heap function, I noticed that simply dividing the size of the heap by 2 would cause one iteration of the loop to do nothing during the max-heapify operation. After debugging and stepping through the code, I discovered that that size/2 yielded a child node instead of the parent. The solution to this was to subtract 1 from size/2. However, one consequence of this was that if the heap had only one element or less in it, the code would throw an array index out of bounds exception. To fix this, an if statement was added above the for loop to check whether more than one element was allocated in the heap. If this condition wasn’t met, the execution would terminate.

Dynamic Management of the Task List

Another challenge encountered the development process was determining a way to dynamically manage the heap. The first instance where this became an obvious problem was in the implementation of the heap max insert and heap extract max operations. Since both operations add and remove elements from the array, it is necessary to increment and decrement the size of the heap respectively. However, the pseudo code assumes that you can add an instance variable to the array being passed to the function on the fly. This is possible in some languages (Lua and meta-tables come to mind) but in java this is simply not an option. The solution was to create an instance variable in the TaskHeap which would store the size of the heap over time, initialize this variable whenever a build heap or heapsort operation was performed.

Later, during the development of the GUI, it became clear that some form of a buffer would be necessary in order to allow the user to enter Tasks into the heap while the application was running. A simple Task array was created for this purpose. However, the build heap operation was initially implemented to assign the heap size to the length of the array. This assumes that all entries in the array have already been allocated, an assumption which is not true in the instance of a buffer. The solution to this problem was to overload the build max heap method to be able to take the initial size of the heap as a parameter. This prevented the build heap operation from failing due to a null pointer exception.

Displaying the Heap Tree

Another challenge during the development of this application was coming up with a way that the heap could be displayed to the user. I wanted to be able to display the heap in the way that showed its heap structure. Initially, I attempted to do this with a toString method. The results were inconsistent and messy, as the padding didn’t make a clear pyramid structure.

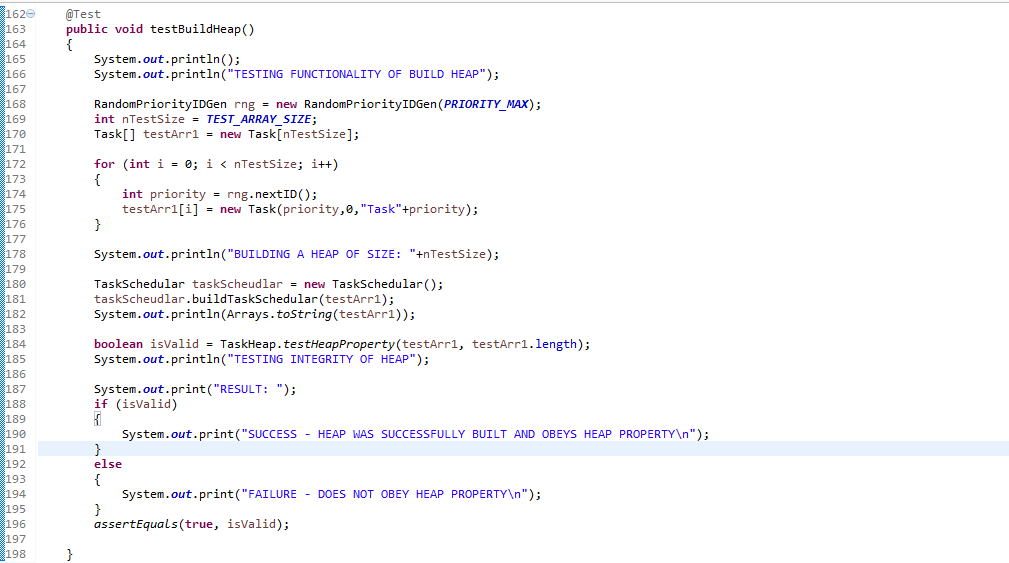
However, during the development of this toString method, I figured out a few ways to calculate how nodes should be position on screen which I thought would translate well to a GUI application. I decided that String methods were not an adequate solution to displaying the tree structure, so I began working on a JPanel that could take a heap as an input and display a graphic representation of it’s binary tree structure. The system I came up for doing this was to use rectangular UI elements as nodes, and then position them using calculations which I derived by drawing and measuring heap trees. I wrote methods for calculating the offset of nodes from a central line, the ‘row’ on which a node should be placed, and the amount of padding that need to be provided in the window in order to make all nodes viewable. After nodes were properly positioned, the relationships to each node were calculated using another for loop. Once both these operations were done, the only thing which was necessary was overload the paint method for the JPanel to draw a line from each child node to its parent node.

Due to the variable size of this tree, the JPanel “HeapTreeDrawer” was placed inside a JScroller to allow the user to view the tree without the window taking up the entire screen. A button was also added to allow the user to refresh the tree being displayed if there were any changes made to the underlying data structure.

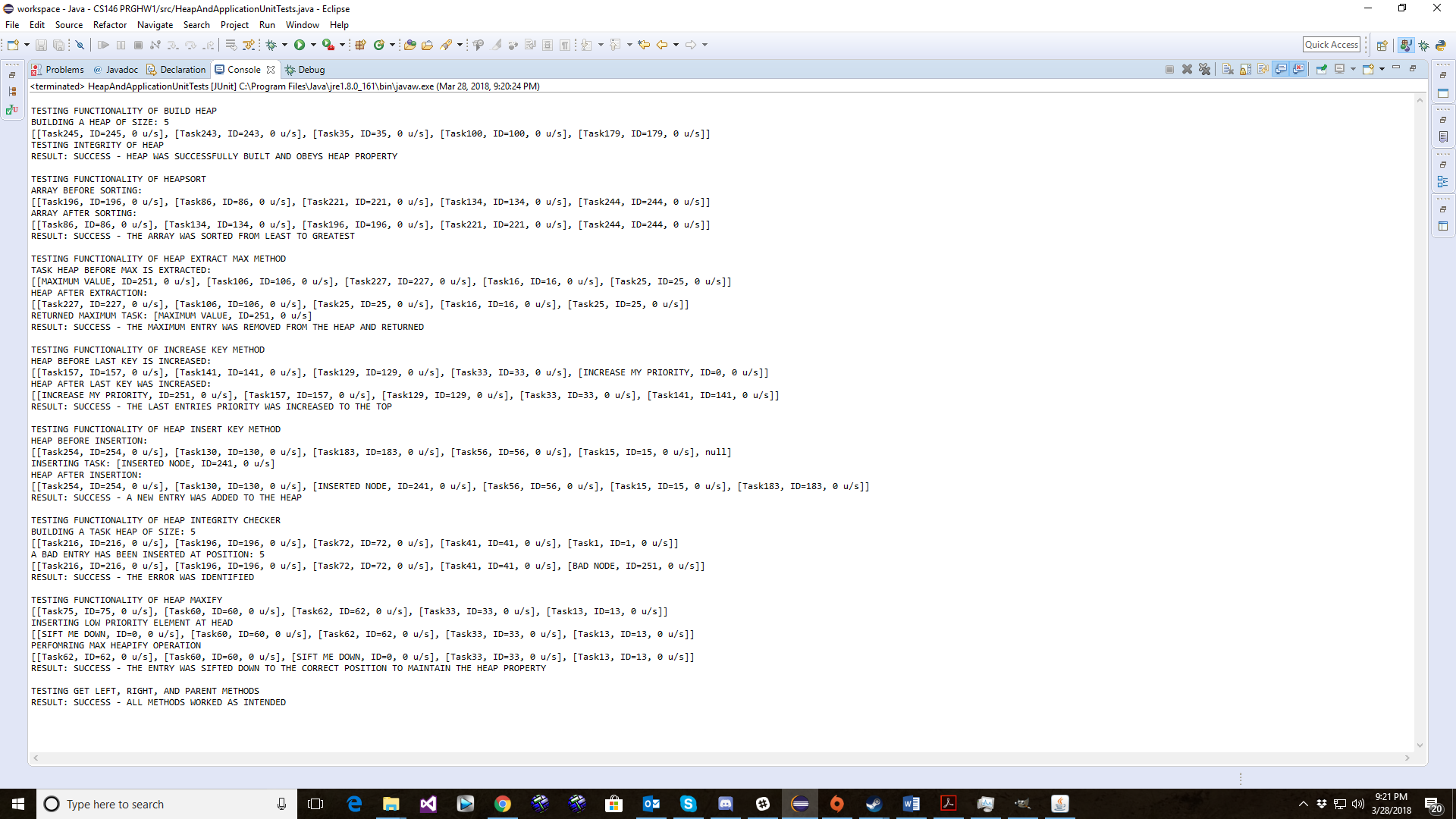
Testing

Heap

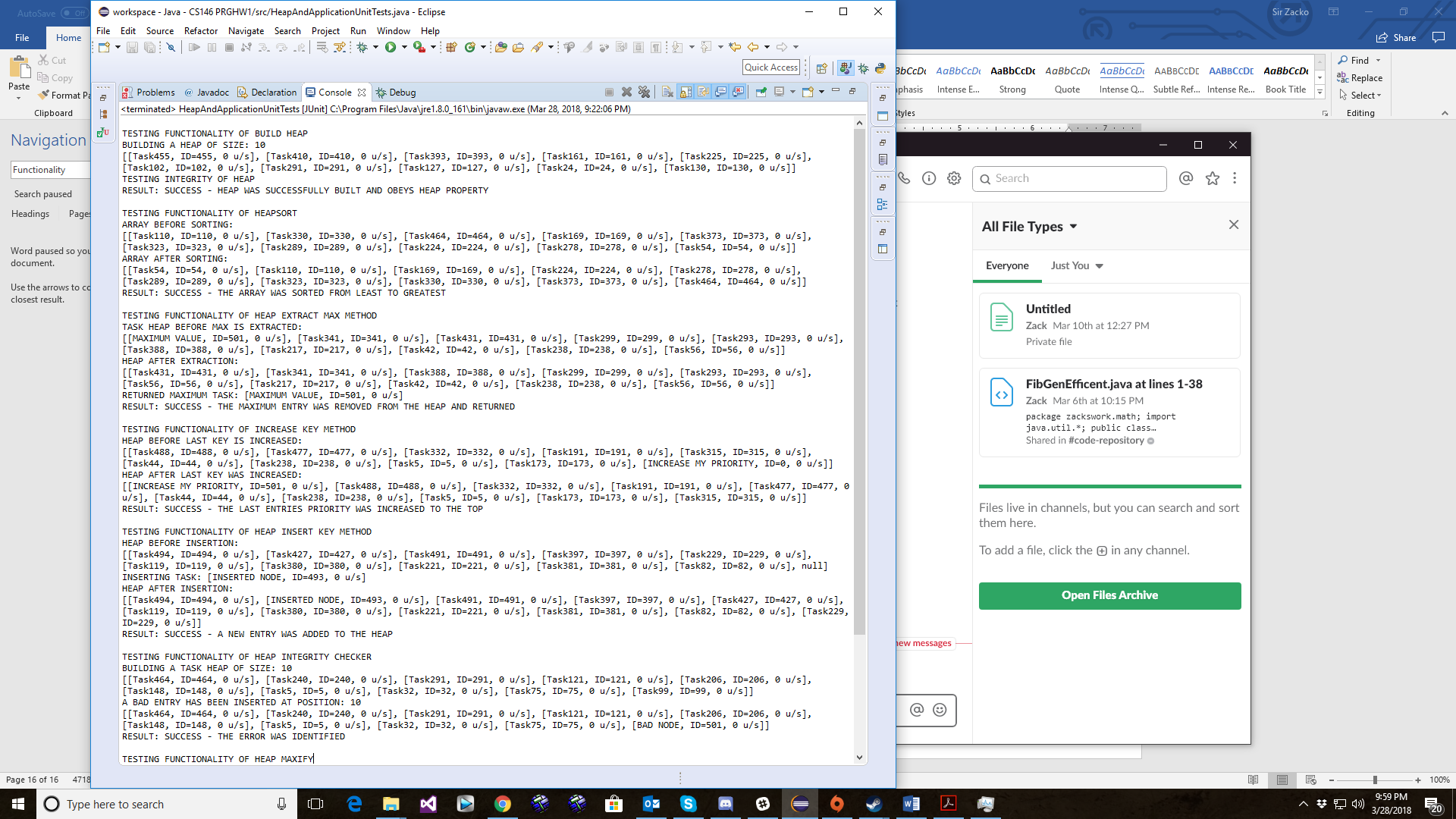
For the sake of code cleanliness, nearly all testing on the heap data structure was done through the JUnit tests. You can find the source code for these unit tests in the source file titled “HeapAndApplicationUnitTests.java”. Each test focuses on the functionality of a single function. The tester is configured to be able to test functionality data sets of different sizes. Testing was done using small data sets at first (5-15 element arrays), and later using larger data sets (512-1024 element arrays). Functions such as the testHeapProperty method are employed to check the integrity of heaps which are too large to be checked by eye.

Build Heap Test

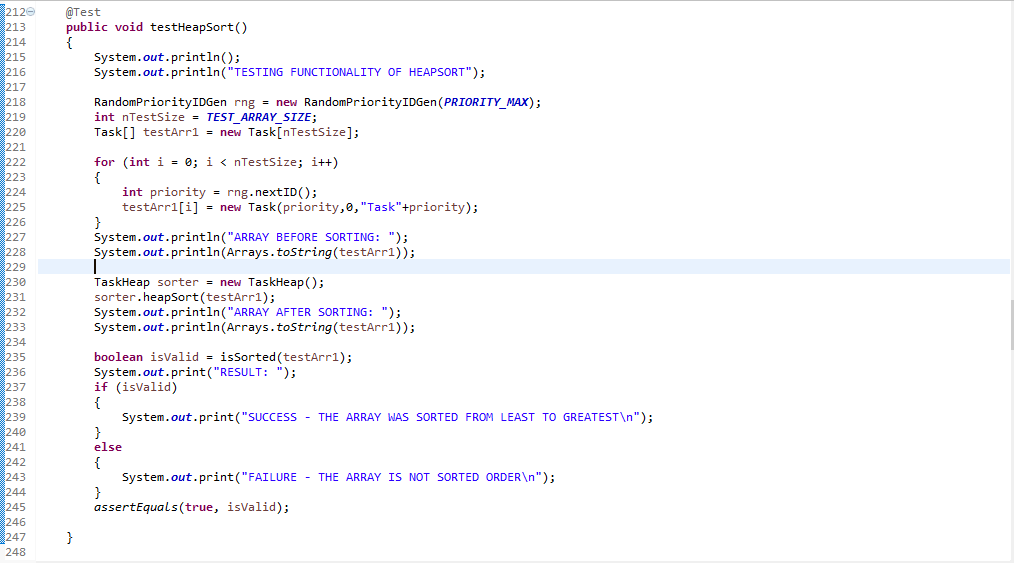
Output for 5 elements:



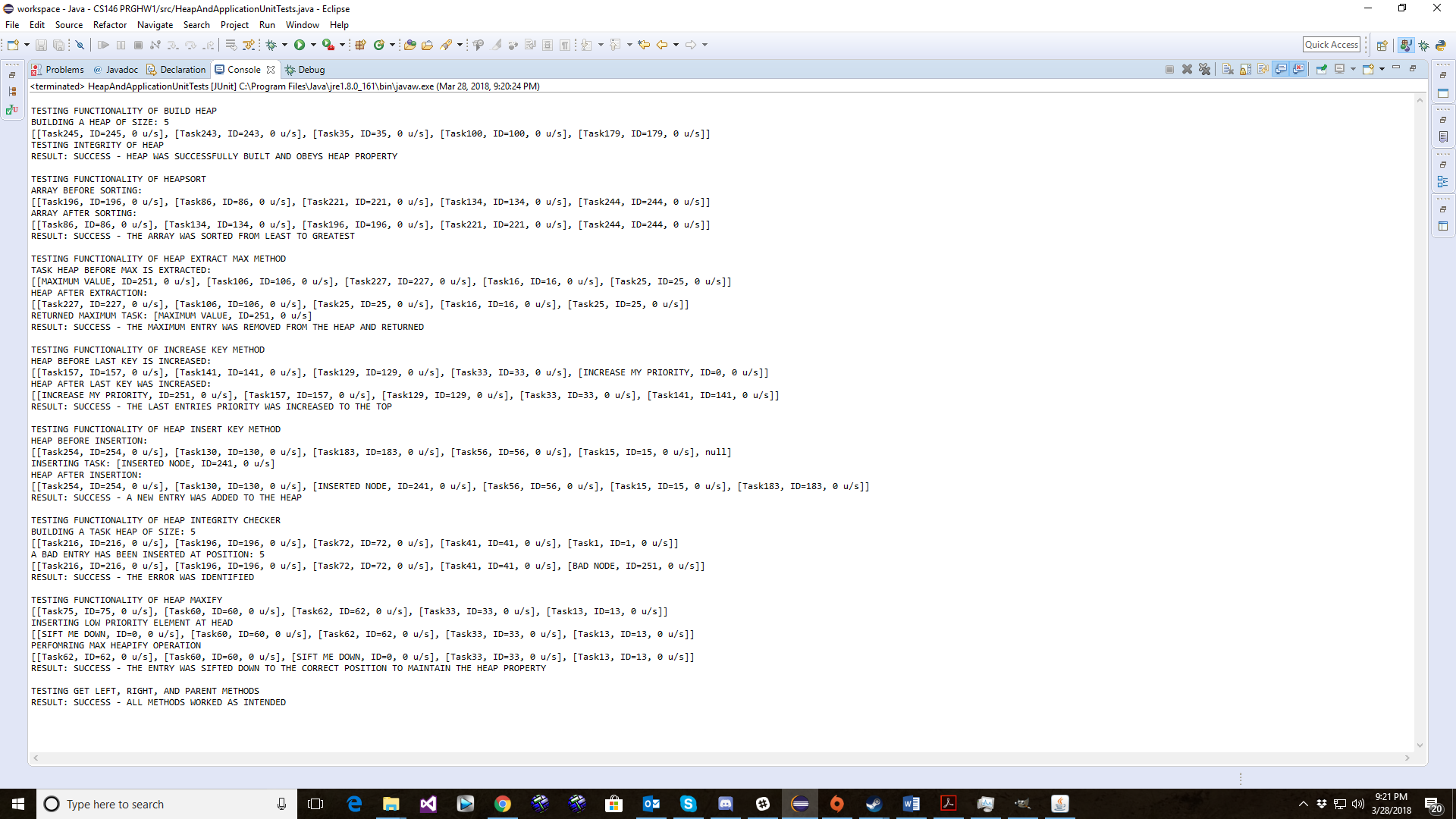
Output for 10 elements:



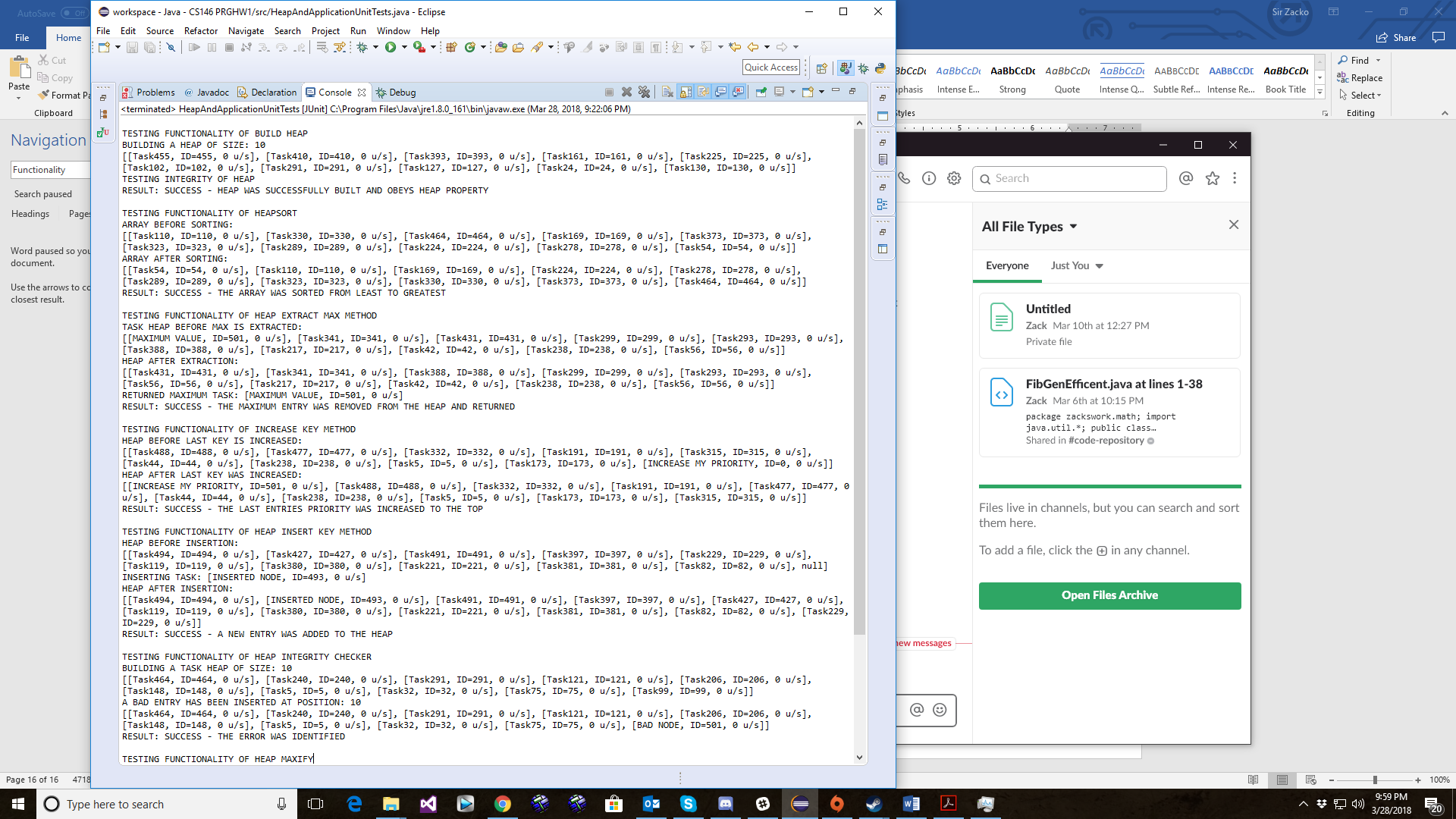
HeapSort Test



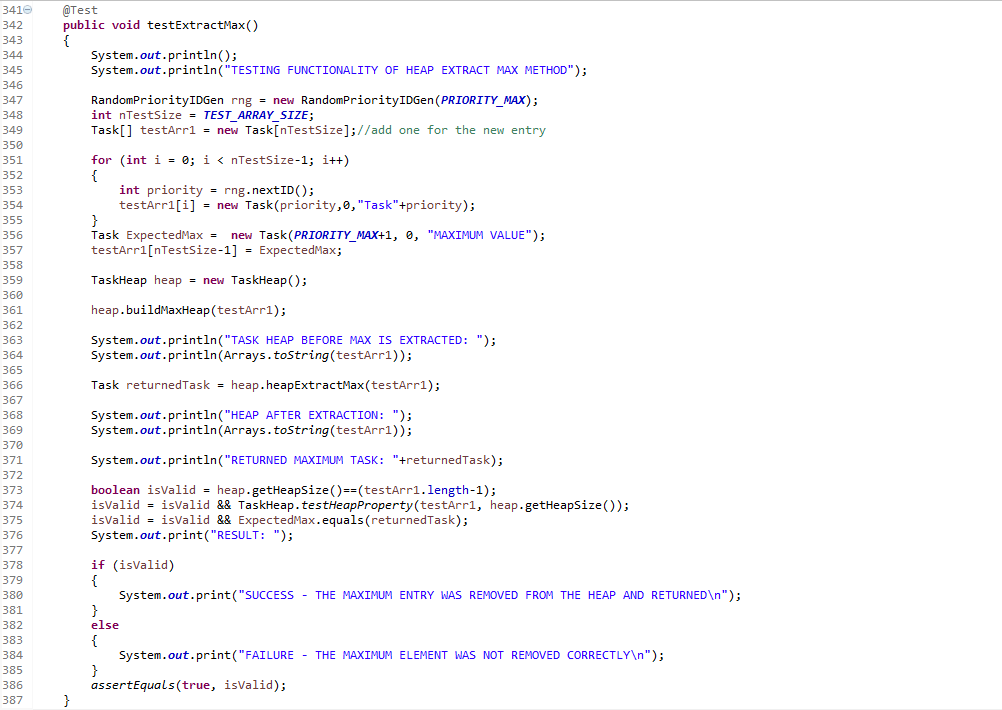
Output for 5 elements:



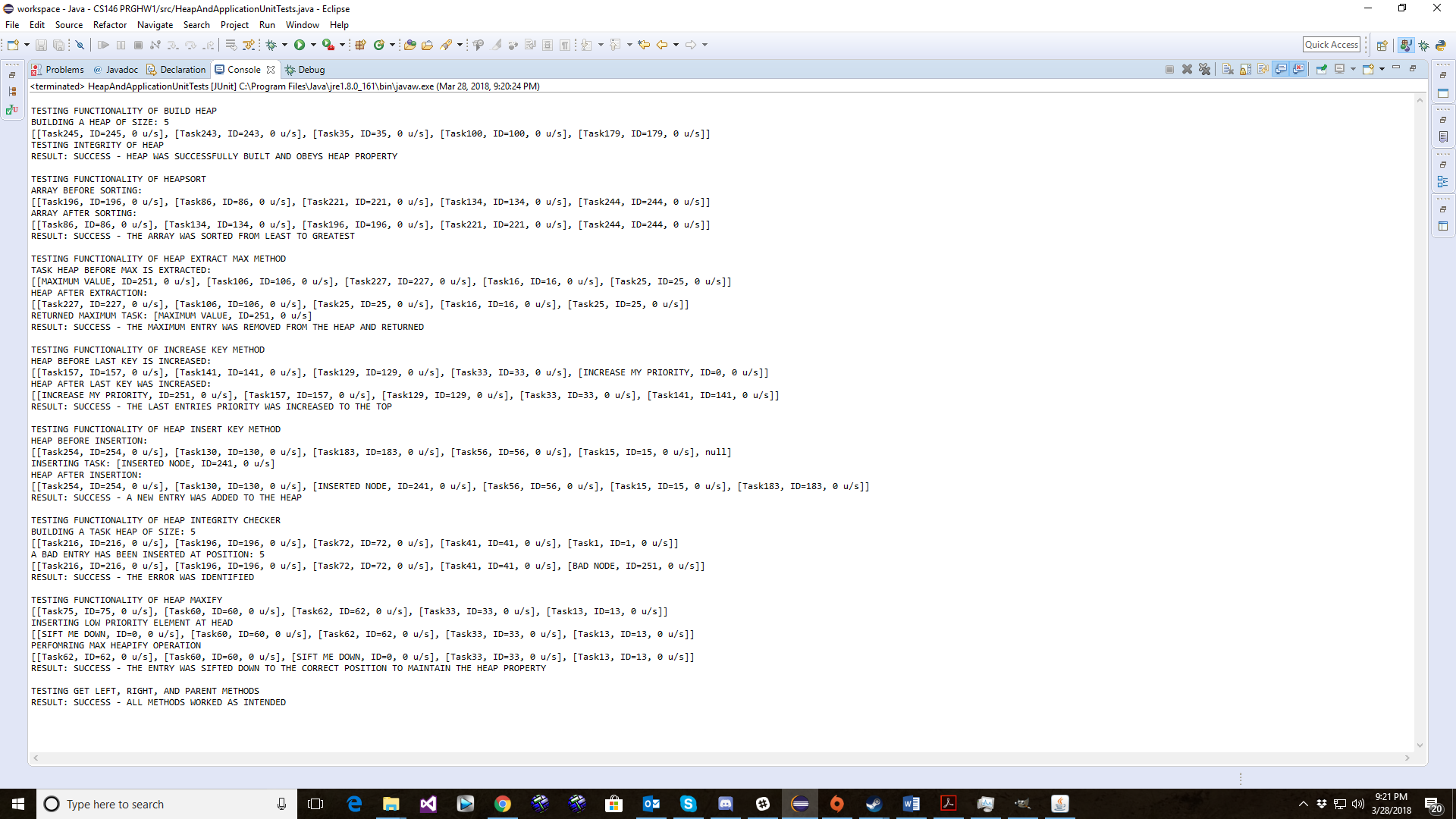
Output for 10 elements:



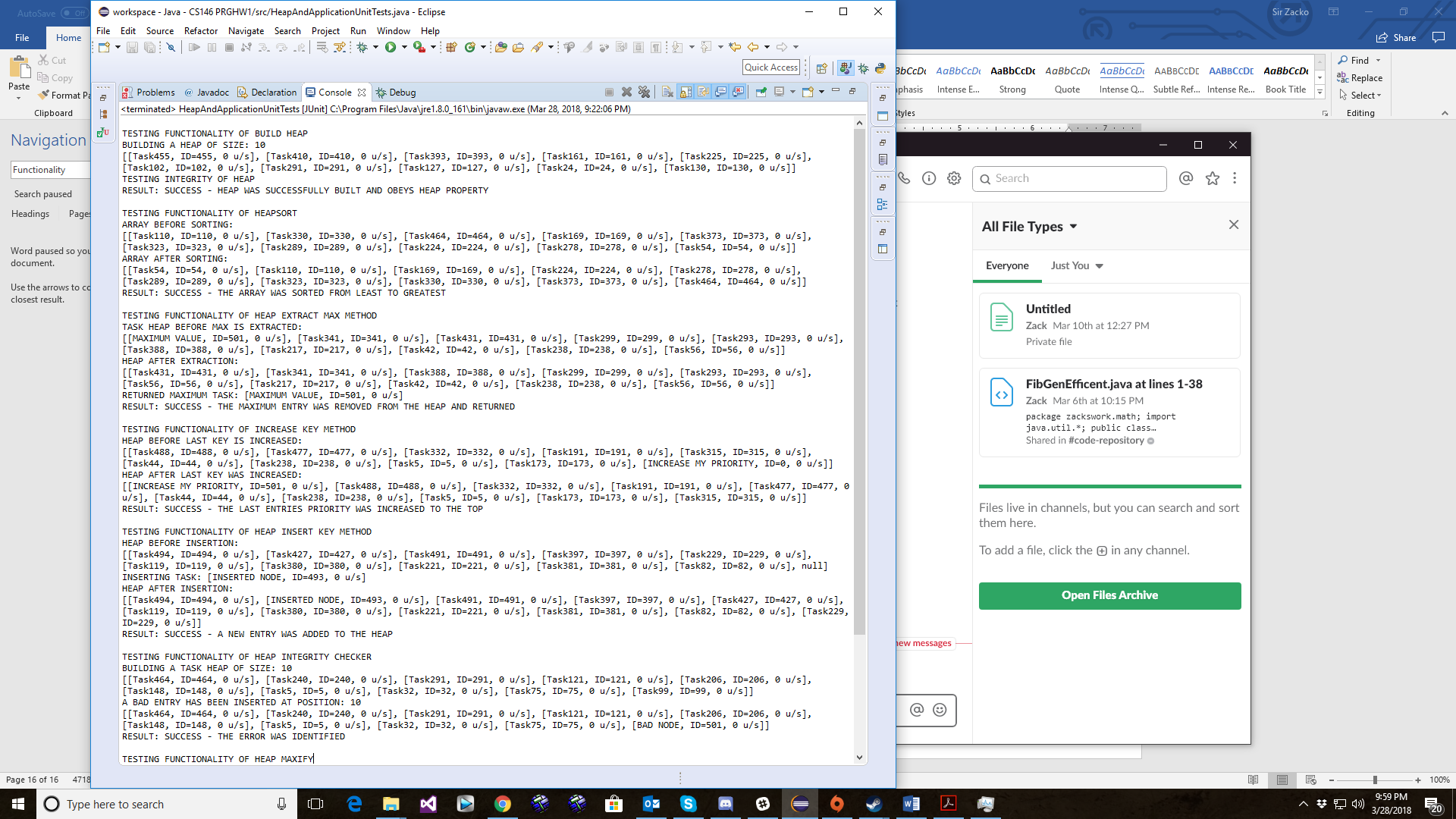
Heap Extract Max Test



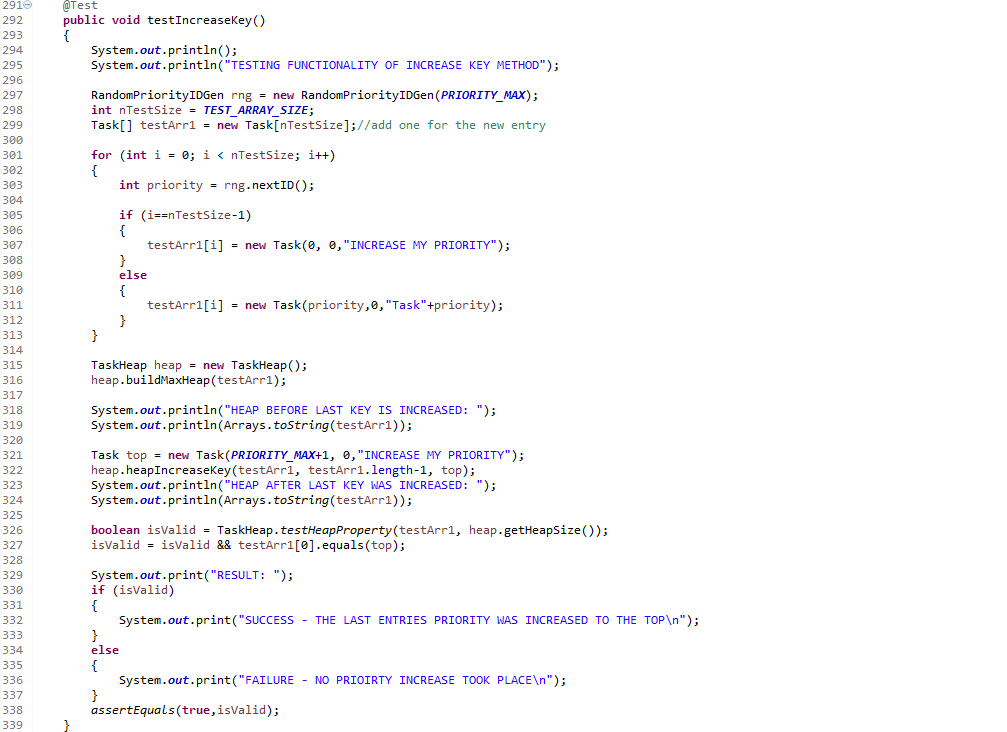
Output for 5 elements:



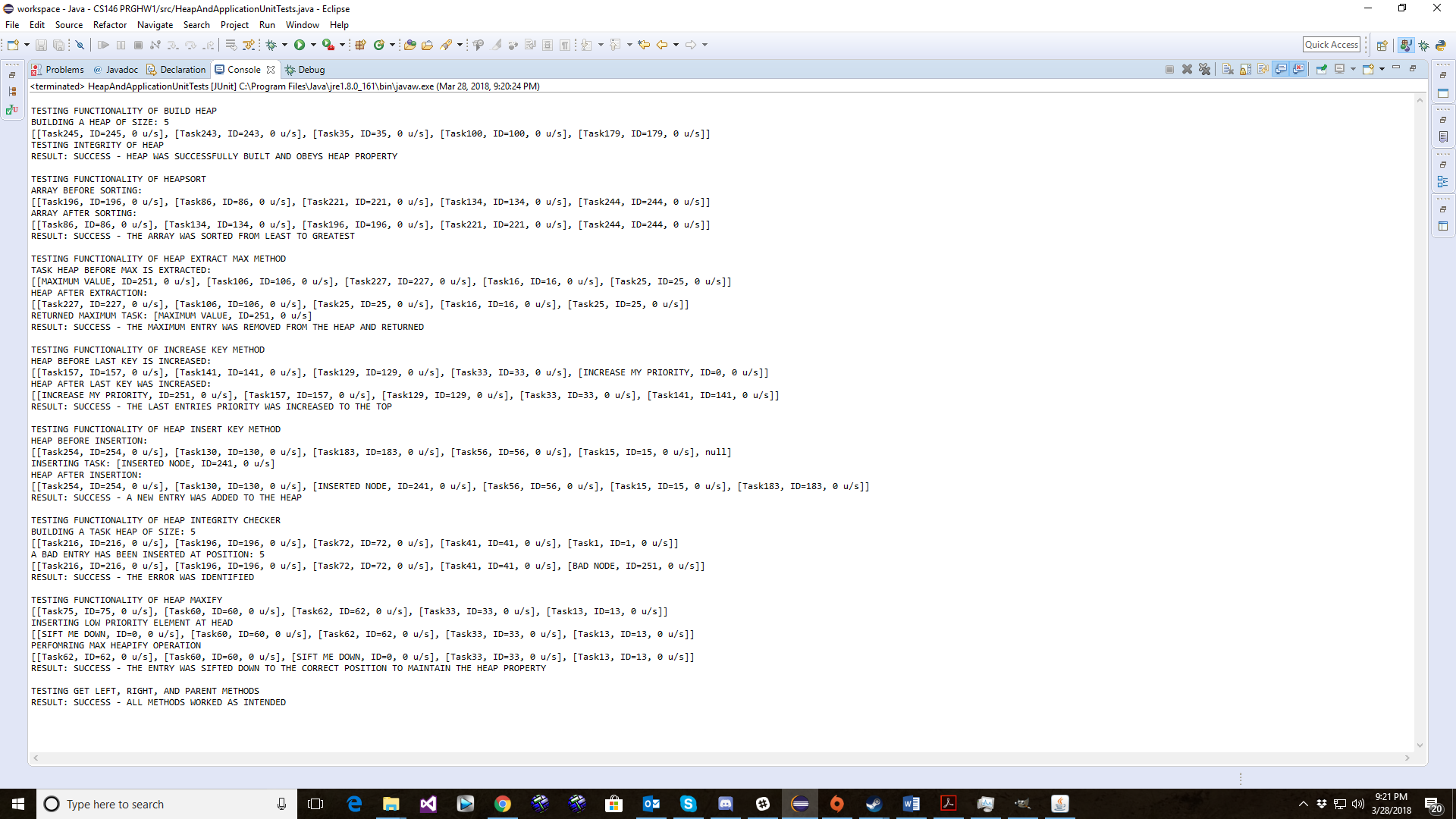
Output for 10 elements:



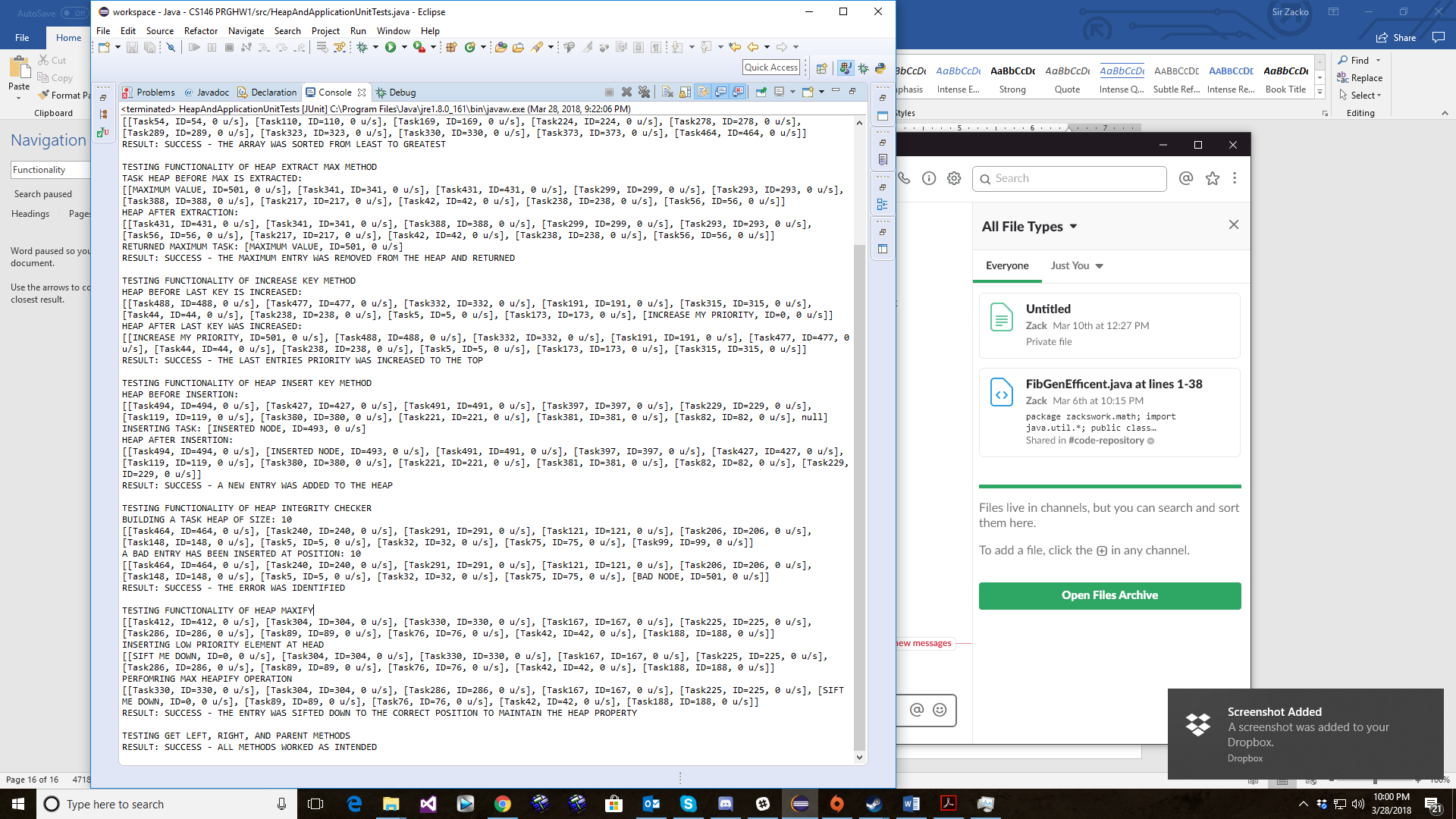
Heap Increase Key Method



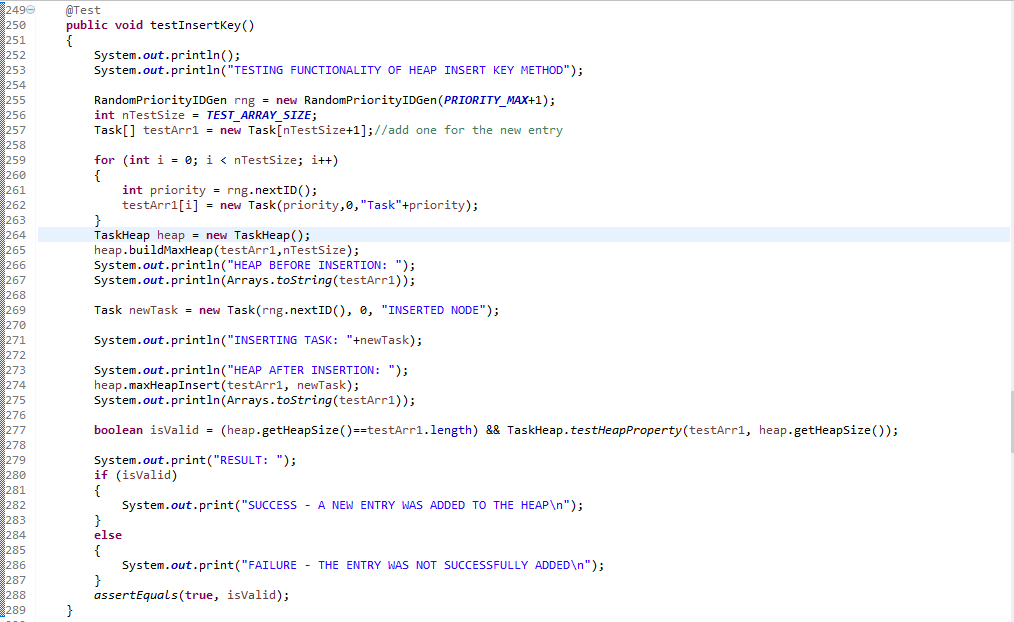
Output for 5 elements:



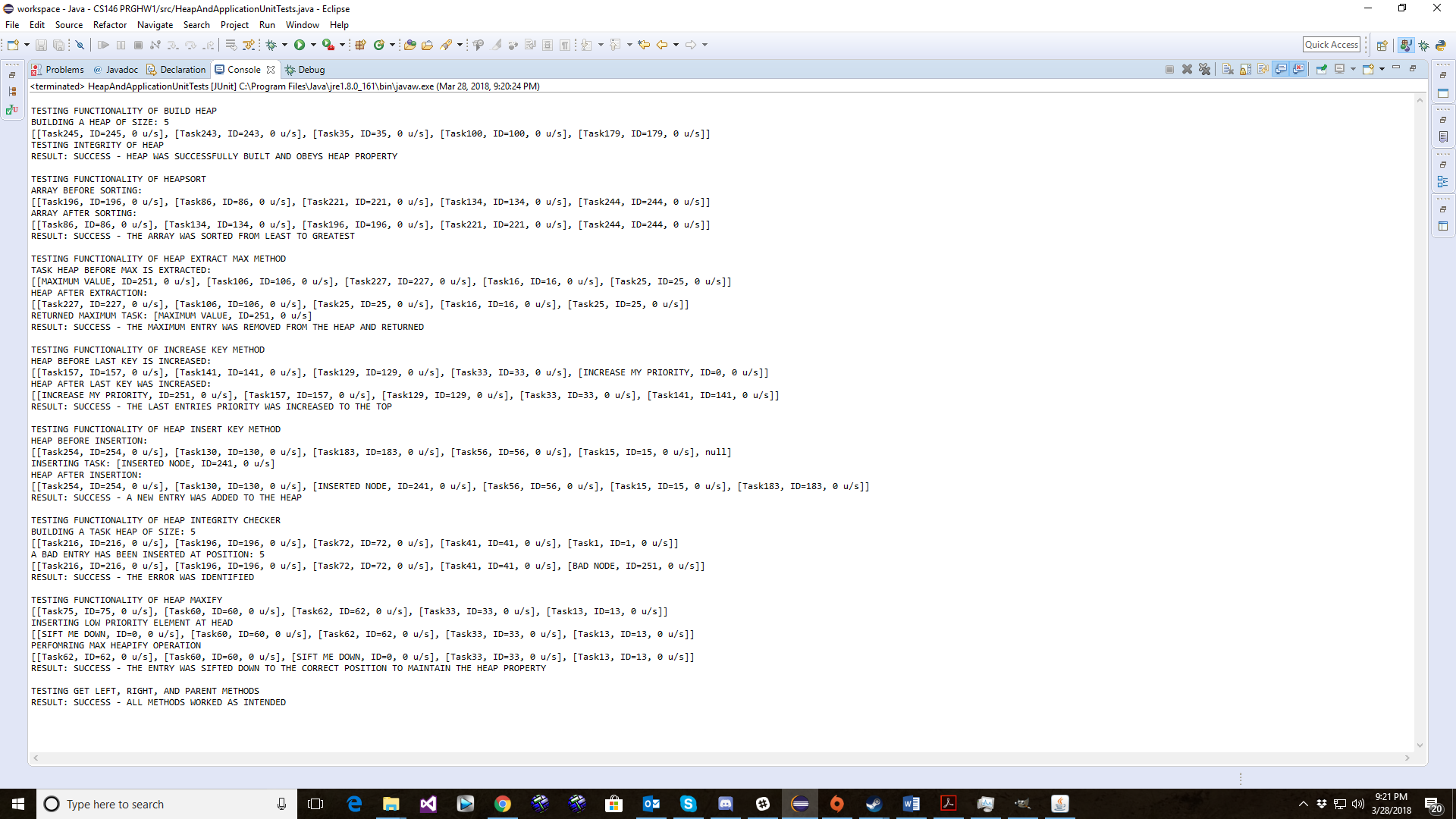
Output for 10 elements:



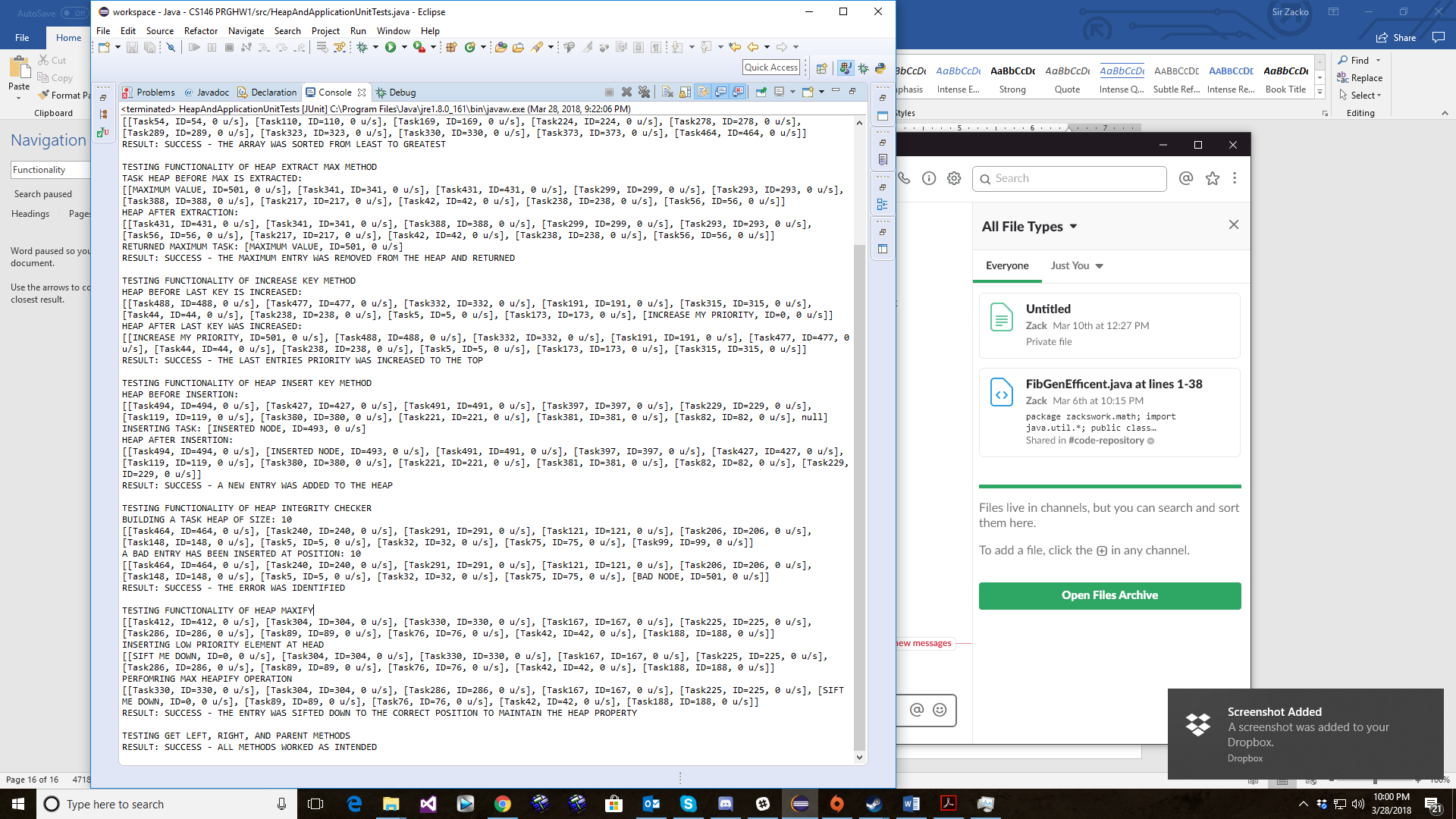
Heap Insert Key Test



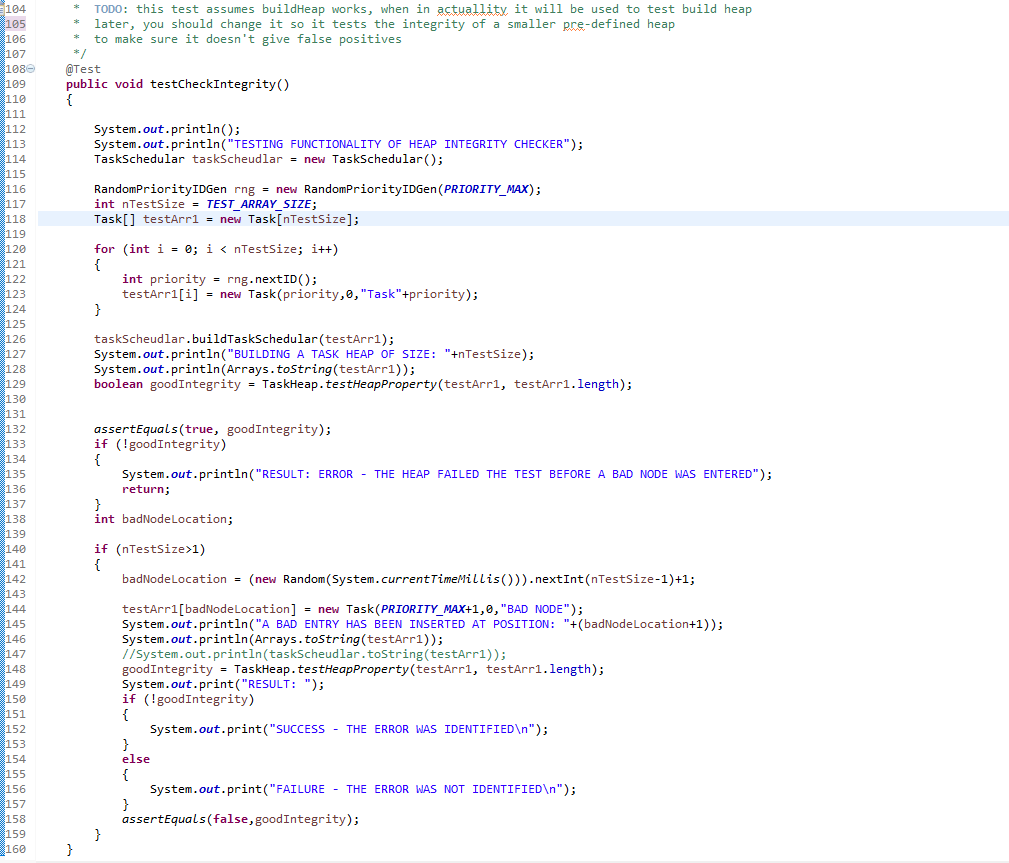
Output for 5 elements:



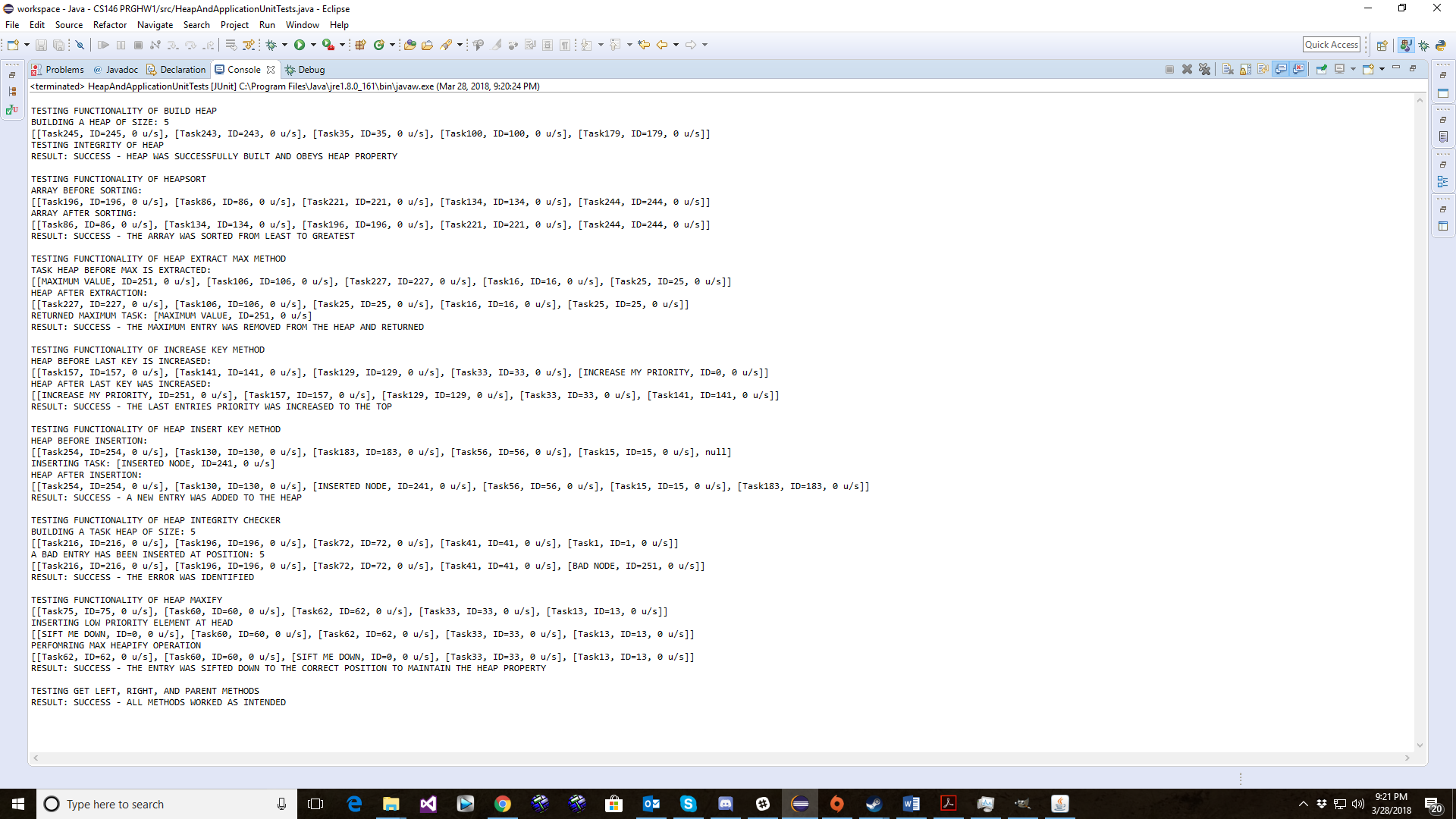
Output for 10 elements:



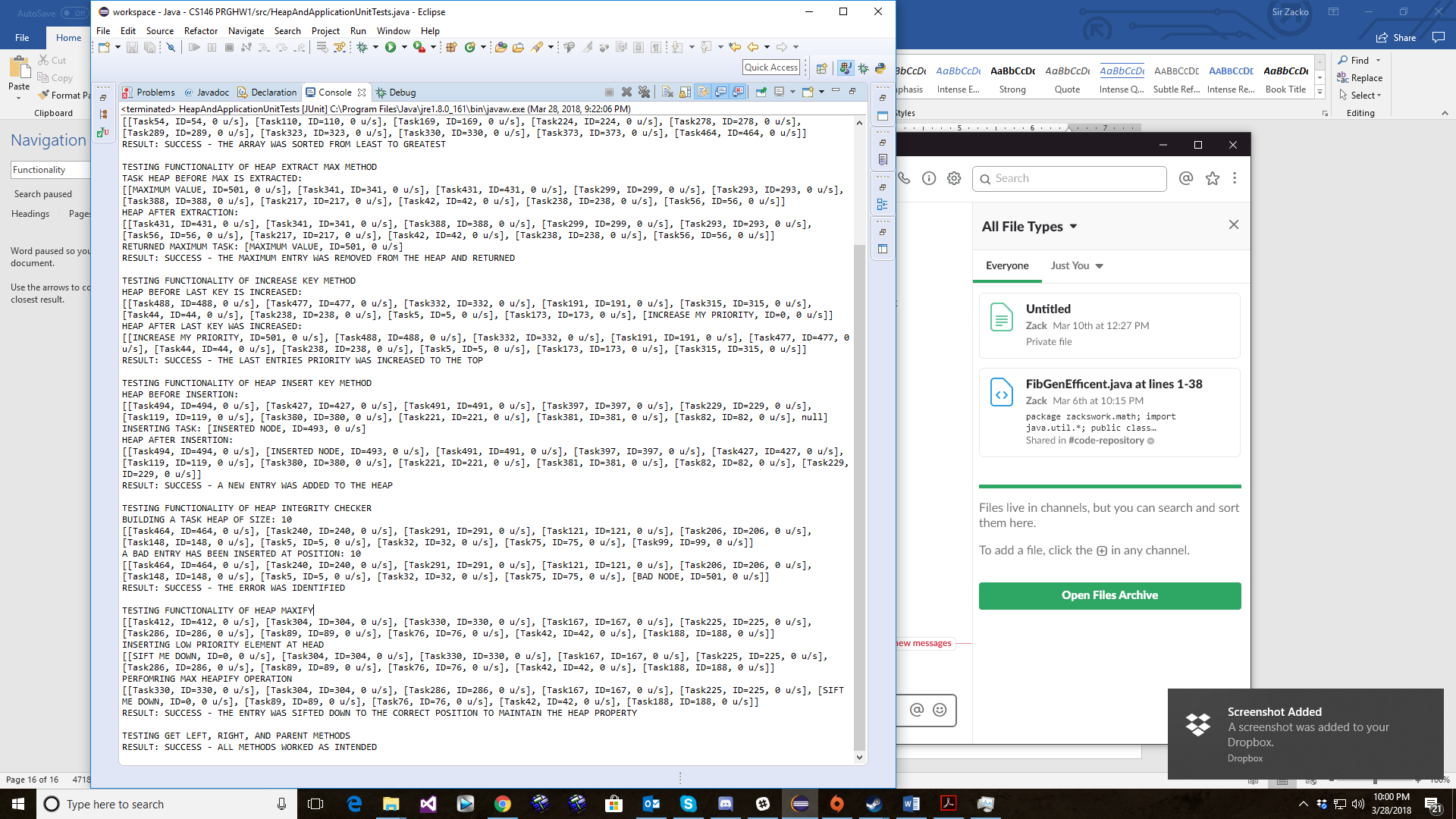
Testing Functionality of Integrity checker



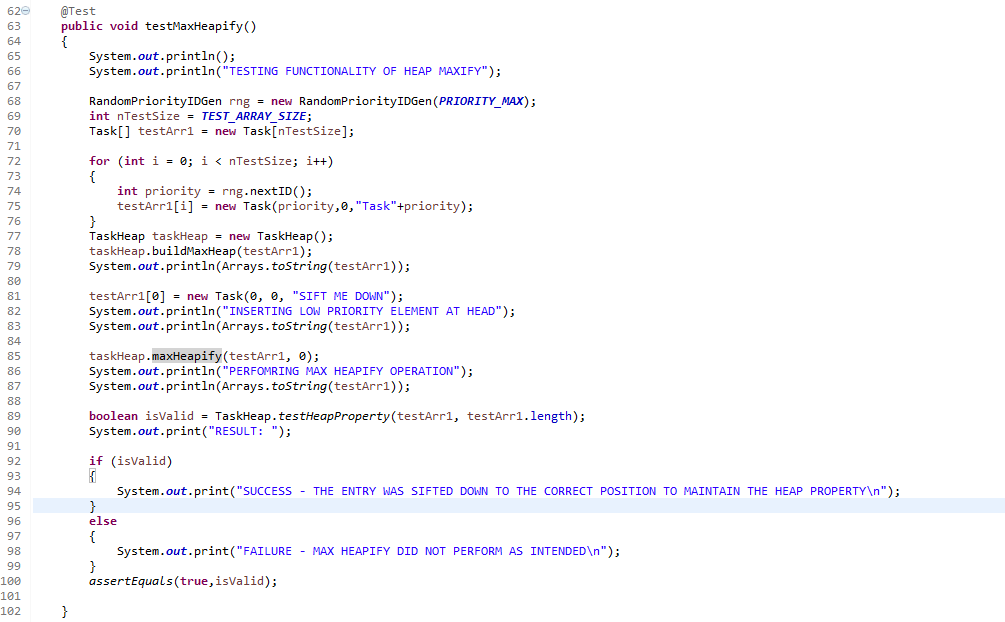
Output for 5 elements:



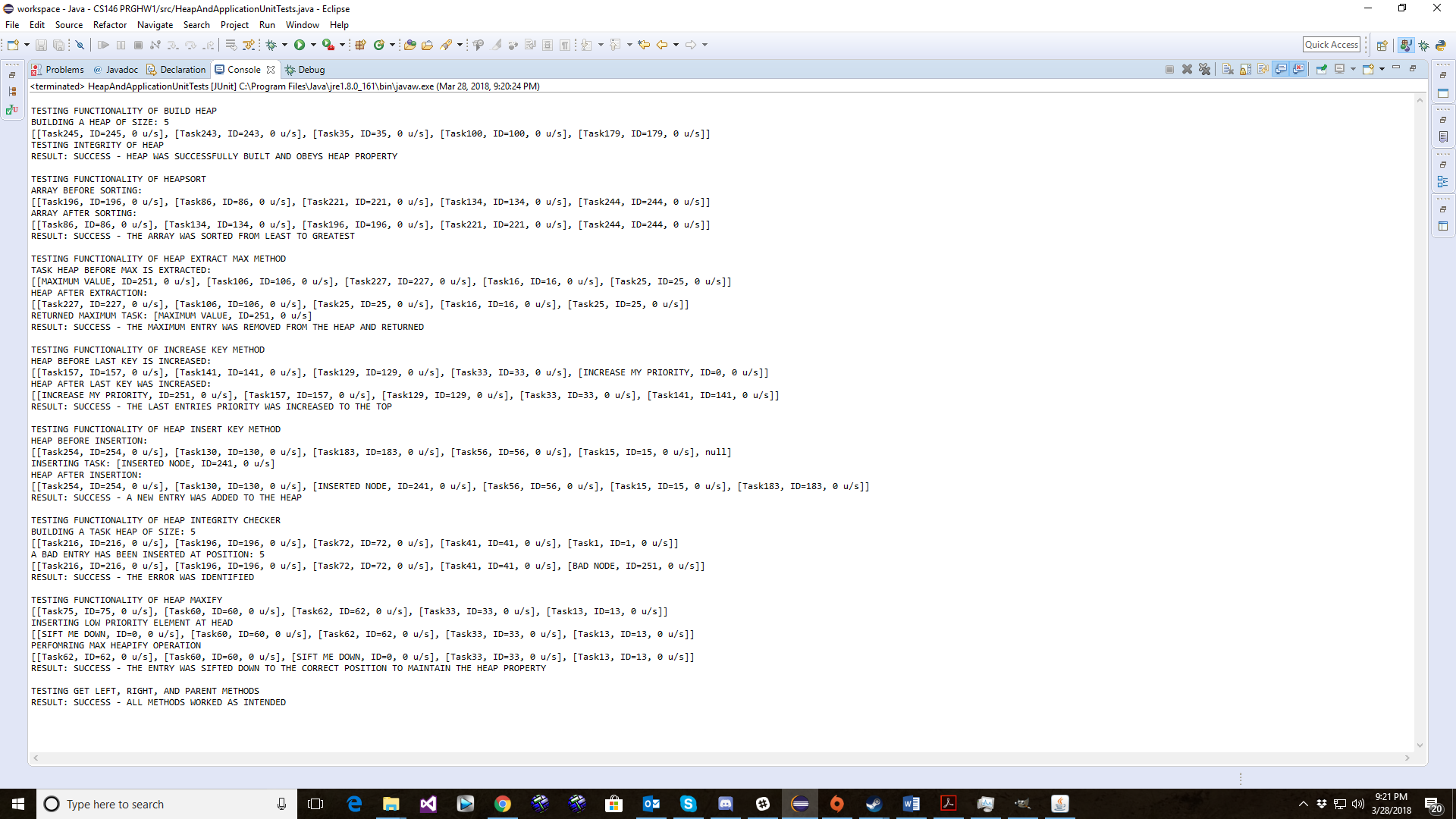
Output for 10 elements:



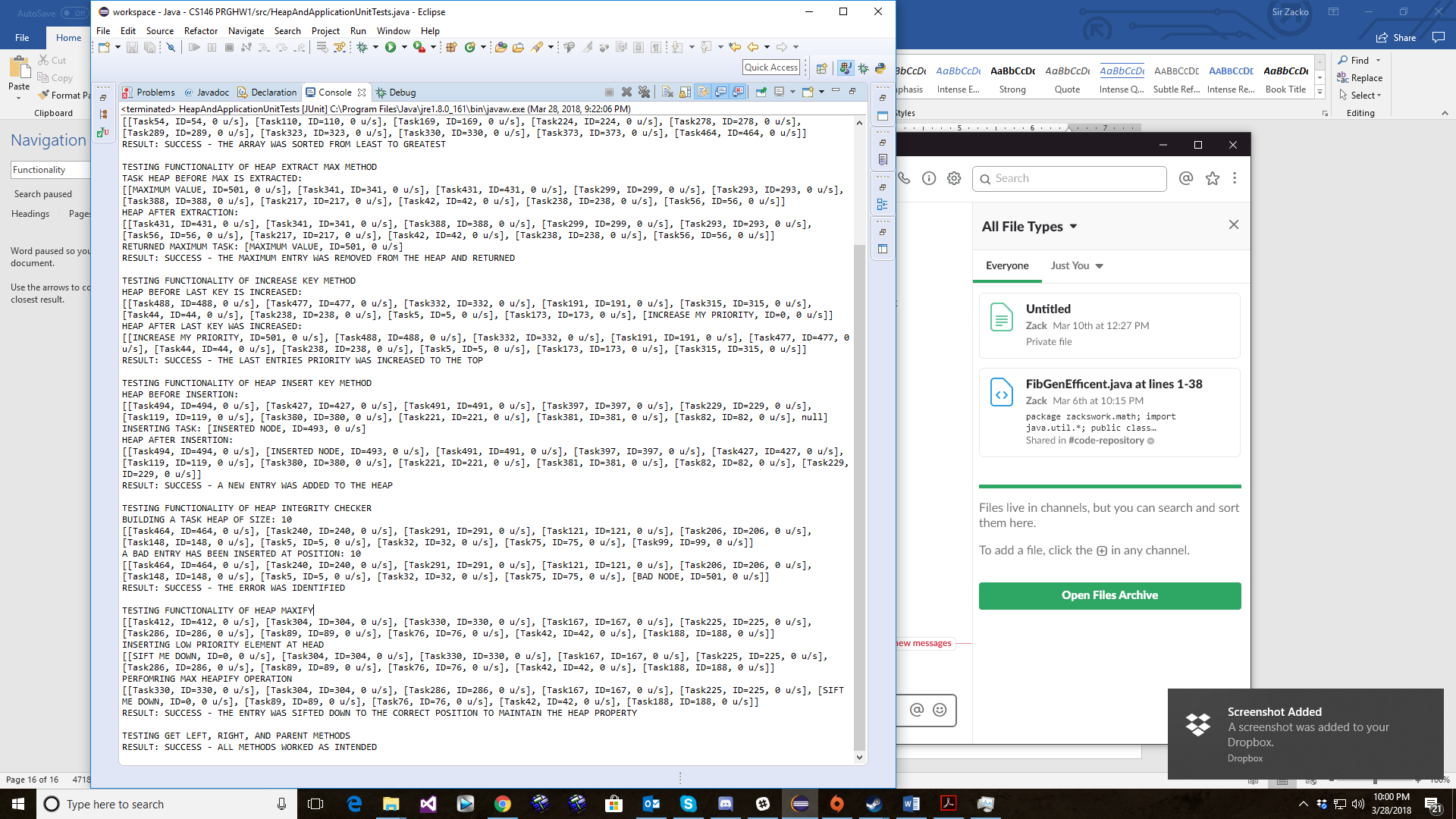
Heap Maxify Test



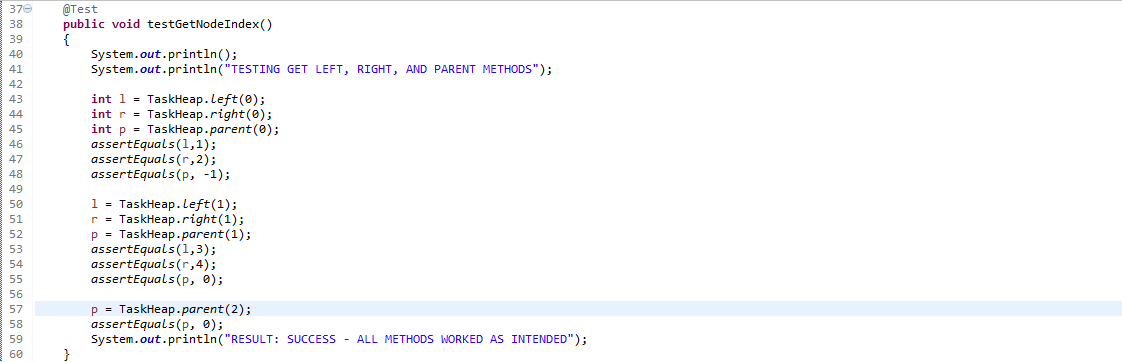
Output for 5 elements:



Output for 10 elements:



Parent/Left/Right method Test

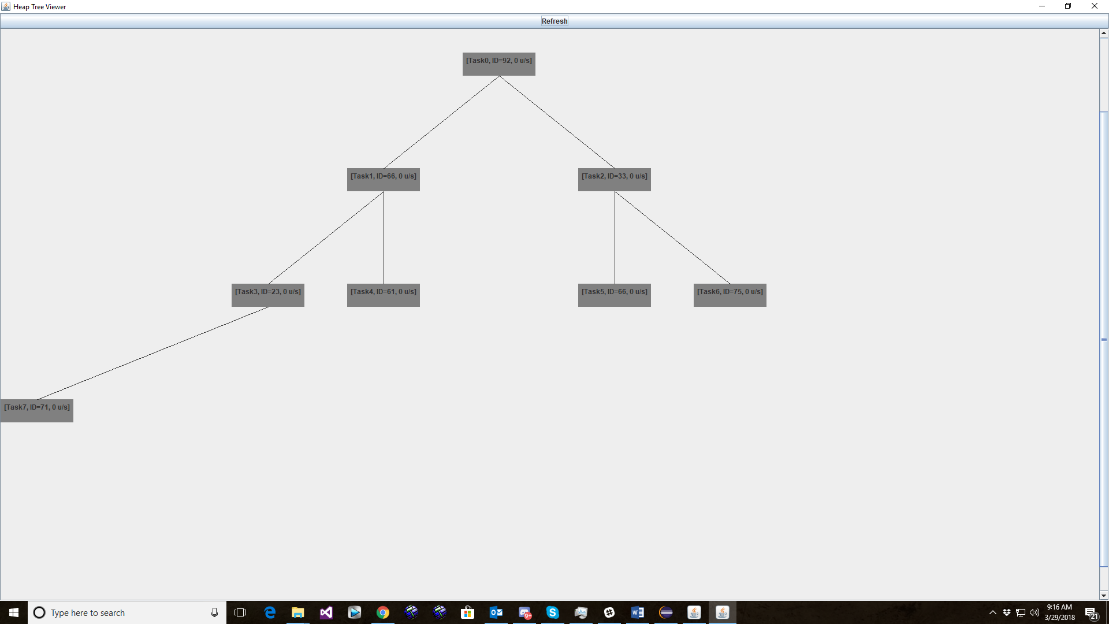
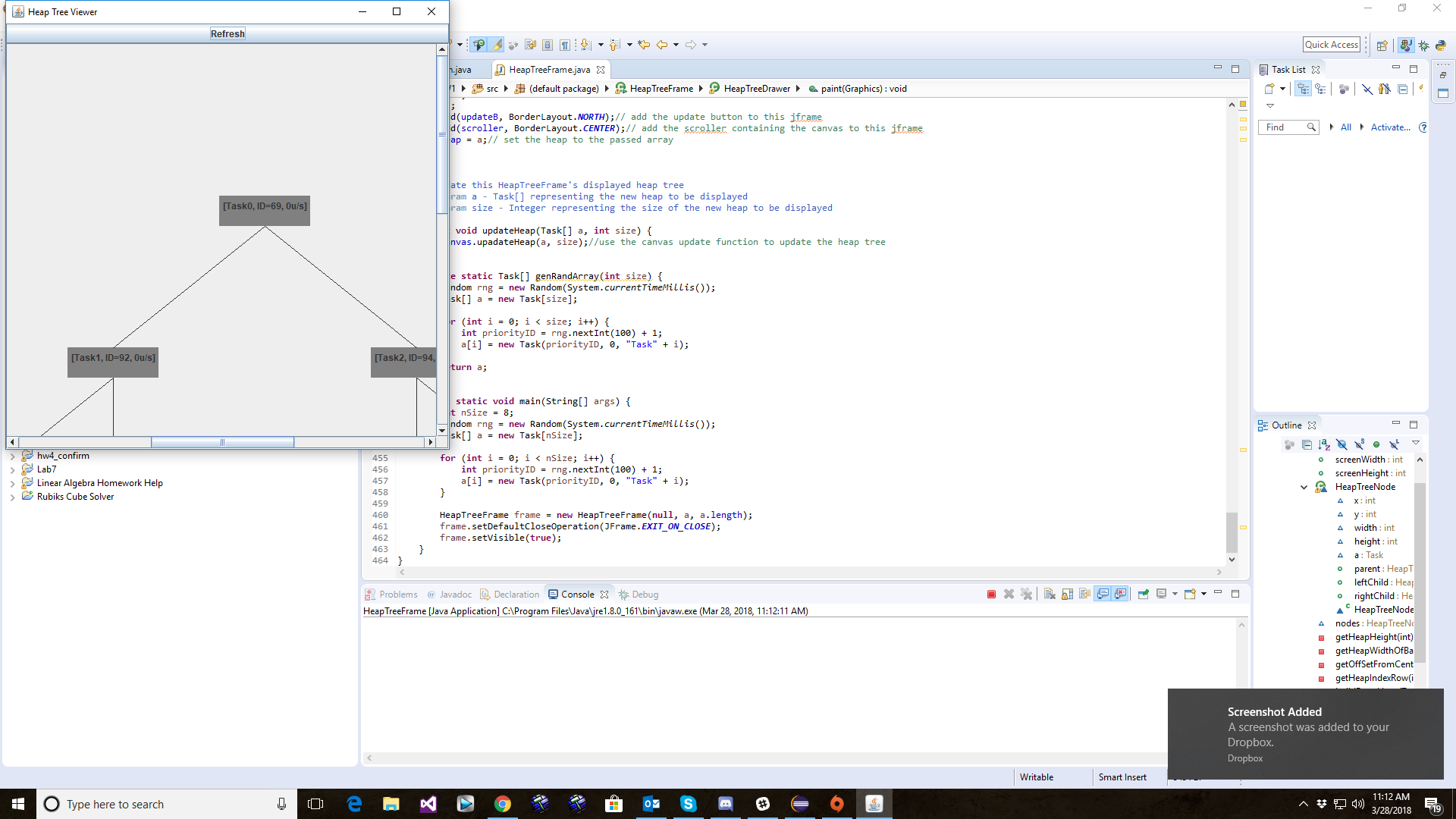
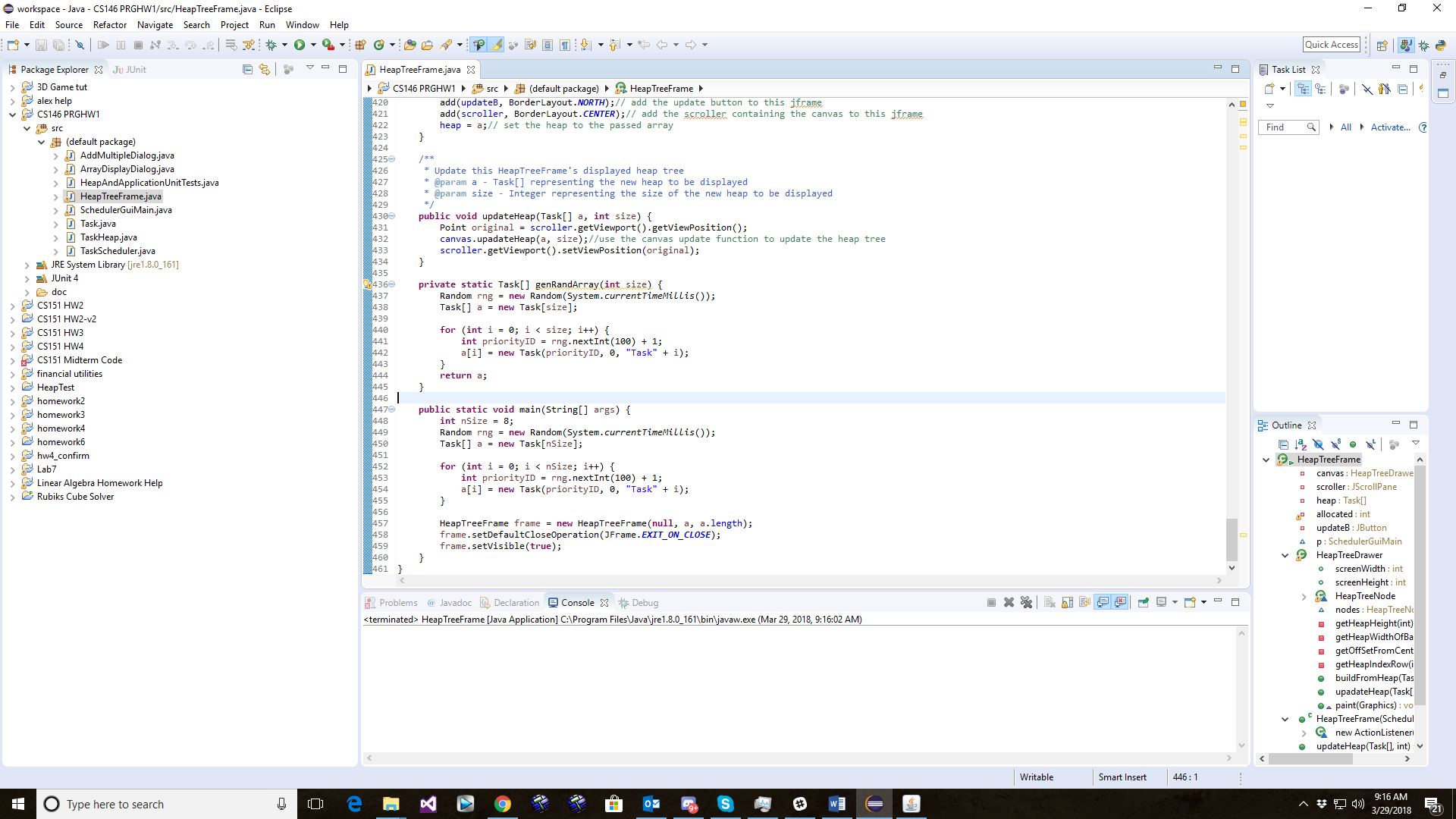


Output:

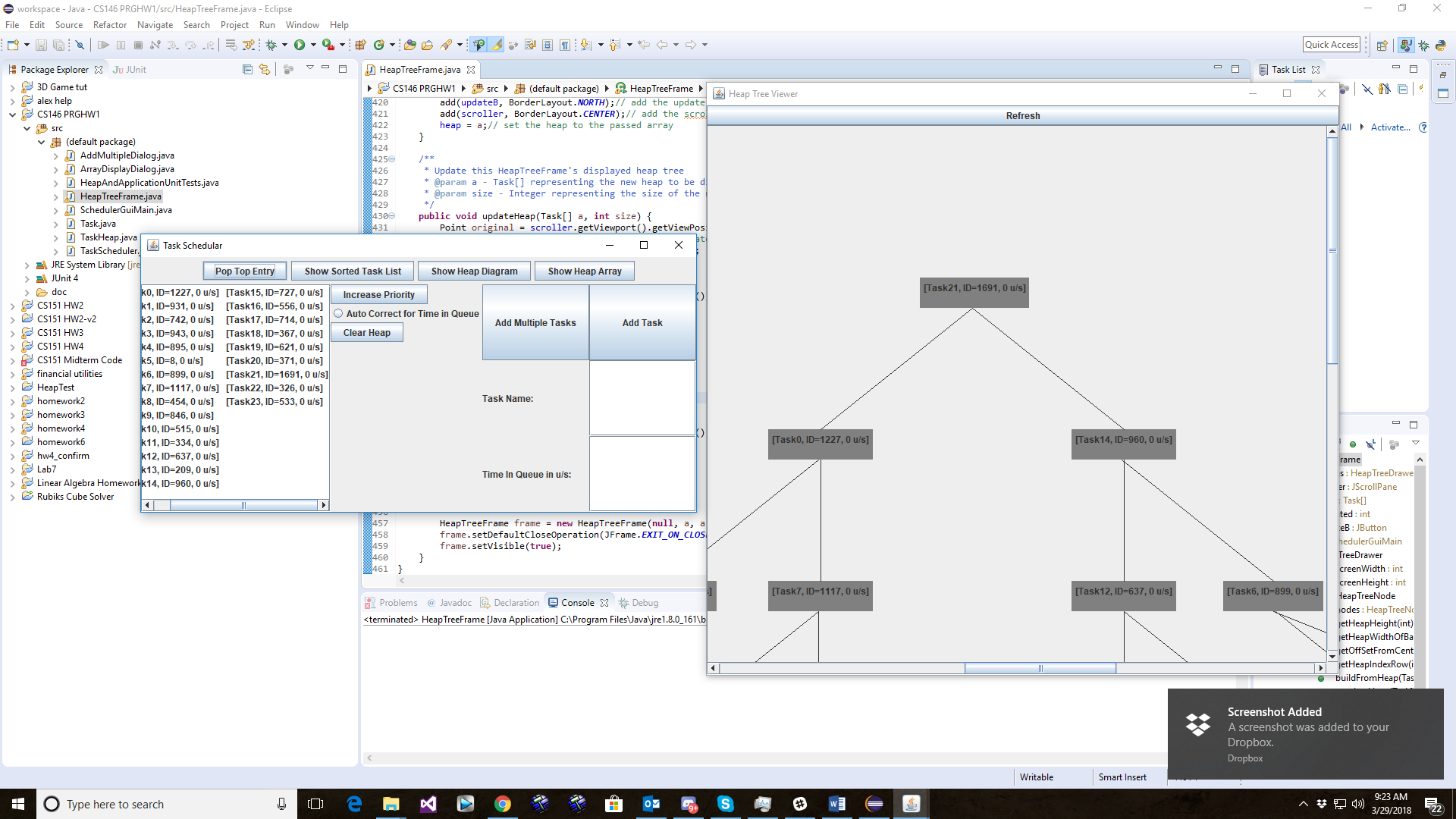


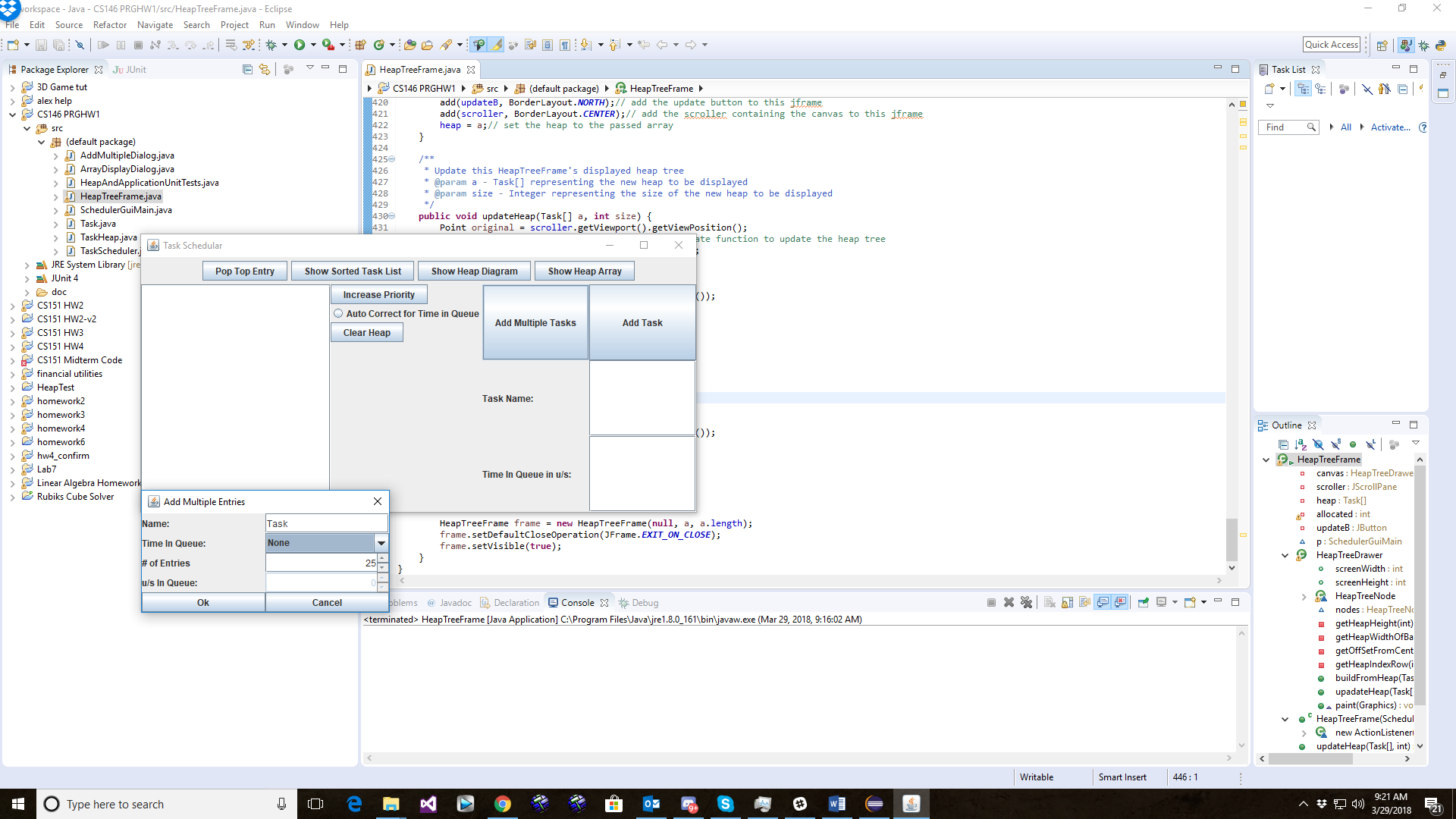
GUI Tests

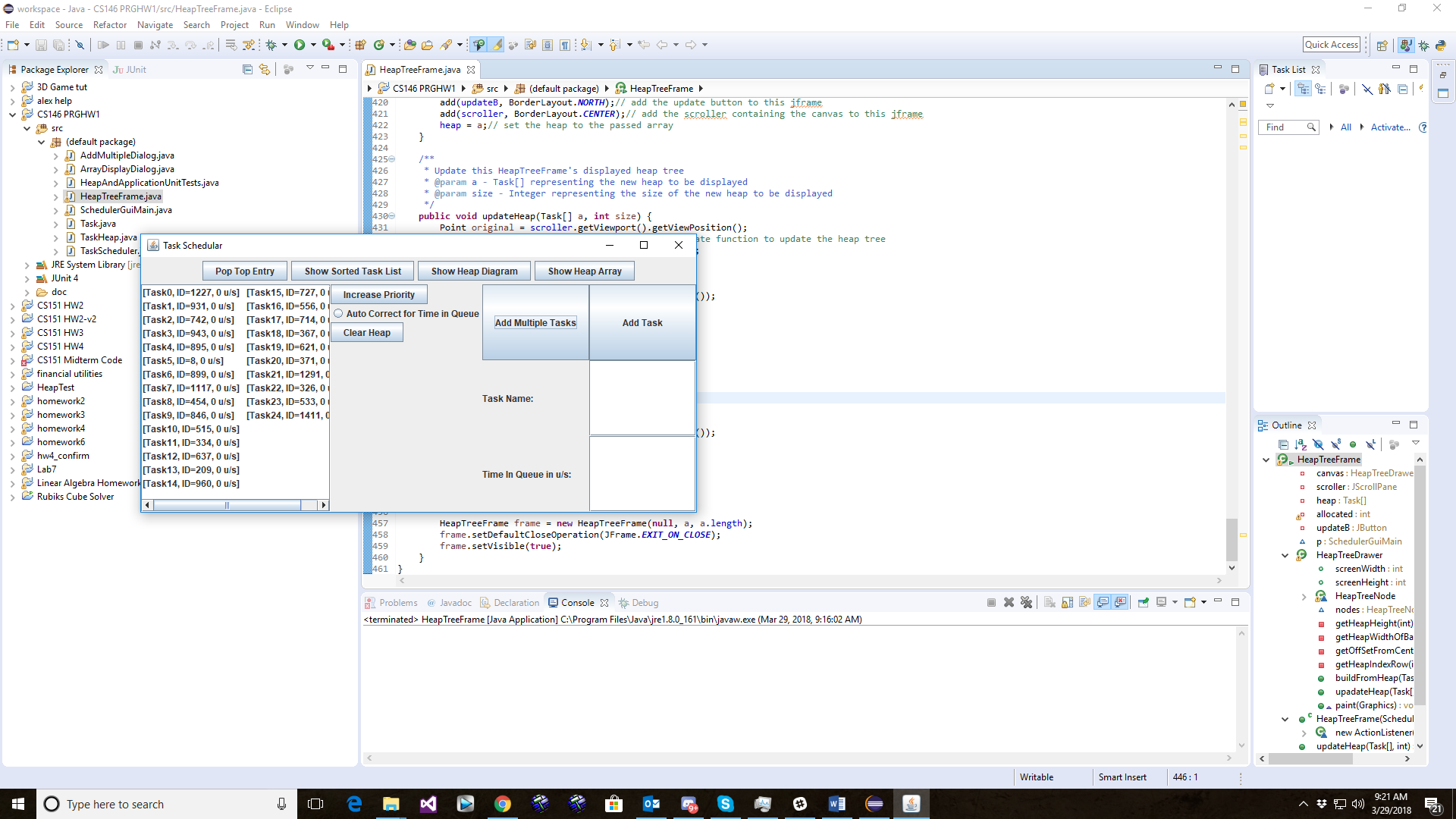
Heap Tree Viewer:

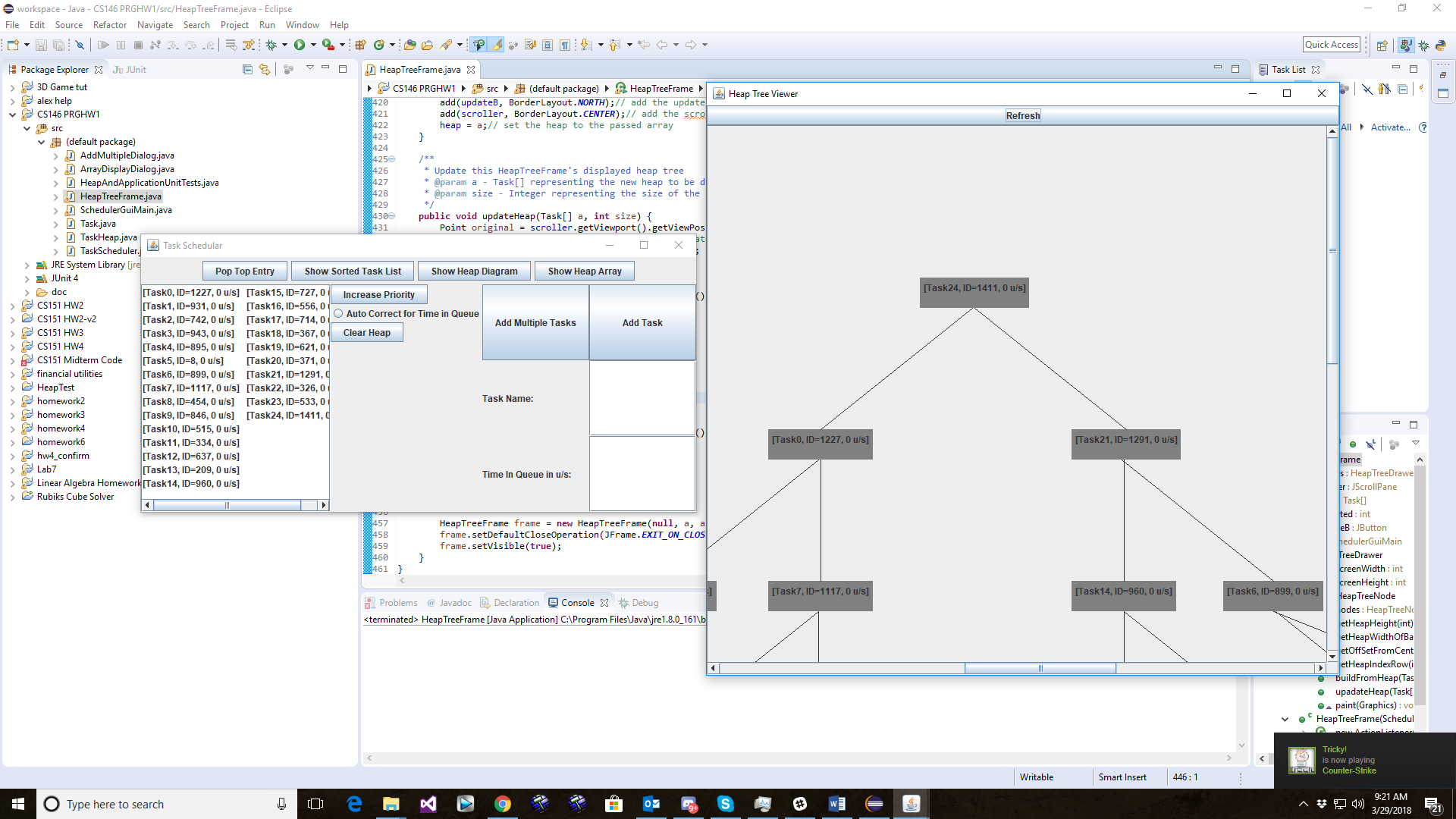


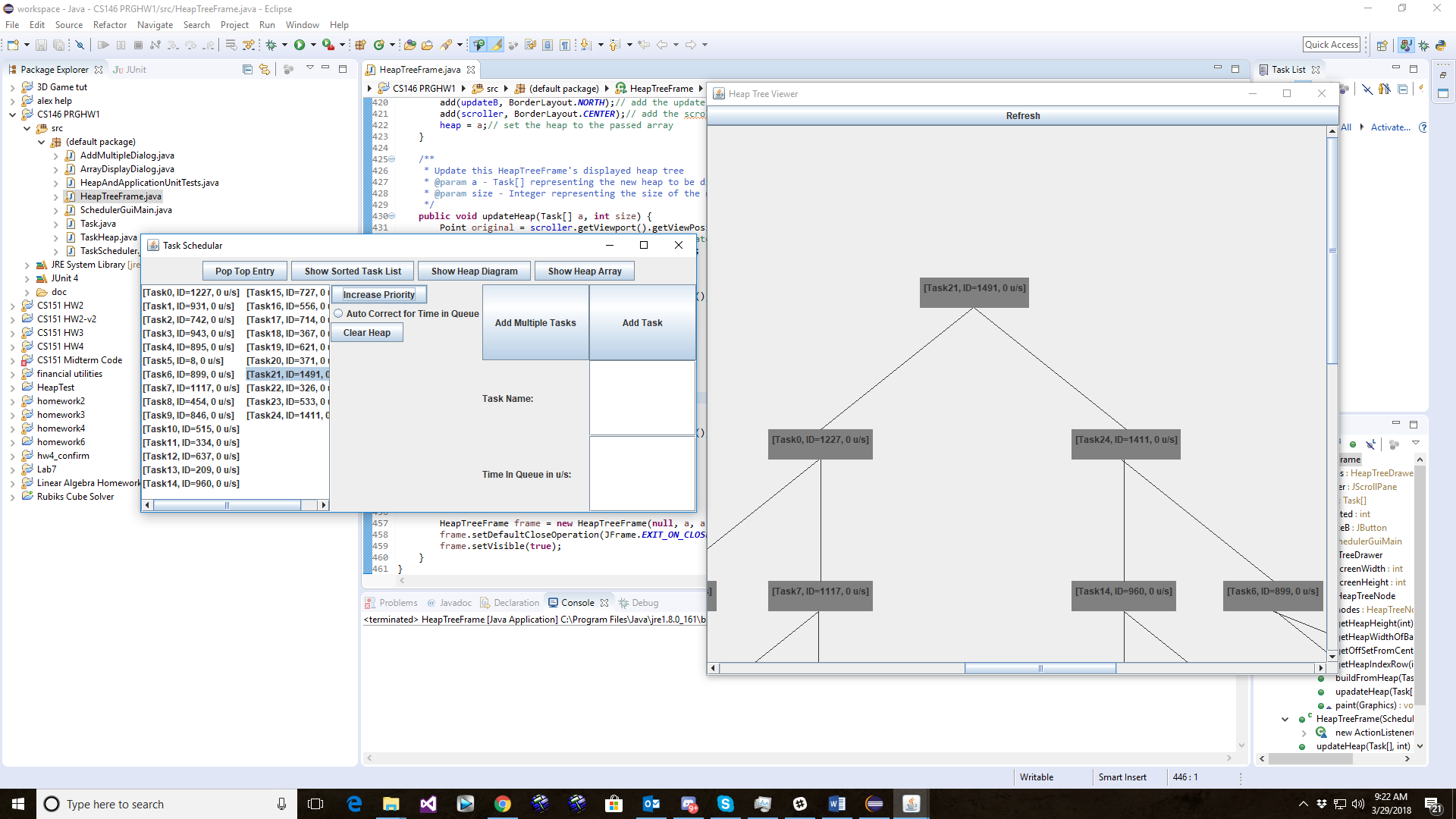
Main UI Tests:

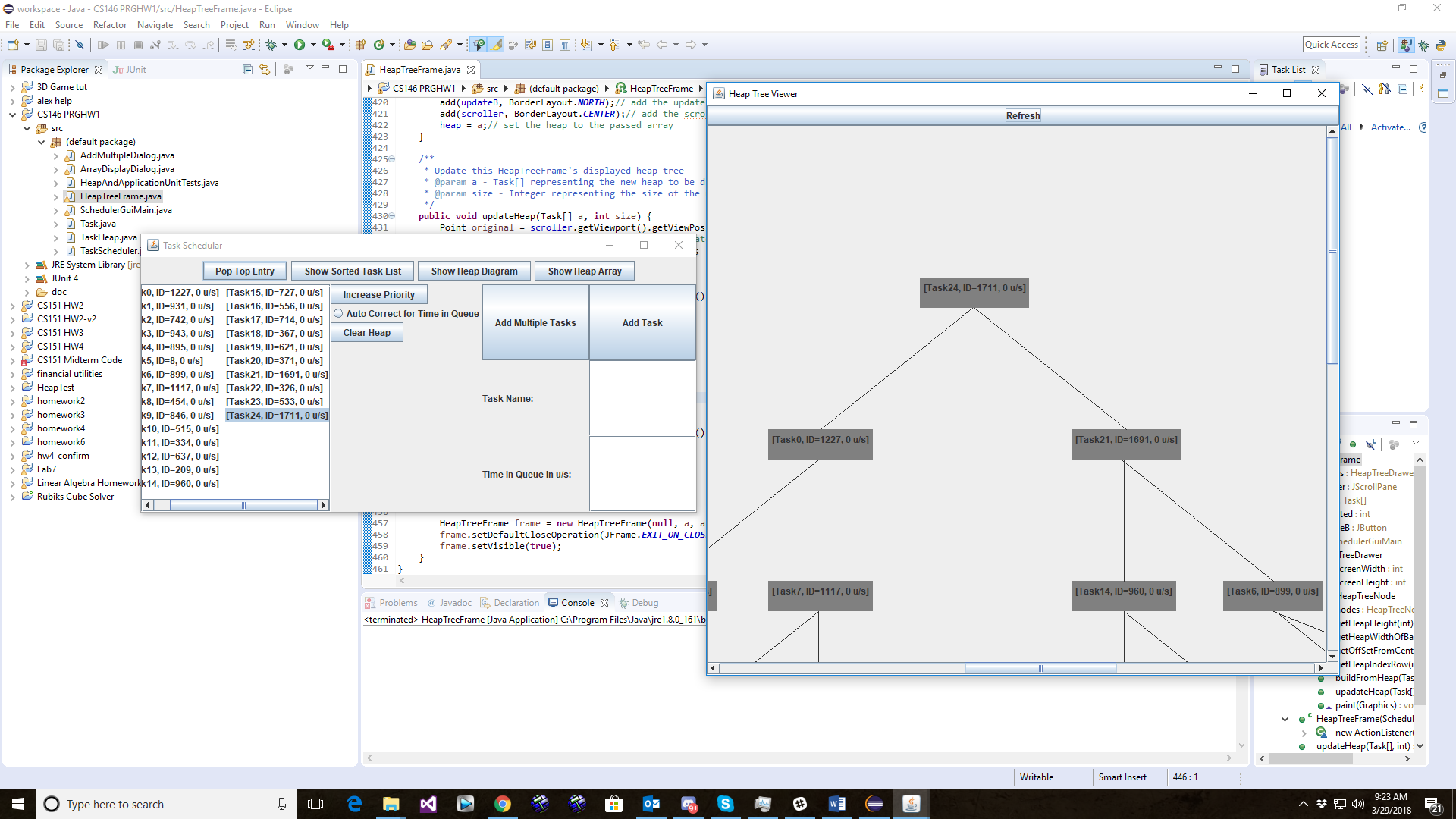














Conclusion – Lessons Learned

Properties of Heap

Over the course of my work on this assignment, I learned various properties of heaps. Some of the most useful exercises for me to do this were writing the methods for placing nodes in the HeapTreeDrawer class. I had to draw the tree structure of heaps and determine how to calculate information like offset from center, maximum width, height, and what row to place nodes on based on index. I used algebraic manipulation to derive the formulas for calculating this information.

GUI Programming

I learned a lot about the swing library while developing the application for this assignment. Before embarking on this assignment, I didn’t know what a scroll pane was, or a j-list. Creating the GUI for this application was challenging, especially the tree viewer. At first, I was not clear exactly how to go about placing objects in a j-panel. Eventually though, I learned that setBounds could be used to place a re-size objects manually. From there, the solution was clear: position node objects in a panel, then draw lines between them.

What I Would Do Differently

In retrospect, I wish I had employed better design patterns from the start of development. I should have broken up the responsibilities of the SchedulerGuiMain class into several classes; at this point in time it has the responsibility of keeping track of the heap, processing inputs from the user, and outputting information on the heap. I should have employed the observer pattern for the heap and the Tree View. Instead, I had to come up with quick fixes, like passing references to the main ui object and making methods and member variables public which would have normally been private or protected. Practices like these are dangerous, as they violate encapsulation and increase the possibility of data corruption in the class.