Team 16:
Christian Valladares
Joe Kallarackel
Fayaz Khan
Jaydeep Patel

Min-Heap Illustration Report

Project Overview:

As a self chosen team we were assigned a task to create a data structure visualization. The data structure must visualize and illustrate how our min heap changes. As a group, we thoroughly learned about our data structure. We chose to illustrate min heap as our data structure. When creating our program we kept in mind our audience, our target audience is CS students who intend to learn min heaps.

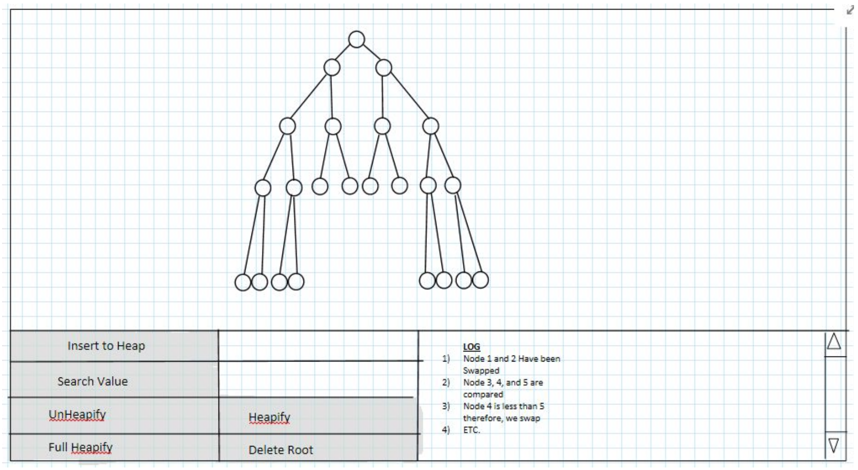
To effectively to show how min heaps work we focused on showing how the creation, insertion, deletion and search of nodes. Our program also has the ability of handling varying reasonable sizes. Our visualized tree is made to handle six levels to illustrate the numerous actions of how to work with a min heap tree.

The user has the ability control the basic operations and program illustrates the numerous changes happening at each iteration. For operations such as Heapify that require multiple steps the user is able to view changes happening one step at a time.

To illustrate our min heap tree we were given technical requirements for creating our Java application. We used classes, the benefits of using classes is encapsulation, abstraction, protecting data members, inheritance and polymorphism. To implement the min heap tree we used primitive types and arrays and used layout managers that were taught in class. Our GUI, is scalable with different reasonable window sizes using the ScalePoint class.

The ScalePoint class gives us the ability make all of our graphics resizable. The objects of this class carries the location on the screen as horizontal coordinates that are percentages of the width and vertical coordinates that are percentages of the height of the drawing space. The ScalePoint class, will hold all of the functionality for storing points and scaling them to the proper size in pixels. The ScalePoint class has public functions such as default and initializers constructors. We also have methods for setting either coordinate based upon its pixel coordinate value at the overall screen size in that dimension. Additional methods for getting the pixel value of either coordinate for a given drawing canvas width or height. We also added several other functions for cleaner, simplify and maintainability of code.

UI Overview:



LOG
1) Node 1 and 2 Have been Swapped
2) Node 3, 4, and 5 are compared
3) Node 4 is less than 5 therefore, we swap
4) ETC.

In order for us to achieve this user interface and organization, we nested different layout managers within others to tailor the GUI to our needs. The mechanism for us to nest these layouts was that of applying layout managers to a set of panels, and nesting these panels inside other panels containing layout managers themselves. A general overview of the layout hierarchy for are GUI is as follows:

- **1. Base : BorderLayout**
  - **1.1. North**
  - **1.2. Center**
    - **1.2.1. Top: FlowLayout**
  - **1.3. South**
    - **1.3.1. Bottom: GridLayout(1 ,2)**
      - **1.3.1.1. bottomLeft: GridLayout(4 ,2)**
      - **1.3.1.2. bottomRight: BorderLayout**
  - **1.4. East**
  - **1.5. West**

In more detail, *section 1* of our hierarchy contained a BorderLayout called base. This layout was the base of all of our layouts to allow for proper resizing to our widgets. This was particularly useful considering the fact that we used grid layouts extensively, and these do not resize easily (or not at all).   Of this layout, we only used the South and Center fields. The south field having full horizontal resizing and partial resizing vertically (Simply offsets the objects vertically), while the centerfield resizes completely. *Section 1.2.1* makes use of the Center portion of the base layouts by placing a panel called Top which employed the FlowLayout.   We use the panel *Top* mainly because it isolates *BorderLayout.Center* portion to a single call of a panel; this was handy because we sent this panel into our scaledPoint class in order to calculate node positions based on this panel size (width and height). This section is used primarily to draw our heap. *Section 1.3.1* is the *Bottom* panel and it employs a GridLayout of two columns and a single row.   *GridLayout*s in general are inconvenient because they don't resize; however that is why we nested it within a BorderLayout. The purpose of *Bottom* is to create a split between our message log which displays what actions are performed within our backend, and the set of buttons. *Section 1.3.1.1* is a panel called *bottomLeft* employing GridLayout of four rows and two columns. The purpose of *bottomLeft* is that of organizing the *JButtons* and *JTextFields* on the bottom left corner of the screen.   *Section 1.3.1.2* is a panel a called *bottomRight* which employs a *BorderLayout*.   The purpose of this panel is containing the message log,   and we decided to use a *BorderLayout* to be able to resize. The rest of the sections not explained are not used and therefore will not be addressed.

Code Map:

In our main UI class we extend JApplet and implements Action Listeners. As soon as you run our program, inside the init we have a message popup explaining how the program works. The program originally starts with dimensions of 640 by 400; it will resize properly because of our layout manager choices and because we are using scaled point class to resize. We then initialize the min heap. Next, we initialize the flags that keep track search success and button disabling. After this we call the following functions: initializePanels(), initializeWidgets(), setUpPanels(), addAllListeners(), addToPanels(), and finally addToGui(). The general purpose to these function calls is as follows: to implement a layout manager hierarchy to handle our gui through the use of panels. These calls also initialize our widgets and adds them to the GUI. Next after these function calls, we disable following buttons: heapify, unHeapify, fullHeapify, deleteValue, and at the end of init we create a new instance of scaled point.

Furthermore, inside the paint() function, we perform all the necessary operations. Paint is the most important function, and it does most of the work. It makes calls to the scaledPoint class and it takes care of drawing heap nodes and edges in its appropriate positions and sizes. In order to draw our nodes in clean, modular code, we employ drawNode, which takes care of the actually placing of the node along with color and computing the font size. We also update the the message log, and we finally handle the printing of the board whenever we have successfully searched for a node.

We also have actionPerformed that handles the event handling for all of our widgets. All of the backend calls are made from action performed, so anything pertaining to the heap--insertion, deletion, and balancing--are performed from actionPerformed(). We also take care of the error handling in action performed using try and catch.

Start the program:

- display Tutorial
- set up everything - by calling setup functions
- action performed- start the process and error handling
- paint - accessing the heap functions (from parent class)
- drawNode - using the graphics g