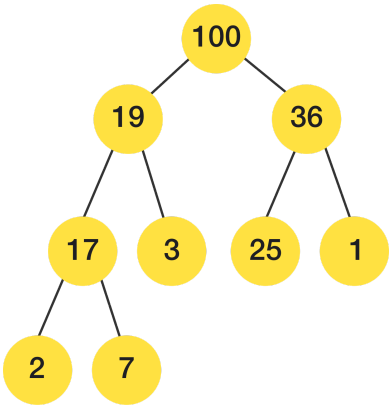
Assignment 3 Report

David Friedman

Introduction

Binary heaps are a data structure used to store data in a tree-like structure like the example below:



While abstract by design, these structures work in the background of many algorithms, including heap sort for sorting lists of integers, priority queues that prioritize specific elements based on defined criteria, and more. Consequently, there are many distinct ways to assemble a binary heap. For example, the above heap is organized so that each "parent" element is greater than its respective "child", and this is known as Binary Max Heap.

To broaden our understanding of how heaps work, we'll analyze how to assemble the Binary Min Heap, which is similar to the max heap, except the elements get bigger the further they are from the root (top-most element). We'll primarily design it to work in Python due to its commonplace usage, and we'll also include a basic visualization function for the tree.

Design & Difficulties

There's two standard ways to structure a binary tree: Either creating nodes with explicit links to their left and right children, or creating a simple array and using math to abstractly link nodes to one another. We practiced the former a good amount in class, so I opted to design my code around the latter. Once I defined my primary variables of **self.__heap** (list) to store my

elements and **self.__size** (integer) to store the number of elements in the tree, I created my getters for the parent and left/right children of any specific index in the tree. Originally I set these up to fetch the actual values of these indices, but I later changed these to simply retrieve the indices themselves as it made my later code more succinct. To retrieve these elements, the list is by design set up to ensure it always follows specific formulas based on the current index i:

- **Parent** - $\lfloor (i - 1) / 2 \rfloor$
- **Left Child** - $2i + 1$
- **Right Child** - $2i + 2$

Meanwhile, there were four distinct functions that I made to modify and display the tree:

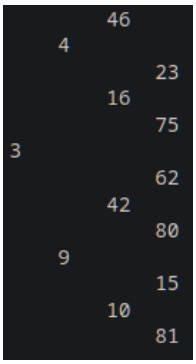
- **Insert(self, value)** - Adds a new value to the tree
- **Extract_Min(self)** - Removes the minimum value (root) from the tree
- **Display(self)** - Visualizes the tree using the code we discussed in class on April 30
- **Visualize(self)** - Visualizes the tree using code I came up with myself

For the **insertion** of a value, I first appended it to the end of the list and incremented the list size by 1. This alone obviously isn't enough as we need to move this element to the correct position such that its parent is smaller than it and both of its children (if it has any) are bigger than it. Thankfully, this is fairly simple as this only requires continuously swapping it with its parent value until the expected numerical relationship holds. I pulled this off with a basic **while** loop that ended as soon as either no parent node existed (meaning our element was swapped to the root) or the parent node was smaller than our element.

Coding the **extraction** function was more challenging and it took longer for me to get it to work without hitting a TypeError or an out of bounds index violation. First I created the trivial case where if the tree size is 0 or 1, then an empty list of size 0 is automatically returned. Otherwise, the first and last elements get swapped, and then the former root gets removed. Following a similar idea as the insertion function, the new root swaps with children values that are smaller than it until it either has no children or all of its children are bigger than it. This took a bit to properly implement as I was using a **while True** loop reliant on the **break** operation to end it, and at one point I had to deal with my loop not breaking.

Next, the **display** operation was very similar to the one we practiced in class. It displays our binary min heap sideways such that each element is printed on a different line that is more

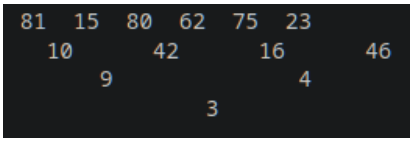
spaced from the left depending on how far from the root it is. Using a helper recursive function, it prints elements similarly to an **in-order traversal** except it traverses the **right** children, then the **root**, and then the **left** children. This effectively prints every element of any tree such that they are visibly spaced apart equally thanks to them being on separate lines:



Lastly, my custom **visualize** function sought to find a better way to display the tree. I wasn't very fond of the fact that the **display** method relied on recursion as if our tree is big enough, some computers might refuse to run the function due to the possibility of a maximum recursion depth error. So instead, I wanted to not only display the binary min heap **vertically**, but also use **iteration** since that's less likely to outright fail due to complexity. My design first computed the number of levels in the tree using the following function where n is the total number of elements in the tree:

$$L = \lfloor \log_2(n+1) + 1 \rfloor .$$

Using this, I made a **for loop** to put all the elements in each level into lists within a list. For example, if **self.__heap** is [1,2,3], the resulting level-storing list would be [[1],[2,3]]. The next step is then to use some clever empty spacing to print out the elements of each sub-list in a pyramid-like manner. Experimentation led me to conclude that starting at the deepest level, if each element has 1 empty space between them (" "), then each k level closer to the root has 2^k additional spacing added to the previous amount of spacing. Lastly, the amount of spacing to get the left-most printed element to the correct spot is the **half** of the between-element spacing rounded down to the nearest integer. With the numerical formulas sorted out, I went ahead and coded the **visualize** function, though I ended up printing the tree upside down so that spacing computations would be simpler. An example output of this follows, which I'm satisfied with:



Results & Reflections

To make sure these operations worked as intended, I experimented with varying amounts of inserts with random values. I started off inserting the values in ascending order to establish a baseline of functionality before having the minimum values come later. Meanwhile, I made sure the intended structure of the binary tree was preserved in both cases after applying the extraction function. This testing did draw my attention to my aforementioned type errors, out of bound index errors, etc, so this was effective in helping me refine my code.

Despite my success in getting this implementation to work, I'm not as sure of whether or not a list-based implementation is preferable to a node-based one in terms of coding complexity. I did enjoy the more mathematical aspect of the list-based version, but I feel as if a node-based implementation might have been less time-consuming to code due to the requirement to explicitly connect nodes to one another and the fact that I could attach information like their level in the tree to them. At the same time, Python already has built-in functions for managing lists while I have to create those functions from scratch when creating nodes, so I'd be interested in learning if it's reasonable to do both types of implementations at once so that I could enjoy the respective benefits of lists and nodes.