Assignment 3: Binary Heap

Jason Hlavac

Git-Hub Project Link:

**Introduction:** For this assignment, I will be implementing a max binary heap using a linked implementation. For the max heap, each node will have at most 2 children nodes. The root, or highest up node in the list will be the highest value in the list(the max). If you go to either of the values that are the children of the root, you will have another part of the tree that will follow this same rule. In other words, no child is a higher value than its parent.

**Node Class:** The building block of the binary heap will be Node objects. Each node will have a left and right pointer that will be of type node. I will also need a variable for the value of the node which will be an integer. I am also adding a parent reference. This will point to whatever node is above the current node. This one is not 100% necessary, but it will make some of my later functions easier. The functions for this class are very simple. It will just be the getters and setters for the variables that I have because they will be private.

**Binary Heap Class:** The second class that I need to make is for the Binary Heap itself. The Binary Heap class will be where the actual functions for the operations on the heap will be defined. The only variables that we need defined for this class is a pointer to the head of type Node and an integer for the size of the heap. The head node will then be the access point for the rest of the heap and the size will be used for keeping track of how many elements are in the heap. In a max heap, this value should be the highest one in the heap. I will have a getter and a setter function for the head as well. The rest of the class will be operations on the heap that I will talk about below individually.

**Insert:** The first operation that I am making is an insert function. This operation will take in an integer. This integer will then be fit into the heap as the value of a node. It will be important to make sure that when it is inserted, it follows the rules of the max heap. The easiest case for this function is when the heap is empty. For this, I will check if the head is null. If it is, then just assign the head to the new value. For all other cases, I will call the get\_next\_slot function that I talk about below to find where to insert. When I have navigated to the node that I need, I set the left reference to the new node if it is open. If it is not, I assign the right node to the reference. I also then set the parent of the new node to the node that I navigated to. This is just for putting the new node into the next open slot into the list. To put it in the correct spot so that it obeys the properties of the max heap, I call my percolate up function that I talk about below on the new node.

**Get Next Slot:** A helper function that I am using for my insert is one that finds the next slot that is open for a new node that is being inserted. This function uses the number of nodes in the heap to navigate to the place where the next node needs to be inserted. It is a recursive function that starts at a node and navigates either right or left from there. The node that you navigate to leaves another smaller Binary heap that you can do the same for until you reach a node with an empty slot. The function takes in the size of the remaining Binary heap(everything under the current node), and the current node that you are on. First, the base case for the function is to return the current node you are on if either one of it’s references are null. If this is not satisfied for a call on the function, you need to determine if the next node you pass should be the right or the left one. The first thing I do is get the height or number of levels by doing floor(log2(n))+1. We know that the time complexity of things like merge sort are O(n log(n)) where log(n) is the height because each “level” has twice as many sub-lists as the previous, so this makes sense. Next I get the number of nodes that are not on the bottom level. Because the binary max heap must fill up a level completely before going to the next, we know that every level above the bottom is full. This allowed me to set up a for loop that generates a geometric sequence. For height count, I add 2^i to my not bottom count(1 + 2 + 4+ 8…). With this, I can get the number of elements on the bottom level by subtracting the not bottom elements from the total elements. Next, I calculate the maximum number of elements that the bottom layer can hold(2^height). I use this alongside the number of bottom elements to decide if I should go right or left. If the number of bottom elements is less than half of the maximum number of elements for the layer or equal to the number of elements for the layer, I go right. Otherwise I go left. The last step of the function is to make a recursive call using the new current node and the new size of the heap given that everything below the current node is the new heap. To do this, I take the floor of notBottomCount/2. I take the floor because each level is cut in half except the one that you are currently on. The one that you are on always appears to only have 1 node, because it is the head of the sub-heap. I then add the bottom count% the maximum number of elements for the bottom layer/2 ((2^height)/2). This gets me the number of elements on the bottom level within only the half of the heap that is being passed.

**Percolate Up:** The percolate up function is another recursive function that is designed to make sure a node abides by the rules of a max heap. It takes in a Node, curr, as its parameter. The base case for the function is when the current node is less than it’s parent. When this happens, the Node should be in the correct spot assuming that all the other nodes had this function called on them when they were added. If the current node is out of place, it is swapped with its parent, using the swap function I talk about below. We also need a check for if the current node does not have a parent(the head node), because otherwise this will raise an error. We also know that if the node is at the head slot, it cannot percolate up anyways. We put this check at the beginning of the if statement for our check and make it an and statement, so the rest of the if statement will not run if this is false. A recursive call is then made to the function with the new location of the Node that was originally called to the function.

**Swap:** The Swap function is a simple function that swaps only the values of two nodes. It takes both the nodes as parameters that need to be swapped. The first step of the swap is to store the value of the first node in a temp variable. The value of node 1 is then set to the value of node 2. Node 2 is then able to be set to the value of the temporary variable. By swapping only the values, it is very easy to keep the structure of the heap and we will not need a special case for things like the head of the list.

**Extract Max:** For extract max, we need to remove the head of the list. The easiest way to do this is by swapping the values of the head node with the value of the last inserted node, deleting the last node added, and then calling swapping the head node value down until it is in the right spot. First, I will start by finding the last node added. To do this, I will repurpose the get next function that I made earlier. I will call the function starting with size-1 so that instead of finding next, it finds the last node entered. I then check if the right node is filled first then the left node because we fill the heap from left to right. Next I call the swap function that I made earlier on the head and last. After this, I remove the pointers to the last node by cutting off the left or right pointer to the last node, and removing last’s parent. Next I call percolate down on the head which I will explain below.

**Percolate Down:** For percolate down, it is only slightly harder than percolate up. We need to see if there is a child of the node that is greater than our node. If there are two children on a Node, then we need to compare both of them and see which one is larger to know which one to switch. The function will be recursive and will call itself again if a node is swapped, this time with the new position. There are a lot of different conditions on this function to make sure that an error is not raised by trying to get the value of a null. I also made a decision here that if the two nodes that are being compared are equal, I will use the left one.

**Display:** The display function that I used was almost exactly the same as the one that we made in class for the UBST. I only had to switch variables names so that it fit my new code. This function displays the tree from the top being on the left and going to the right.

**Testing Process:** I ran into a lot of errors throughout the making of this structure. The most common way that I tested was by using print statements to pin point the line or chunk of code that was causing the errors. Many of my mistakes involved calculated values not being what I expected them to be, so printing them helped as well.

**Problems Faced:** The biggest problem faced by far was designing the Get Next Slot function. When we learned about this data structure, we looked at pictured in which it was very simple to look at the heap and see where the next slot to add is. This is much more difficult in a linked implementation of a heap because you can only ever see the right and left of the node that you are on. Designing this algorithm took a lot of time, so I’m glad I could repurpose it for one of my later functions. Another problem that I ran into is I often had to do quite a bit of coding before I could actually test the code, this made small errors harder to find.