Brendan Bovenschen
CWID: A20323473

CS3353 Homework 3 Program - Min/Max Heap

Introduction

The advantage of a min-max heap over a traditional min or max heap is that it is able to reach both its minimum and maximum value in a constant time, while still maintaining the same running time of common operations such as ExtractMin/Max and Insert. With each new layer, the bounds of the previous layers combine to create a smaller window for which elements can exist causing the median elements to be on the lowest layers and the highest/lowest at the top of the incomplete binary tree.

Build Heap

The process of building a min-max heap contains multiple subprocesses for min and max layers. Just like the max heap studied in lecture, we begin by heapifying downward from the middle element to the head element. This ensures that the set is sufficiently traversed for proper heapification. Since the min-max heap contains two different layers, two methods of heapifying downward must be used: one for a layer containing minimum elements, and one for maximum elements. Each of these functions similarly, but with opposite comparisons. To create this pattern, we must look at the children and grandchildren of the current element. First, we find the smallest (largest) child or grandchild of the element. If the smallest is a child, we will swap these elements and terminate the subprogram. If this element is a grandchild, we will still swap them, and then determine if the parent of the grandchild is larger (smaller) in value. A swap occurs if this is the case. The heapify downward is continued from this point till the smallest element is the entered position.

To determine the worst-case running time of the heap, we must first look at the heapify function. This function has a worst-case running time of O(log(n)) as it traverses at most log(n) elements. The build heap function will always heapify downward a total of $\frac{n}{2}$ times, resulting in a final worst-case running time of O(nlog(n)).

Minimum

Each layer of our min-max heap binds the next based on its min-max value. This means that our absolute minimum value will always be in position ‘0’ in a sorted min-max heap. Getting this value is as simple as reading the position ‘0’ in the array with a constant worst-case running time of O(1).

Maximum

Similar to the minimum, we can find the maximum value of the heap in either position ‘1’ or ‘2’ due to the layer binding of the min-max heap. With one comparison, this value can be found with a worst-case running time of O(1).

Brendan Bovenschen
CWID: A20323473

Extract Minimum

Since the minimum value will always be at position ‘0’, this value can be deleted in the array with each element being shifted down one position and the heap size can be decreased by 1. From here, we must heapify down from the new element in position ‘0’ to re-heapify the list. Through the heap, log(n) comparisons are made to terminate the heapify down process, which would make the worst-case running time O(log(n)).

Extract Maximum

Like the minimum, we can borrow the same process of removing the maximum element and downward heapifying the list. This function also holds a worst-case running time of O(log(n)) as it functions similarly using a downwards heapification.

Insert

Unlike a min or max heap, the grandparent and parent of the inserted element must be compared. To begin our upward heapification, we must first check if the element is on a minimum or maximum level. Our first comparison will determine if our element is greater or less than its parent. This will swap our elements if the element on the min level is greater than the max, and heapify up the inserted element (HeapifyUpMax if in max level, HeapifyUpMin if min). The min and max heapify up functions will check the grandparent of the current element, swap depending on if the element breaks the heap rule, and terminate when the current element has no more grandparent. The heapify up function will require log(n) comparisons by traversing up one branch on the min-max heap. Therefore, the worst-case running time would be O(log(n)).

Example

This example can be found in the user interface of the program. Entering the array,

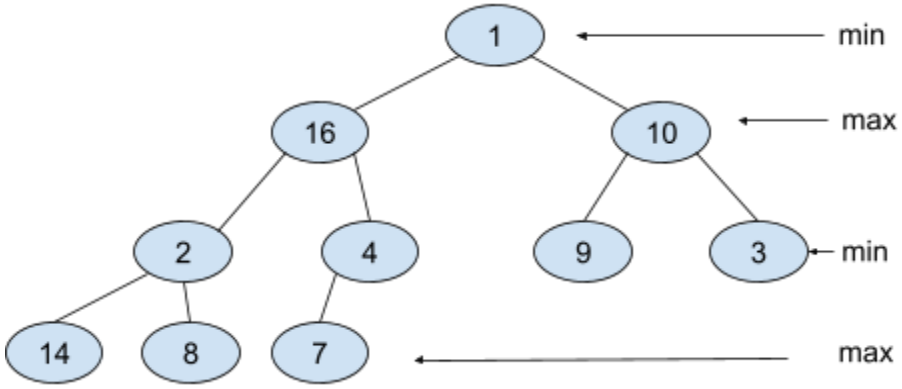
$$\{ 4, 1, 3, 2, 16, 9, 10, 14, 8, 7 \}$$

will result in an output array of

$$\{ 1, 16, 10, 2, 4, 9, 3, 14, 8, 7 \}$$

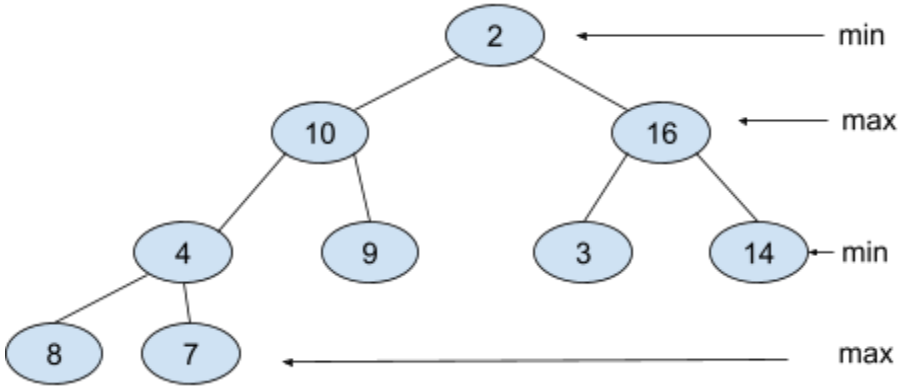
This can be viewed as an incomplete binary tree with alternating minimum and maximum levels as shown below. Here we can see that each layer binds the value of those below it, making values close to the mean near the bottom and polarizing values in the first two layers.

Brendan Bovenschen
CWID: A20323473



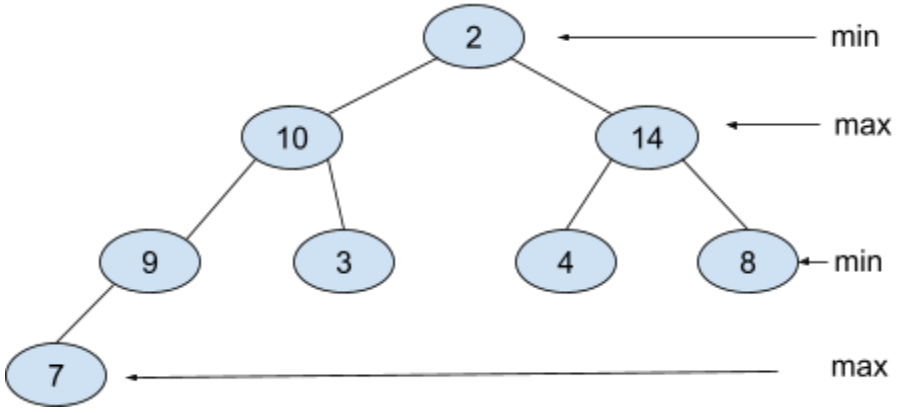
Next, we extract the minimum value of 1. 1 is omitted, and the rest of the array is heapified from position 1, maintaining the properties of the heap.

{ 2, 10, 16, 4, 9, 3, 14, 8, 7 }



Next, we extract the maximum value of 16. This process is similar to that of ExtractMin, and the heap is again heapified.

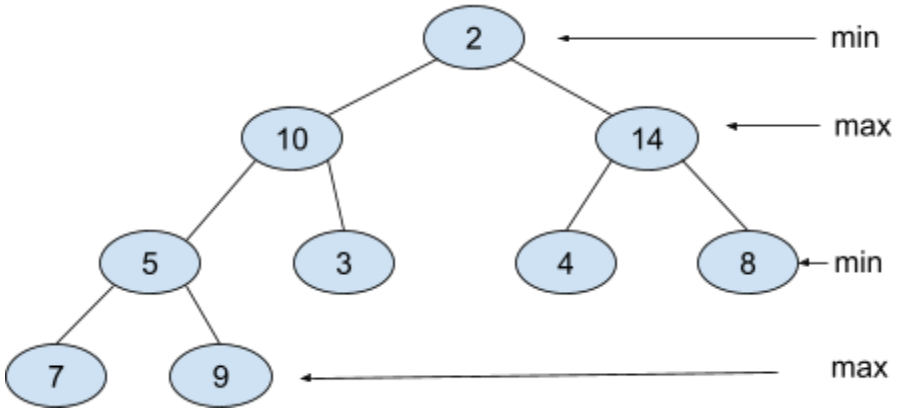
{ 2, 10, 14, 9, 3, 4, 8, 7 }



Finally, we insert ‘5’ into the min-max heap. 5 is initially in the last position of the list, and is swapped with its parent in this case.

Brendan Bovenschen
CWID: A20323473

{ 2, 10, 14, 5, 3, 4, 8, 7, 9 }



These processes are confirmed by our console, outputting

```
Type 0 for example, 1 to create an MMHeap, other to quit: 0

Input Array: 4 1 3 2 16 9 10 14 8 7
Min-Max Heap: 1 16 10 2 4 9 3 14 8 7
Min value '1' extracted, MMHeap: 2 10 16 4 9 3 14 8 7
Max value '16' extracted, MMHeap: 2 10 14 9 3 4 8 7
Inserted element '5' into MMHeap: 2 10 14 5 3 4 8 7 9
```

GetMin and GetMax are used to obtain the min value and max value printed in the console.