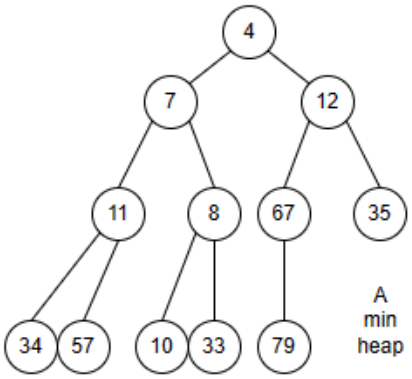
Lake Forest College
CSCI 317 Data Structures and Algorithms
Spring 2023



Benefits of Min Heap Illustrated with DeleteMin Algorithm

Maria Petrova

Benefits of Min Heap Illustrated with DeleteMin Algorithm

Maria Petrova

June 1, 2023

1 Abstract

This project seeks to examine the "min heap" data structure and illustrate some of its advantages by examining how it works with an algorithm meant to delete the minimum value from the data set. The programming language used in the implementation is Java, and the data type used in the structure is Java's Integer class. The heap in this project is implemented with the help of the Java ArrayList class, with the 'nodes' being identified by their index.

This project also discusses two versions of the delete-min method—one for the heap implementation, and one for a regular array implementation. It can be concluded that a min heap has many benefits and is often more useful to a programmer than a regular array. This is obvious when examining the differing time complexities for the delete-min algorithm, although there are many other factors that contribute to this statement.

2 Introduction

There are many different data structures used in the programming world, each with its own advantages and disadvantages. Perhaps the most simple is the array, which can be visualized as a table containing values. An array must first be declared (created) then instantiated (filled with information), just like any other attribute in Java. One of the disadvantages of arrays is that their size must be set when the array is instantiated, and this size cannot be changed later. Elements of the array can be changed or accessed by referencing their index, which is like their address. The first element is stored at index 0, and the last element is stored at index $n-1$, where n is the length of the array. However, elements cannot be added or deleted, since this would change the size of the array. This leads to a waste of memory storage when creating an array when the number of 'used' or 'filled' elements is smaller than the length of the array, especially in extreme cases.

However, Java has many classes within its utility library that can do what arrays do, but better, and these classes can be used to create other data structures that do even more work. For instance, the ArrayList class can store values the same way that an array can, with elements being accessed or updated by their index. However, ArrayLists are dynamically sized, which means that they can expand or contract as necessary. Elements can be added to the end, inserted at a specific index, or removed. It is also possible to check an ArrayList to see if it has a given element, and then check the index of a that element. An ArrayList can be used to implement many other data structures, one of which is the heap.

3 Min Heap as a Data Structure

When talking about data structures, a graph is an arrangement of vertices connected by edges. A tree is a type of graph, where the vertices are called nodes and there are no loops or cycles between the nodes. A binary tree is a special kind of tree where each node has exactly three vertices extending from it, except for the root node which has two vertices. These vertices lead to other nodes, which may store values or be null [1].

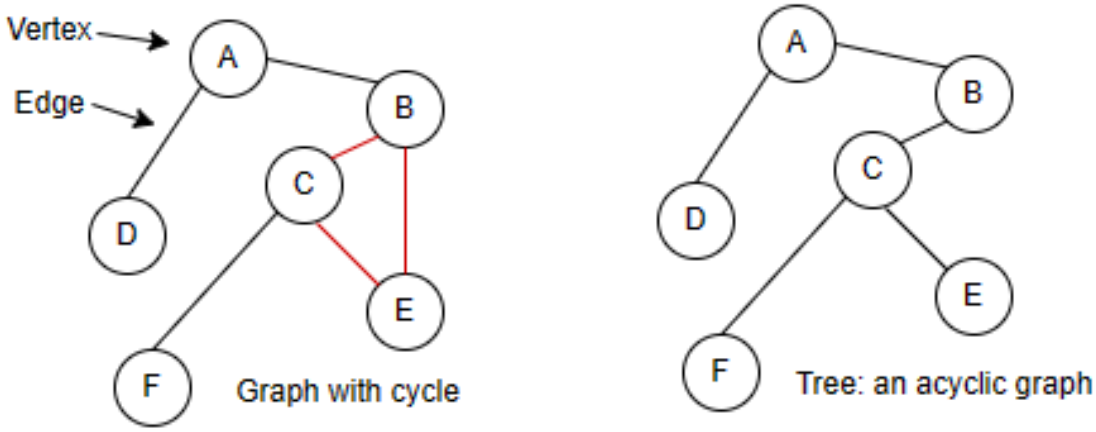


Figure 1: An example graph and tree.

As seen in Figure 2, a binary tree can be organized into a triangle shape, with the root node at the top, and the children extending down below. A typical node of a binary tree has a parent and at most two children (the root has no parent)[1]. A 'leaf' in a tree is a node with no children, whose vertices can be imagined to lead to null. Each regular child node of a tree can be considered to be the root of another subtree. There are many different characteristics that a binary tree can have, and one of those is completeness. A complete binary tree is "full" in the sense that each node has exactly two children at each level, except for the leaves, which have none. This means that a complete binary tree, at any given moment, contains $2^k - 1$ nodes for some k (k being a positive integer). The complete binary tree's height will be equal to k, with the root node having a height of 1. The leaves of a binary tree will be found at this height k. Non-complete trees also have heights, though they cannot be computed so easily [2].

A heap, such as the one in Figure 3, is a nearly complete binary tree. A heap of height n may not have the full $2^k - 1$ nodes, which means that some of the leaves will be found at height $k - 1$. However, insertion into the heap is designed to bring the tree as close to completeness as possible. While heaps are not as rigidly ordered as data structures like binary search trees, heaps do have an order property. There are two types of heaps: minimum heaps and maximum heaps. A min heap has the minimum-valued node as its root, and the children of each node have a greater value than their parent does. In contrast, a max heap has the maximum-valued node as its root, and the children of each node have a lesser value than their parent does. This project takes a closer look at the min heap.

[1]Some interpretations include a blank node before the root, giving it a parent of sorts and thus ensuring that every node has exactly three edges.

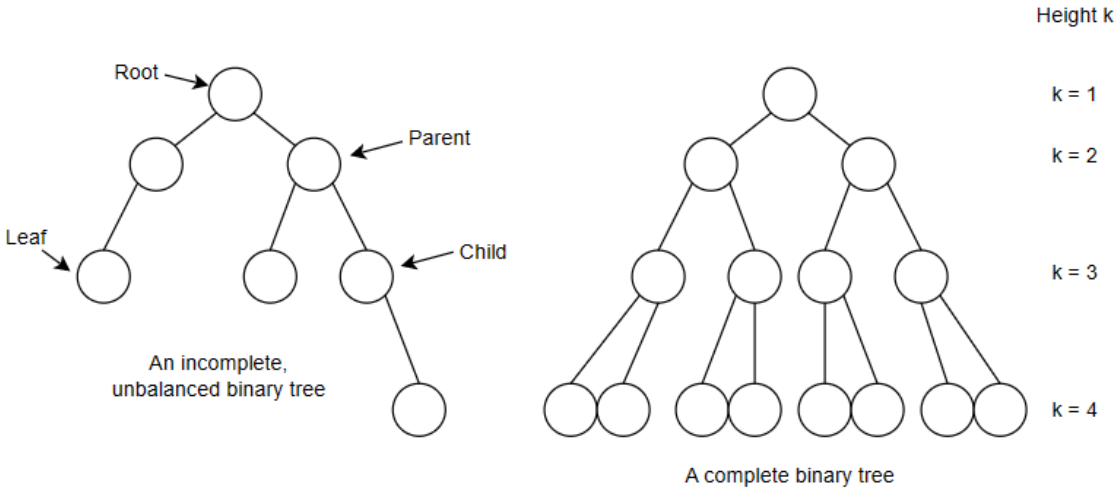


Figure 2: Complete and incomplete binary trees.

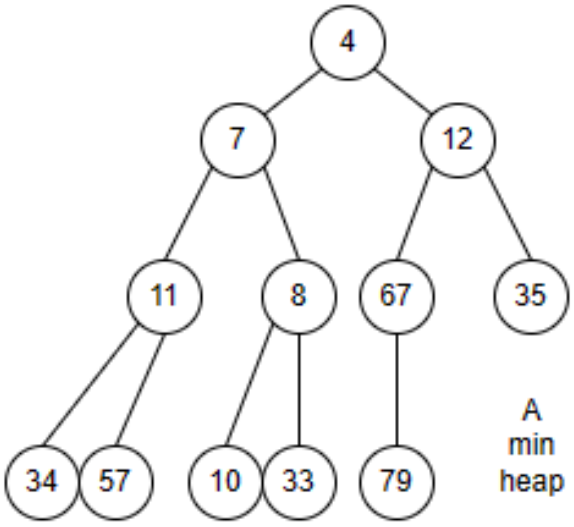


Figure 3: An example of a min heap with 12 elements.

Insertion into a min heap is meant to preserve the heap order property. If a value is entered into the "wrong" node, that node swaps values with its parent to keep the order. Consider, for instance, the sequence of values 11, 7, 67, 4, 8, 12.

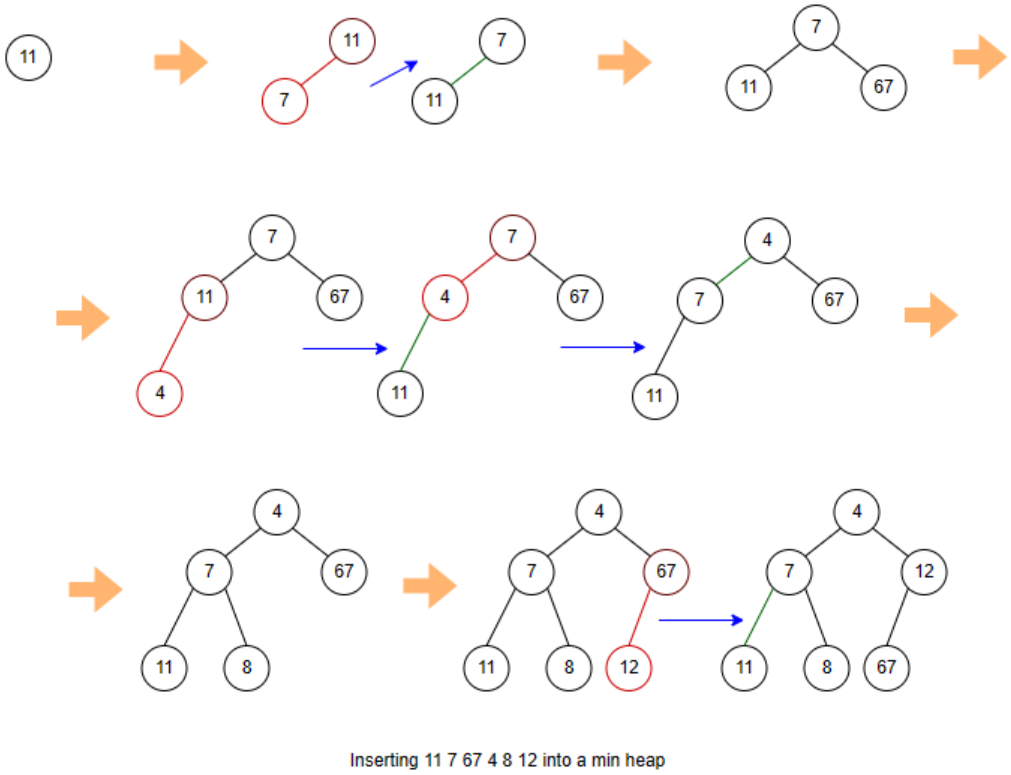


Figure 4: Insertion into a min heap.

As visible in Figure 4, insertion requires swaps sometimes, but not always. When 11 is inserted as the root node, there is no disruption of the heap order property. Then, 7 is inserted into the next spot. Since 7, the child of 11, has a lesser value than its parent, the two nodes switch values. Next is 67, which is inserted without any issues. However, when 4 is inserted into the next node, it needs two swaps in order to preserve the heap order property. 4 is less than 11, so they swap. But 4 is also less than 7, so the two also swap. Afterwards, 8 is inserted with no problems. The final swap of this data insertion happens when 12 is inserted and must swap with its parent, 67. Finally, the heap is in order.

Notice how the maximum number of swaps (2) happened in the worst case, when the number being inserted into the end of the heap was the minimum value of 4. This maximal number of swaps is equal to $k - 1$, where k is the height of the heap itself.

In layman's terms, "to percolate" means to filter gradually through something, as coffee does in an aptly named percolator. This process of "bubbling up" a lesser value within a heap is known as "percolating up" and is an integral part of the insert method in a heap data structure. As demonstrated later, the deletion of an element of a heap involves percolating down.

The min heap in this project is implemented with an ArrayList. The parent-child relationship is modeled mathematically, with the left child of a node with index x found at index $2x$, and the right child of a node found at index $2x + 1$. Thus, the children of the node at index 1 are found at indices 2 and 3. The children of the node at index 2 have indices 4 and 5, as seen in Figure 5.

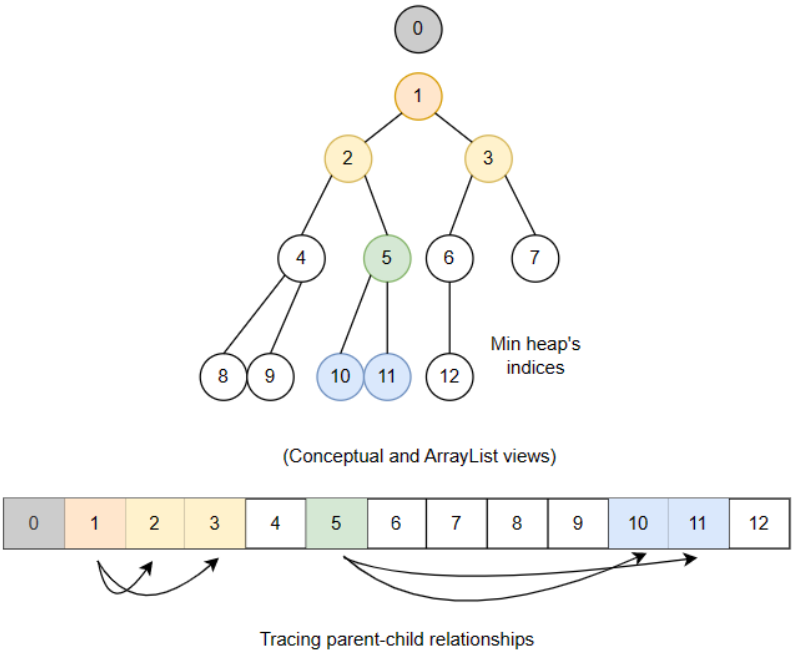


Figure 5: The indices of a heap represented visually and in an ArrayList.

Conversely, this means that the parent of a given node y can be found at the mathematical floor of $y/2$.[2] Notice that for ease of mathematical computations, the ArrayList index of 0 is ignored. Therefore, the value stored at index 0 is irrelevant, since it is never accessed.

Heaps and their variations have various applications. Their structure and methods are used in the implementations of heap sort, Dijkstra's shortest path algorithm, priority queues, and many other algorithms and data structures [3].

4 DeleteMin Algorithm Implementations

This section discusses two versions of the DeleteMin algorithm: one for implementation with a simple array, and another for a min heap implementation. A step-by-step walk-through of both versions is provided for this algorithm, which deletes the minimum value from a given data set.

4.1 DeleteMin for Arrays

The process of deleting the minimum value for an array first requires the minimum value to be found. Of course, if another algorithm is used to first sort the array, the minimum can easily be located by its index. However, if the DeleteMin algorithm is to be examined on its own, it must be assumed that there is no starting information about the orderliness of the contents of the array. Consider the example of an array with values 11, 7, 67, 4, 8, 12.

[2]By "mathematical floor", it is meant that any fractional results are rounded down to the nearest integer. Then 3.9, 3.5 and 3.0 would all be rounded to 3.

The arrayDeleteMin method first instantiates a new variable for the minimum value, and declares its value to be the value of the first element in the array. At this moment, the first element (11) is the min value. Another variable, *location*, is also instantiated, and its value is declared to be 0 (the index of the first element). Now, there are two active variables: the value of the minimum, and its location in the array.

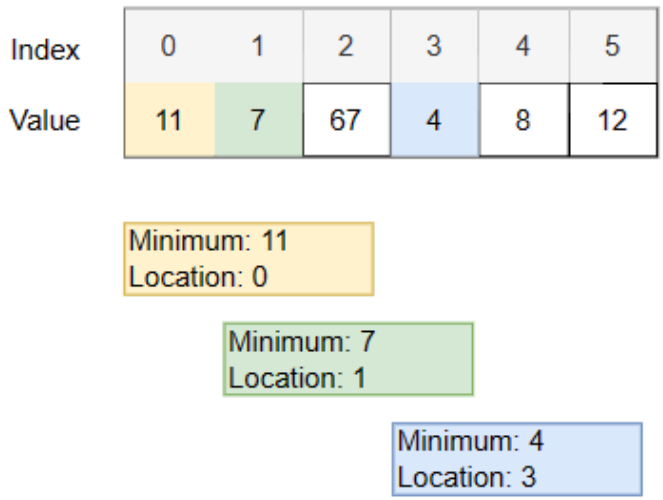


Figure 6: Finding the minimum value of an array.

As seen in Figure 6, the next step is to go through the array, element by element, and compare the value x at a given index a to the saved minimum value. If the value x at index a is smaller than the minimum, the min value is set to x and then min location is set to a. This continues until the end of the array is reached. At the end of this process, the location and value of the minimum are both known. In the example, the values of the minimum and the location are updated twice: first to a minimum of 7 at index 1, and then to value 4 at index 3. There is no value less than 4 after index 3, so that is the minimum.

Since an array is statically-sized, it is impossible to truly 'delete' an element. Therefore, a new array with a size smaller than the original array must be created, and the data must be transferred over to this new array with the minimum element excluded in the transfer. This is illustrated in Figure 7.

The transfer is done in two steps, as seen in Figure 7: first, the values from index 0 up to the index of the min's location are transferred (more specifically, copied) directly index to index: the element at index b from the original array is transferred to index b in the new array. In the example, the elements at indices 0, 1, and 2 from the original array are transferred to indices 0, 1, and 2 in the new array.

Second, the rest of the old array is transferred to the new array, beginning right after the location index to the end of the array. Now, the values at index b in the old array are transferred to index $b - 1$ in the new array. This -1 accounts for the skipped minimum element. The example shows that the values at indices 4 and 5 from the original array are copied over to indices 3 and 4 in the new array. Thus, this version of the algorithm deletes the minimum value from an array of values.

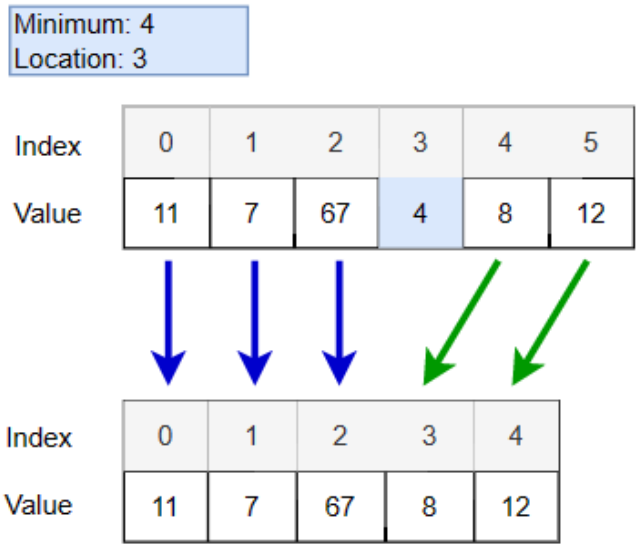


Figure 7: Transferring original array contents to new array.

4.2 DeleteMin for Heaps

Finding the minimum of a min heap is no difficult task: it is found at the root of the heap, at index 1. Due to the ArrayList implementation, it would be easy to simply use the ArrayList method remove() to get rid of the root. However, this would destroy the structure of the heap, and so the heapDeleteMin method is a bit more complicated.

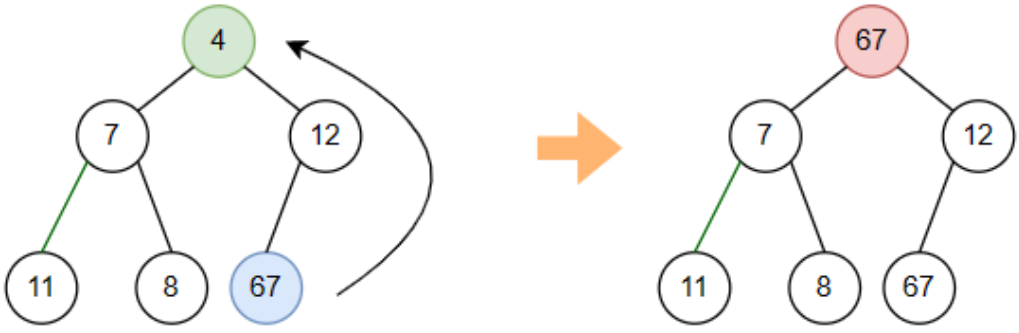


Figure 8: Replacing the root with the heap's last value.

In the heapDeleteMin method, the value at the last index of the heap is copied over to the root. In the example, seen if Figure 8, '67' replaces '4'. Thus, the minimum value of the heap is effectively deleted. However, in doing this, the heap order is disturbed. The way to remedy this is to call the percolateDown method. It works similarly to the percolateUp method described during insertion–however, it works in the opposite direction. This downward percolation is seen in Figure 9. Notice that the maximum number of swaps when percolating down is still $k-1$, with k being the height of the heap, just like with percolateUp.

In the percolateDown method, the out-of-place parent swaps places with the lesser of its children. This node is then compared to its two children, and if one of them has a lesser value, a swap occurs. In the example, the root node with value 67 is swapped with 7, the lesser of its children. Now 7 is the root of the heap (and the new minimum), but 67 is still out of place. Next, 67 is swapped with 8, which is the lesser of its two children. With this swap, the heap is in order, but there is still a problem.

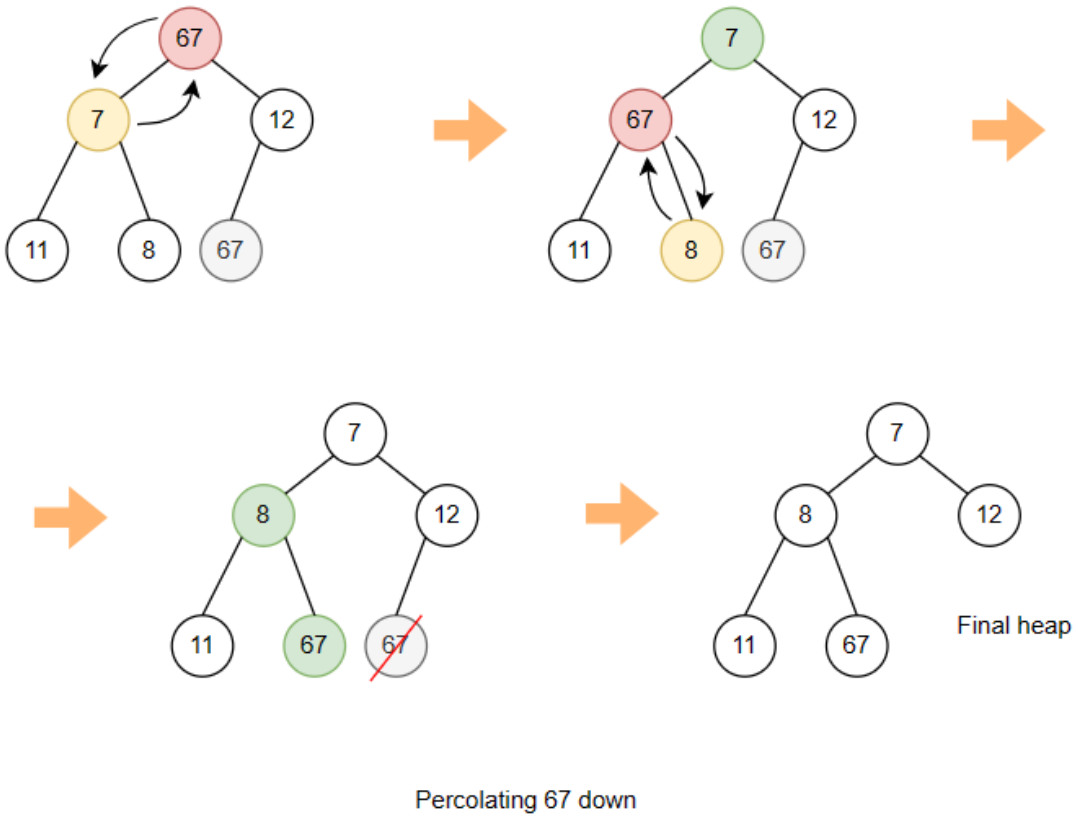


Figure 9: Percolating down 67.

As seen in Figure 9, After the last swap is completed, the last element of the heap is truly deleted using ArrayList's remove method. In the example, the node that contained 67 in the original heap is no longer needed, since it was copied and percolated down. So the size of the heap is reduced by one as the minimum is deleted and the order of the heap is preserved.

5 DeleteMin Algorithms Time Complexity

The two versions of the DeleteMin algorithm have differing time complexities, which is understandable given their different executions. In the best, worst, and average cases, the heapDeleteMin method is more efficient in completing its task than the arrayDeleteMin method.

5.1 heapDeleteMin Complexities

5.1.1 Best Case - Constant

The complexity function for this algorithm of a heap of size n is $f(n) = 9$, meaning that the complexity is of constant time.

heapDeleteMin has five statements that are always executed, no matter what. The first step is retrieving the ArrayList that stores the heap data, and next is saving the last value of the heap. Third is setting the value of the first node to this last value. Then percolateDown is called, and finally the last value of the heap is deleted. The rest of the method contains an if-statement that is not called in the best-case scenario.

The percolateDown method increases the complexity of our function $f(n) += 4$, as follows. First, the index of the second-to-last heap element is saved.[3] Second, an index marker called parent is set to the first element. Then index markers for the left and right children of this parent are assigned values mathematically based on the value of the parent. The rest of the percolateDown method is contained in a while loop that does not execute in the best case scenario.

Thus, the complexity function for a heap of size n is $f(n) = 9$ and $\Omega(1)$.

5.1.2 Worst Case - Logarithmic

The function for the worst-case heapDeleteMin complexity is $f(n) = 14 + 11(\log_2(n + 1)$, meaning that the worst-case time complexity is logarithmic.

As with the best case, there are five statements in heapDeleteMin that are always executed, no matter what. There are two points worth noting: the call to the percolateDown method, and an if-statement that is executed when there are only three elements in the heap and the parent is greater than one of its children, which calls the swap method (4 additional statements, in all cases).

PercolateDown has 4 statements that are always executed. The rest of the method is contained in an if-statement that executes a while loop. This block of code is executed in the event that the heap is out of order, with the root node being something other than the minimum value. The number of times that this while loop is executed depends on the number of swaps between parent and child that occur in the percolating process, as described in Section II.

Consider that the maximum number of swaps made is equal to the height of the heap. Since the maximum number of nodes n for a tree of height k is $n = 2k - 1$, the height k of a tree with n nodes will be $k = \log_2(n + 1)$. Since a heap is not always a complete heap, the expression for the value of k may not always be a whole number. In such cases, the mathematical roof[4] of the expression is taken, since a partial height is not possible.

So, in the worst case, the number of swaps for an element inserted into a tree to make n elements will be equal to $\log_2(n + 1)$. Code-wise, this results in one call of the swap method (4 statements) and one variable reassignment (1 statement) in the while loop, assuming the conditions for the inner if-statement are met. Outside of the if statement block, there are two more variable reassignments within the while loop that execute for every iteration of the while loop. There is also one break statement.

[3] Later, when the last node that has been percolated is deleted, the element at this index will become the new last node.

[4] By "mathematical roof", it is meant that any fractional results are rounded up to the nearest integer. Thus 3.1, 3.5 and 3.9 would all be rounded up to 4

So, the equation for the overall complexity of the heapDeleteMin method is $f(n) = 5 + 4 + 4 + (4 + 1 + 2)(\log_2(n + 1)) + 1 = 14 + 11(\log_2(n + 1))$, for an overall logarithmic complexity.

Thus, this algorithm has $O(\log_2(n))$.

5.1.3 Average Case - Logarithmic

Although it is difficult to compute an exact equation for the average case, it is most likely to also be of logarithmic complexity.

The precise best and worst cases seem to be about equally likely–the best case scenario will happen if the input is sorted in ascending order, and the worst case scenario will happen if the input is sorted in descending order. As for the rest of the cases, some percolation will happen, but not always up the full height of the heap. However, due to the percolation that does happen, the average complexity will be closer to $\log_2(n)$.

Thus, the average complexity for heapDeleteMin is $\Theta(\log_2(n))$[4].

5.2 arrayDeleteMin Complexities

5.2.1 Best Case - Linear

The function for the best case time complexity of the arrayDeleteMin method is $f(n) = 3n-1$.

As discussed in Section 4.1, first it is necessary to create variables to store the minimum value and its location (2 statements). Then it is necessary to go through each element of the array once. Each element is compared to the minimum. In the best case, the first element is the minimum, and no variable re-assignment takes place. Then the complexity only increases by n, the size of the array, for the assignment of the i variable which keeps track of the for loop iterations.

A new array of smaller size must be created, which adds one statement to the bill. Then the contents of the original array are transferred to the new array. Although this is broken up into two for-loops in the code, essentially the iteration variable i is assigned $n - 1$ times and the value of each index in the new array is set $n - 1$ times, for a total of $2(n - 1)$ statements. Finally, the array is returned (1 statement).

Altogether, there are $2 + n + 1 + 2(n - 1) + 1 = 3 + n + 2n - 2 + 1 = 3n + 2$ steps, making the complexity of this method linear with $\Omega(n)$.

5.2.2 Worst Case - Linear

The function for the worst case of arrayDeleteMin is very similar to the best case, with the function being $f(n) = 5n - 2$.

The difference happens in the if statement that checks whether the min value and its location need to be updated. In the worst case scenario, where the array is sorted in descending order, the minimum variable and the location variable are re-assigned in each iteration of the for-loop. This adds 2 statements to each iteration of the loop, for an additional total of $2n$ statements in the overall equation.

Thus, $f(n) = 3n - 2 + 2n = 5n - 2$ for a linear complexity of $O(n)$.

5.2.3 Average Case - Linear

The average time complexity for the arrayDeleteMin method will be linear with a function somewhere between $3n - 2$ and $5n - 2$.

It is difficult to name the exact complexity equation for the average case (although $f(n) = 4n - 2$ is a tempting solution, further analysis must be done to determine whether the typical cases average out to this).

Regardless, since both the best- and worst-case complexities are linear, the average case must also be linear for a $\Theta(n)$ complexity.

6 Comparing the Algorithms

Although the array structure and therefore the arrayDeleteMin method are easier to understand conceptually, they are not exactly the most efficient. The minHeap structure is better than a regular array when dealing with minimums than an array in terms of both time and space complexity.

Consider the time complexities as derived in Section 5: the worst-case time complexity for the heapDeleteMin algorithm is better than the best-case time complexity for arrayDeleteMin. When deleting the minimum repeatedly for a given data set, this time difference will build up, especially for large data sets.

MinHeaps also win out in terms of space. Although a single ArrayList takes up more memory than a single array [5], the fact that the arrayDeleteMin method requires the creation of a new array means that it begins to lose out in terms of space complexity. Again, if the operation of removing the minimum is executed many times repeatedly, it becomes obvious that the minHeap data structure and its heapDeleteMin method will be better, since they will only edit the values of the one original ArrayList–while the arrayDeleteMin method will be creating a new array each time.

7 Conclusion

Although the array data structure is easy to understand, it is not well suited for complex operations that require anything more than accessing or updating an element at a given (known) index. Many data structures have been written for different purposes, and the min heap is one of the most useful when the minimum value of a given set of data needs to be accessed, updated, or even deleted. The very similarly organized max heap is likewise very helpful when doing the same for the maximum value of a data set.

References

[1] N. Dale, D. Joyce, and Chip Weems. *Object-Oriented Data Structures Using Java.* Jones Bartlett Learning, 2018.

[2] Mark Allen Weiss. *Data Structures and Algorithm Analysis in Java.* Addison-Wesley, 2012.

[3] GeeksForGeeks. Applications of heap data structure, 2011. Last accessed 3 May 2023.

[4] OpenGenius IQ. Time and space complexiy of heap data structure operations, 2021. Last accessed 4 May 2023.

[5] Baeldung. The capacity of an arraylist vs the size of an array in java, 2020. Last accessed 3 May 2023.