## Problem Statement

Let’s say you want to take a large set of data and remove the smallest element each time you take a value out of the data. While there are many ways to achieve this, a priority queue is a very optimized and efficient option to choose.

While a priority queue interface aligns with this issue, there are multiple different data structures that can be used to implement a priority queue. To determine which data structure implementation is best, taking into factor the time and/or memory needed to run the program will help make this decision. To demonstrate this, I will compare the time complexity of a rootish array stack and a binary heap to determine which implementation is better for this scenario. My test for the problem statement involves adding all of the elements of a dataset into the program, then removing the smallest value repeatedly until the single largest value is left.

## Analytical Performance Analysis

- **Rootish Array Stack**

My implementation of the rootish array stack is more of a list data structure than a stack. I store the data with an array of pointers that point to increasing array sizes, calling each array a “block”. I add data to the end of the array, and remove the smallest value by searching through the entire array to find the smallest element.

The following chart shows the runtime of all implemented functions within this program:

(Note: add() and removeSmallest() are bolded since these are the two functions looped through in the main program, and are the determining factors of both programs’ overall complexity.)

| Function | (Worst Case) Run time in number of operations | Time in Big-oh Notation | Notes |
|---|---|---|---|
| Indextoblock() | 8 | O(1) | This is a constant time since size n does not affect this function. |
| set() | 17 | O(1) | This is a constant time since size n does not affect this function. Indextoblock() is used within this function. |
| grow() | 9+5$\sqrt{n}$ | O($\sqrt{n}$) | This function loops based on the number of <u>blocks</u>, not elements. I made this conclusion by determining the relationship number of blocks $b \approx \sqrt{n}$. |
| erase() | 1+32n | O(n) | erase() mainly consists of a loop through all of the elements. |
| **add()** | 32+5$\sqrt{n}$ | O($\sqrt{n}$) | With the worst case scenario, grow() is assumed to be used every iteration. |
| **removeSmallest()** | 5+53n | O(n) | erase() is used within this function, causing this time complexity |
| <u>**Overall**</u> | n(37+5$\sqrt{n}$+53n) | O($n^2$) | The looping of removeSmallest() *n* times causes this complexity. We can ignore 37+5$\sqrt{n}$ since 53n eventually grows large enough for rest to be negligible. |

- **Binary Heap**

My implementation of the binary heap is a min-tree data structure. Each time a value is added, the tree shifts to always have the smallest value at the top. I remove the value on top, then reshift the tree each time the smallest element is removed.

The following chart shows the runtime of all implemented functions within this program:

| Function | (Worst Case) Run time in number of operations | Time in Big-oh Notation | Notes |
|---|---|---|---|
| siftDown() | 4+20log(20) | O(log(n)) | The loop within the function doesn't iterate through *n* elements as if it goes down a part of the tree, and calculates to approximately log(n) iterations instead. |
| reserve() | 4+5n | O(n) | The function loops through every element to fill in the new array. |
| **add()** | 15+5n+14log(n) | O(n) | With the worst case scenario, reserve() is assumed to be used every iteration. |
| **removeSmallest()** | 9+20log(n) | O(log(n)) | While removing the value itself isn't costly, the function uses siftdown() to readjust the heap |
| <u>**Overall**</u> | n(24+5n+34log(n)) | O($n^2$) | The looping of removeSmallest() *n* times causes this complexity. We can ignore 37+5$\sqrt{n}$ since 53n eventually grows large enough for rest to be negligible. |

**Before comparing the overall time of both programs**, I used amortized analysis for both add functions since grow() and reserve() would not be used in every instance.

For my amortized analysis, I used the aggregate method to determine an average time of the function. To demonstrate this, I will show the process I took when looking at the binary heap:
Since the aggregate method examines the total running time for a sequence of operations, I made a table to keep track of multiple variables:

- Index: This keeps track of how many times the add() function is used, starting with an empty array of size 1.
- Size: This will tell the size of the array each iteration. This is NOT how many elements are in the array, rather how many elements the array can hold currently.
- Cost: To determine the cost over time, we will make the following assumptions:
    - Cost = 1 if the add() function is used without growing the array, since the function does not need to loop through the elements to add to the array.
    - Cost = current index when the reserve() function is used, since this is when the function loops through the elements to copy the array. This cost also includes adding another element to the array as normal.

| Index | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|
| Size | 1 | 2 | 4 | 4 | 8 | 8 | 8 | 8 | 16 | 16 |
| Cost | 1 | 2 | 3 | 1 | 5 | 1 | 1 | 1 | 9 | 1 |

With this information, I want to make an expression for the total cost of the function, $C_n$.
For every element, there is always a cost of 1 guaranteed. Since this happens *n* times, we can add n to our expression.
Looking at the table, the cost includes the reserve() function is index-1 when the previous index is a power of 2. Not only do we need a summation

of this reserve function cost, but we also need to figure out how often the added cost occurs.
For the value, we know every index is a power of 2, so we can assume every added value is $2^k$. This value is added when n = 2,4,8,16… which is the same pattern as $log_2(n)$, knowing this, we can create this expression:

$$C_n = n + \sum_{i=0}^{log_2(n)} 2^i$$

The summation can be simplified to $2^{log_2(n)} - 1$

$$C_n = n + 2^{log_2(n)} - 1$$

To make this bounded:

$$C_n \leq n + 2^{1+log_2(n)} - 1 = n + 2n - 1 = 3n - 1$$

Thus, the amortized cost satisfies (AC meaning amortized cost):

$$AC(n) \leq \frac{3n-1}{n} = 3 - \frac{1}{n} < 3$$

So, $AC(n) = O(1)$.

This reduced the Big-oh notation of both programs into:

- Rootish Array Stack: O(5n$\sqrt{n}$)
- Binary Heap: O(34log(n))

Since log(n) is less complex than $n\sqrt{n}$, the binary heap has smaller growth rate than the rootish array stack

Amortized analysis was not used for the removeSmallest() functions since no function to shrink the list/tree was implemented.

## Empirical Performance Analysis

- **My Benchmark**

To compare my programs Empirically, I created a benchmark with these points of assessment:

- I used a data set of random inputs, inputs in increasing order, and inputs in decreasing order
- Each of these data sets were tested with a  data amount of 10, 100, 1000, 10000, 50000, and 1000000

I ran each program 5 times with the dataset chosen and used the bash *time* command to see the real time of each run. I then averaged the time and slotted this result onto a graph. I also used Callgrind on a single run of each dataset to create a table of instruction reads the program produced.

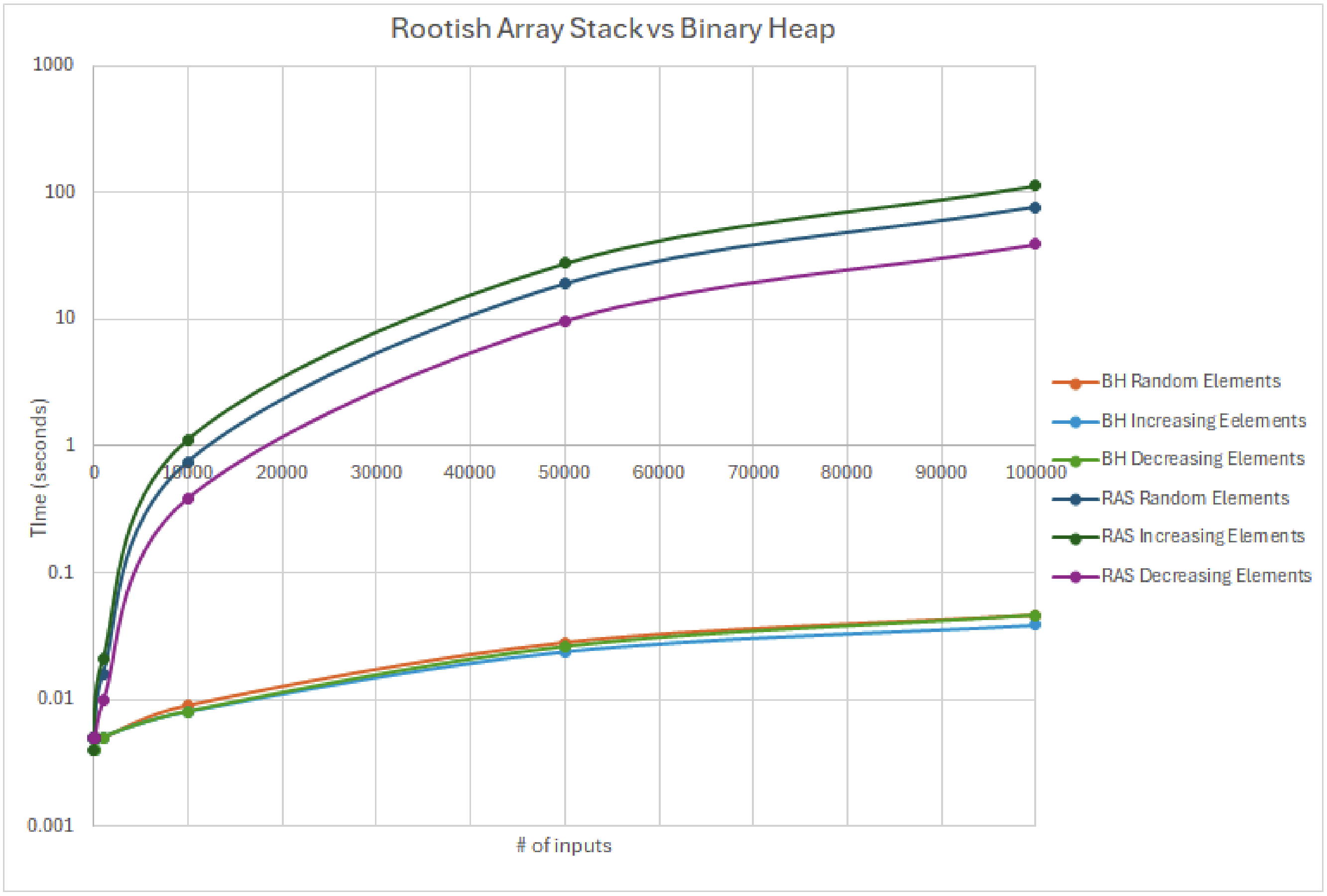
- **Results**

| (ir of entire program) | 10 | 100 | 1000 | 10000 | 50000 | 100000 |
|---|---|---|---|---|---|---|
| Binary Heap (random elements) | 35,268 | 195,193 | 2,038,050 | 22,857,927 | 140,369,715 | 290,820,244 |
| Binary Heap (increasing elements) | 30,804 | 159,733 | 1,809,305 | 22,167,586 | 127,941,366 | 265,799,162 |
| BInary Heap (decreasing elements) | 31,833 | 189,276 | 2,326,192 | 29,717,292 | 173,871,677 | 364,718,997 |
| Rootish Array (random elements) | 44,347 | 1,031,742 | 86,466,355 | 8,561,602,514 | 214,465,081,040 | 856,970,500,666 |
| Rootish Array (increasing elements) | 44,994 | 1,403,817 | 128,248,875 | 12,713,693,187 | 317,575,181,343 | 1,270,152,067,848 |
| Rootish Array (decreasing elements) | 37,659 | 597,047 | 46,830,375 | 4,564,508,187 | 113,829,256,343 | 455,160,217,605 |

Looking at the amount of instruction reads each program run had, Binary Heap has much fewer reads than the rootish array stacks. Interestingly, Using a dataset consisting of increasing elements was the worst case for the rootish array stack, while decreasing elements was the worst case for the binary heap.

For the binary heap, I believe this is due to how the implementation adds new elements into the tree. Every new element added is placed in an available space farther down the tree, then shifted up to the correct position. Since the root of the tree is supposed to be the smallest element, and every new element added is the smallest with a decreasing dataset, that means each new element is guaranteed to shift up to the top of the tree. This adds a lot more operations in the process compared to an increasing or random data set.

On the other hand, the rootish array stack is a little more tricky. At first, I thought a decreasing dataset would make the removeSmallest() function check and update the smallest value with each value in the list, creating the worst case scenario. In reality, the increasing data set is the worst case since the smallest element is always at the front of the list. This makes the program move all of the values in the list up one index in every single removeSmallest() iteration, which uses n-1 operations at a minimum consistently.



This graph uses the runtime of each program when I used the time bash command. The graph above shows a dramatic difference between binary heap and rootish array stack, regardless of the kinds of inputs used. While the difference is less obvious for the binary heap, the rootish array stack seemed to take much longer with a dataset of increasing elements, and performed best with the dataset of decreasing elements. Both datasets are less complex than my analytical predictions, with the binary heap arguably closer in accuracy with its slightly logarithmic shape.

Overall, my analytical and empirical analysis shows that the binary heap has a lesser time complexity than the rootish array stack.

## What did I learn?

I was able to explore further into analytical performance analysis and big-oh notation outside of the data structures course curriculum. I also learned about amortized analysis, and how it can be used to create a more accurate prediction of a program's complexity growth rate

I also learned the process of empirical performance analysis with hands-on experience, creating my own benchmark to compare multiple programs. Through this I was surprised to learn that the order of the data imputed can make a noticeable difference based on the program.

Lastly, I was able to learn about other data structures outside of the course and implement one to test with (rootish array stack).

## Resources

https://opendatastructures.org/ods-cpp.pdf - Interfaces, what is a priority queue, and root array stack implementation

https://www.w3schools.com/cpp/cpp_polymorphism.asp - Polymorphism (used to make testing the programs quicker and more efficient)

https://www.geeksforgeeks.org/creating-array-of-pointers-in-cpp/ - how arrays of pointers are implemented. This is used for helping construct the root array stack implementation.

http://www.cs.cornell.edu/courses/cs3110/2011sp/Lectures/lec20-amortized/amortized.htm - Goes into detail of amortized analysis, and how to use the aggregate method, financial method, and potential method

https://www.math.umd.edu/~immortal/CMSC420/notes/amortized.pdf - explains how to create the expression for the total and average cost of a program using the aggregate method

https://persis-randolph.medium.com/big-o-notation-for-binary-search-trees-8f0f50b016ef - Talks about big-o-notation for binary search trees, but helped me determine big-o-notation when traveling up or down only a portion of a tree.