2XB3 Lab 5

Casey Doede – 001223493 – doedecd@mcmaster.ca – L01

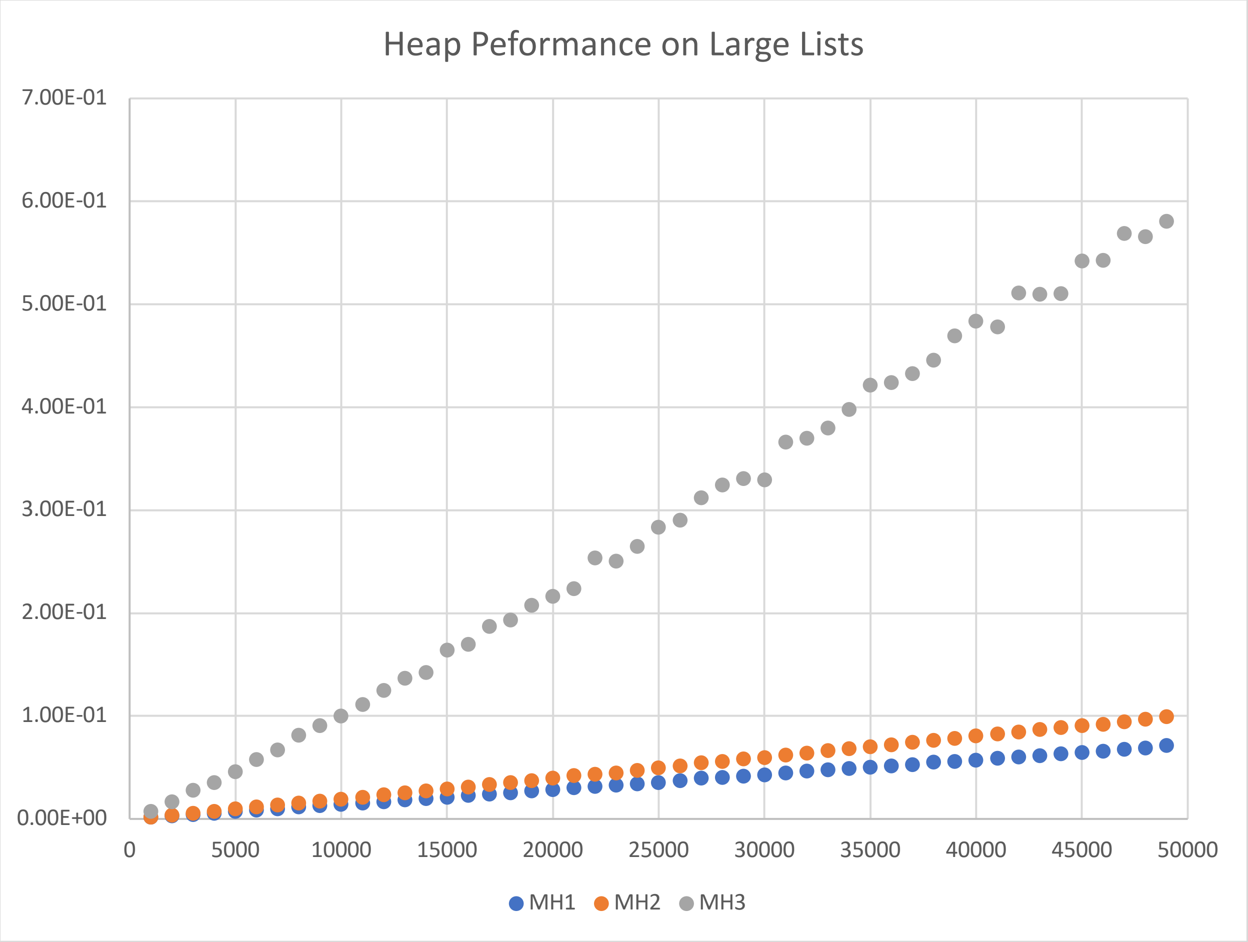
Amaan Ahmad Khan – 400230523 – khana251@mcmaster.ca – L01

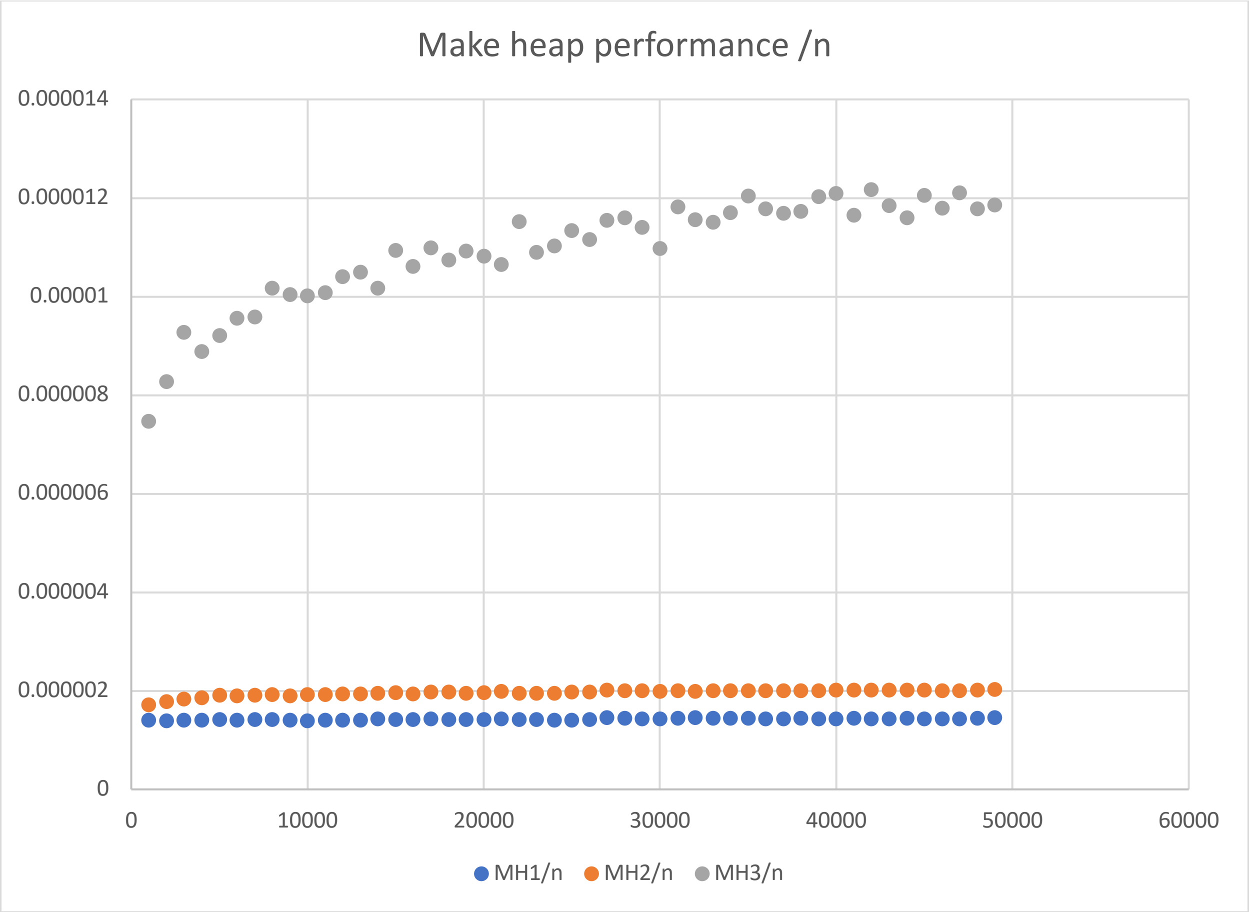
Alex Axenti – 400268807 – axentia@mcmaster.ca - L04

Heap Implementation

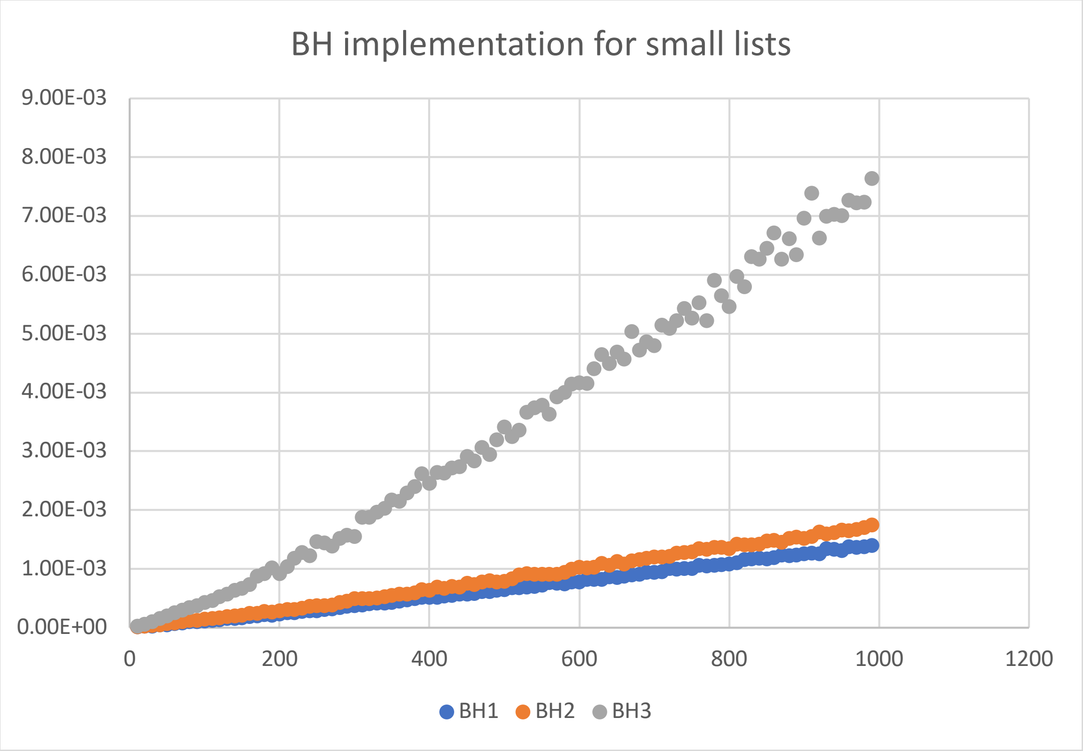
Both the initial implementation of heapsort and the second implementation of heapsort utilize the sink and insert methods, which are O(log n) worst case. It would then be fair to assume that if one iterates across a list n/2 times and n times for MH1 and MH2 respectively that they would be O(nlogn). In practice this is not the case and the first build heap function is actually O(n). This is due to the fact that sink operations time complexity changes based on the position in the heap. The second implementation, build_heap_2 iterates across each item in the list, thus n times, and each time swim is called, which has performance of logn. Thus one can analyze that the runtime is O(nlogn).

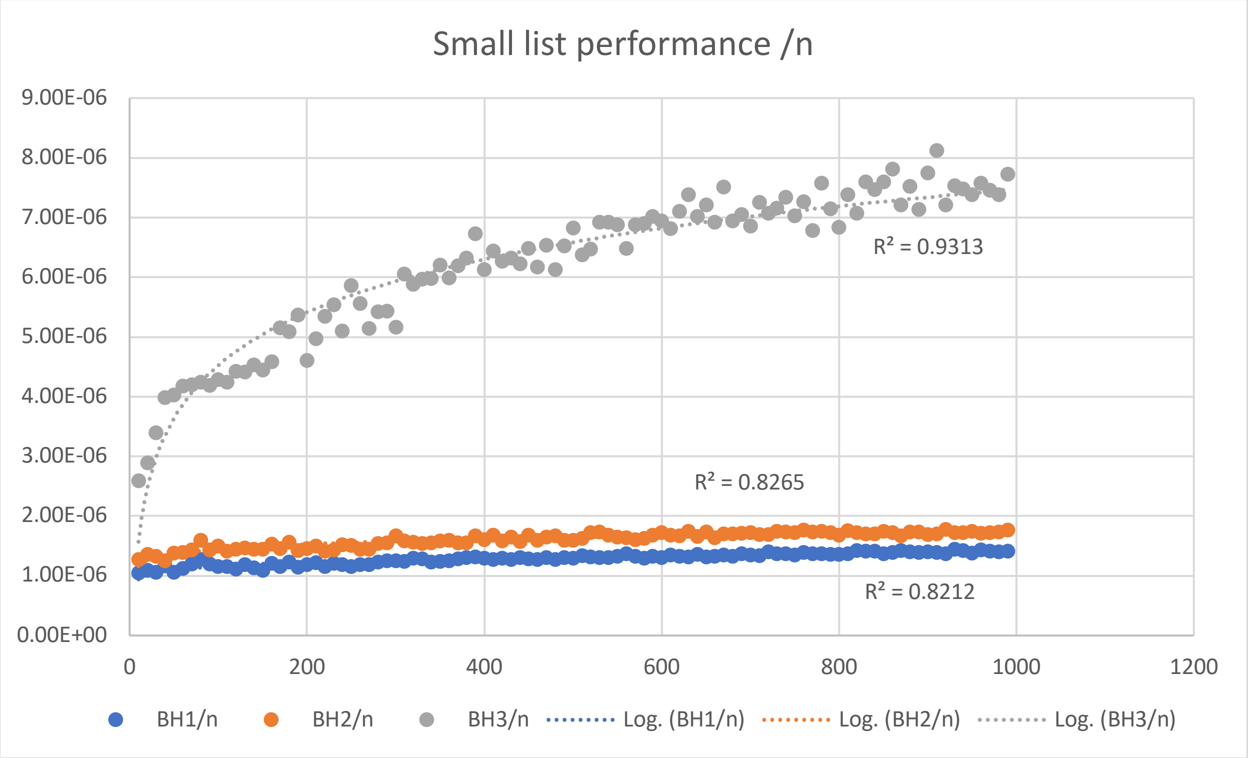
For MH3 again the sink function is used which is O(logn) in the worst case. This is then called n times, and then repeated an unknown number of times until the is_heap returns true. One can assume in the worst case it would be O(n^2log n)





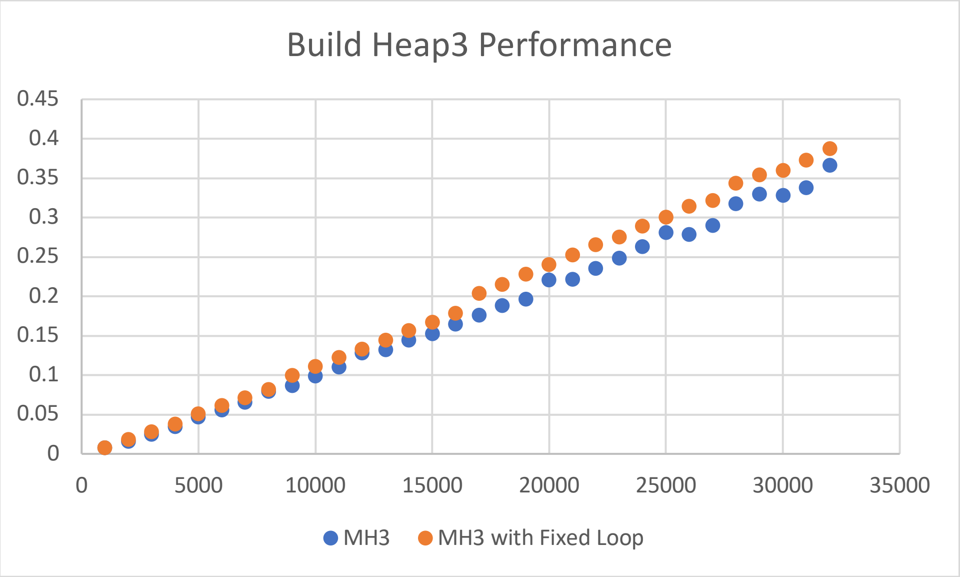
As expected, one can observe a large performance disparity between the first two implementations of heaps and the third implementation of heap. The first implementation appears to be O(n) complexity as stated. The second implementation appears to be something more akin to O(nlogn) as one can notice the datapoints seem to follow a logarithmic trend. Finally, the third implementation it is very obvious that the solution is in the realm of O(nlogn)


BH implementation for small lists
9.00E-03
8.00E-03
7.00E-03
6.00E-03
5.00E-03
4.00E-03
3.00E-03
2.00E-03
1.00E-03
0.00E+00
0
200
400
600
800
1000
1200
BH1
BH2
BH3

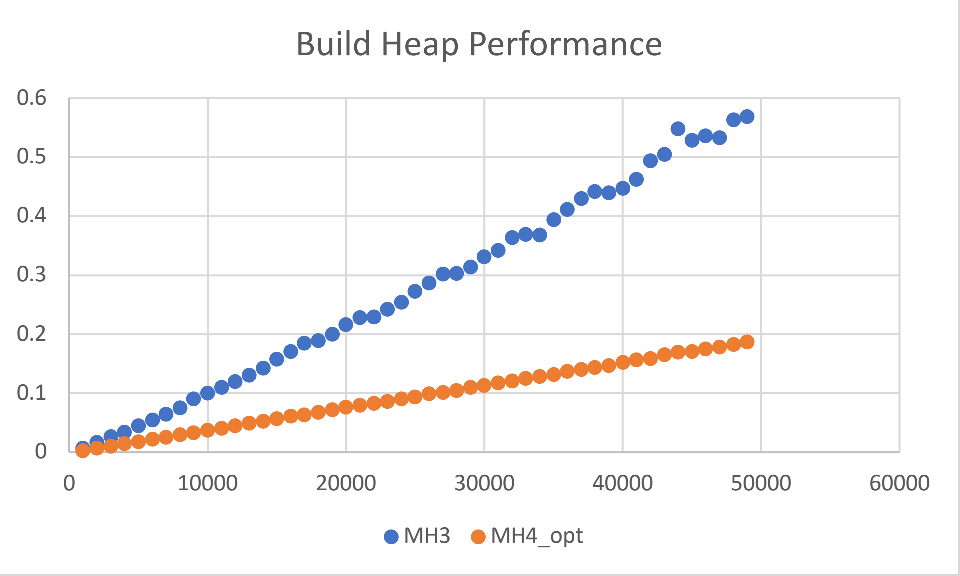

Small list performance /n
9.00E-06
8.00E-06
7.00E-06
6.00E-06
5.00E-06
4.00E-06
3.00E-06
2.00E-06
1.00E-06
0.00E+00
0
200
400
600
800
1000
1200
R² = 0.9313
R² = 0.8265
R² = 0.8212
BH1/n
BH2/n
BH3/n
Log. (BH1/n)
Log. (BH2/n)
Log. (BH3/n)

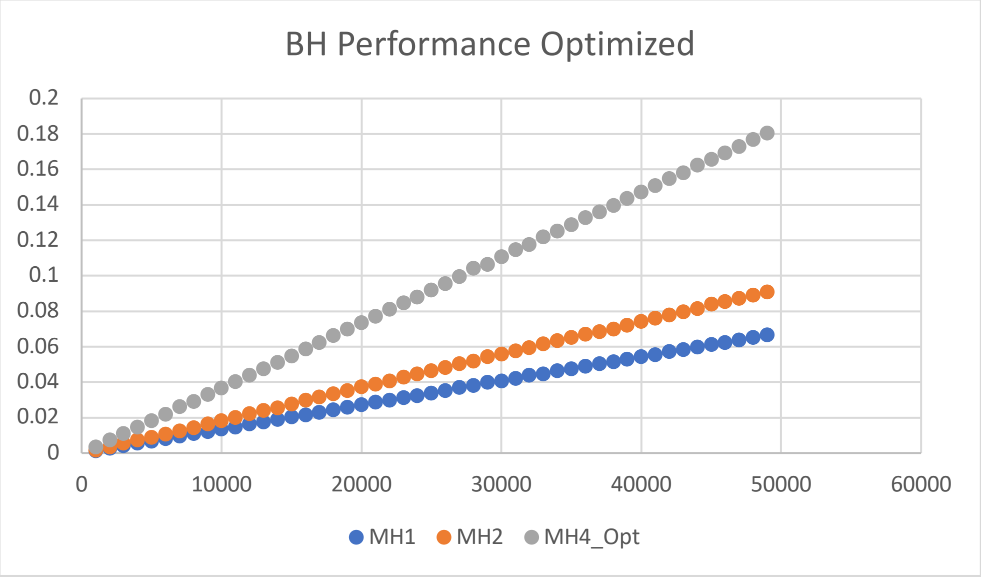
After some empirical testing it appears each node must sink at most logn times, thus one can omit the is_heap check and just run sink logn times to improve performance. As is_heap involves multiple comparisons and was ran after each sink loop, we assumed it would improve performance.

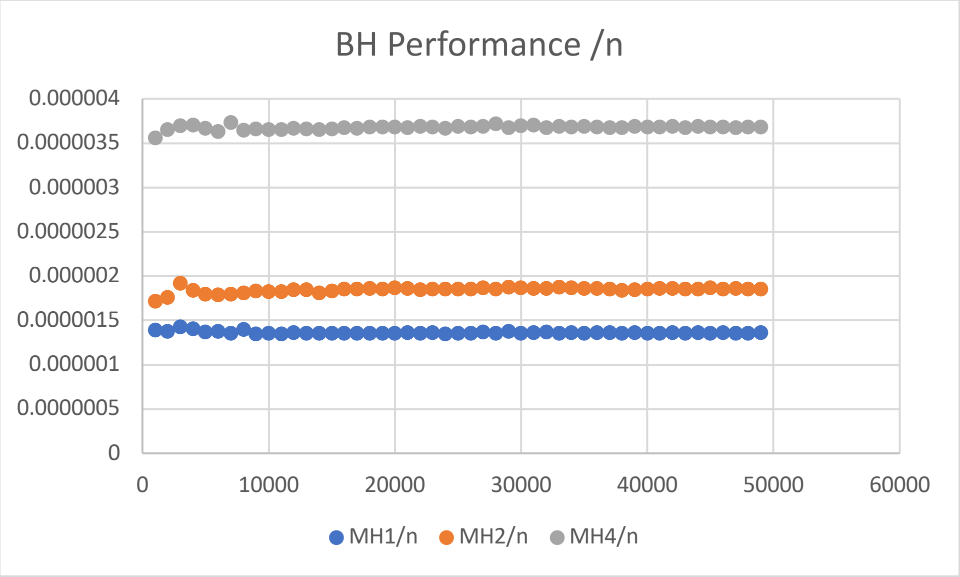
Instead, we found that looping for a fixed number of iterations, that is logn times, one would end up with reduced performance in comparison to checking if the list was structured as a heap. After testing further it was found with the is_heap check, the loop would sometimes break early as the heap structure was achieved in less than logn sink loops.

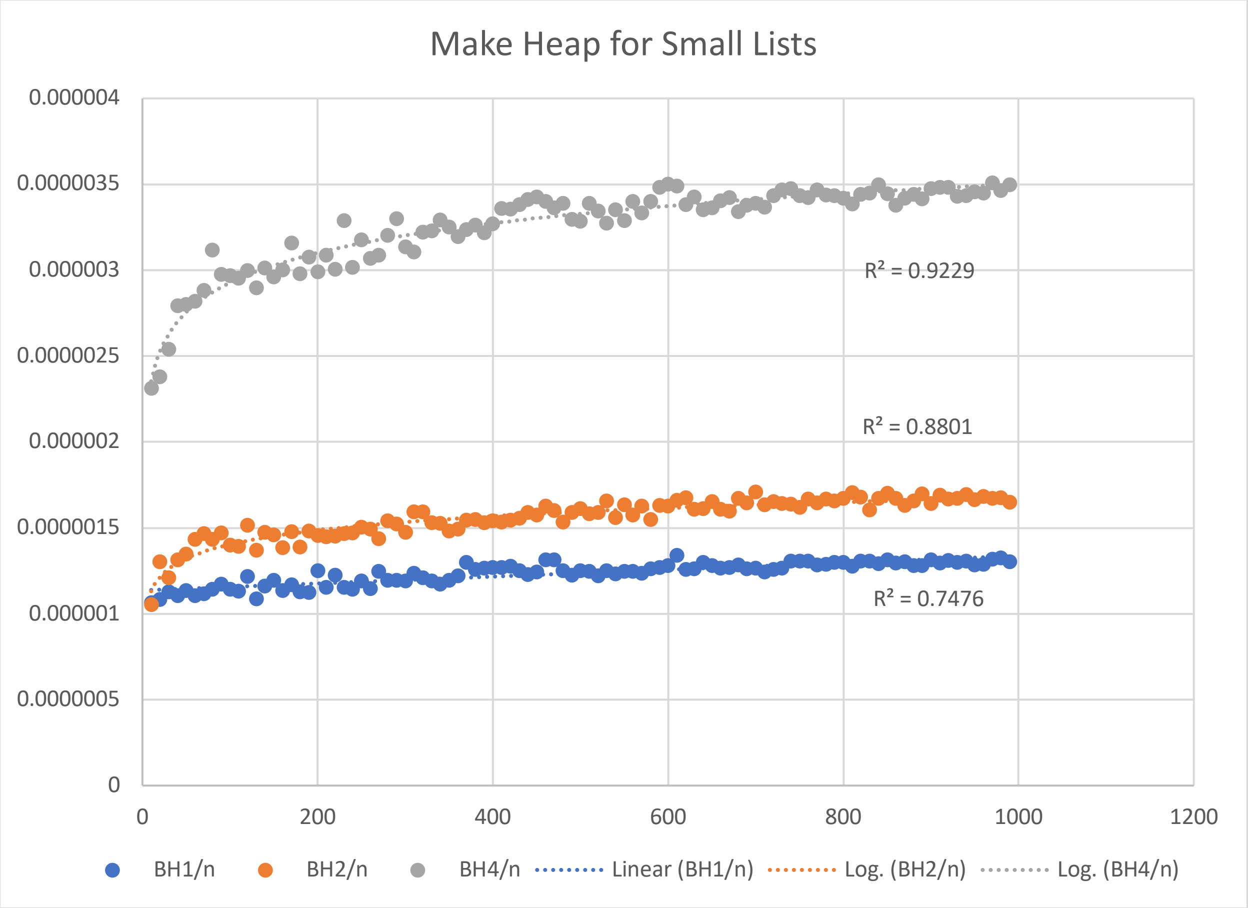


Next, we instead chose to consider the fact that after sink is ran on each node the bottom leaves will be in their proper location, thus sink does not need to be called on these nodes again. This means that each loop of sink is reduced in size by n/2, greatly increasing performance. This can be observed below.




BH Performance Optimized
0.2
0.18
0.16
0.14
0.12
0.1
0.08
0.06
0.04
0.02
0
0
10000
20000
30000
40000
50000
60000
● MH1
● MH2
● MH4_Opt


BH Performance /n
0.000004
0.0000035
0.000003
0.0000025
0.000002
0.0000015
0.000001
0.0000005
0
0
10000
20000
30000
40000
50000
60000
● MH1/n
● MH2/n
● MH4/n



One can now observe that both the second and fourth build heap operations appear to have a logarithmic nature to their operation, whereas it appears the original build heap implementation appears to remain roughly constant.

K-Heap

By adding more children to each node, the height of a tree is reduced. This can be beneficial, as there are less nodes to go through and compare until you've reached the leaves. The downside is that you need more compares on each node to make sure that all k children are less than or equal to the node. Therefore, when building the heap, there would be less sink operations needed, but each sink operation would require more compares.

In addition, if we were to implement the extract_max() operation which is needed for heap sort, instead of a worst case of $2\log_2 n$ compares, the worst case would be $k\log_k n$ compares. Therefore, depending on what k is, there could be more or less compares than a normal heap. For example, for a list size of 1000, $3\log_3 1000 < 2\log_2 1000 < 5\log_5 1000$.

The asymptotic complexity of sink for a k-way heap would be $O(k\log_k n)$ as traditional sink is O(logn) and the modified sink would involve lower heap heights, but at each height a greater number of elements must be compared, in this case k. For instance for a standard heap, when sink is called one compares the parent node value to that of its two children, and then reclusively calls itself logn times. In a k-way heap one must compare the parent node to its three children, and then it is recusively called $\log_k n$ times.