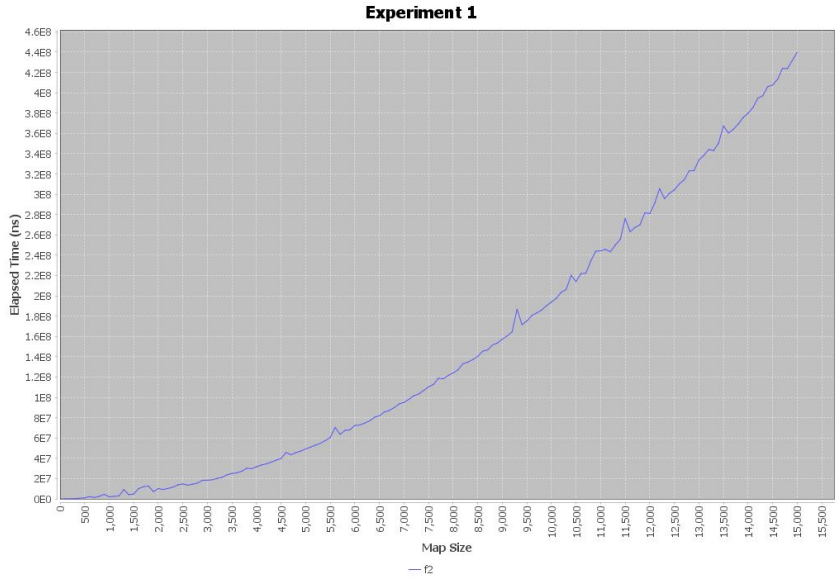
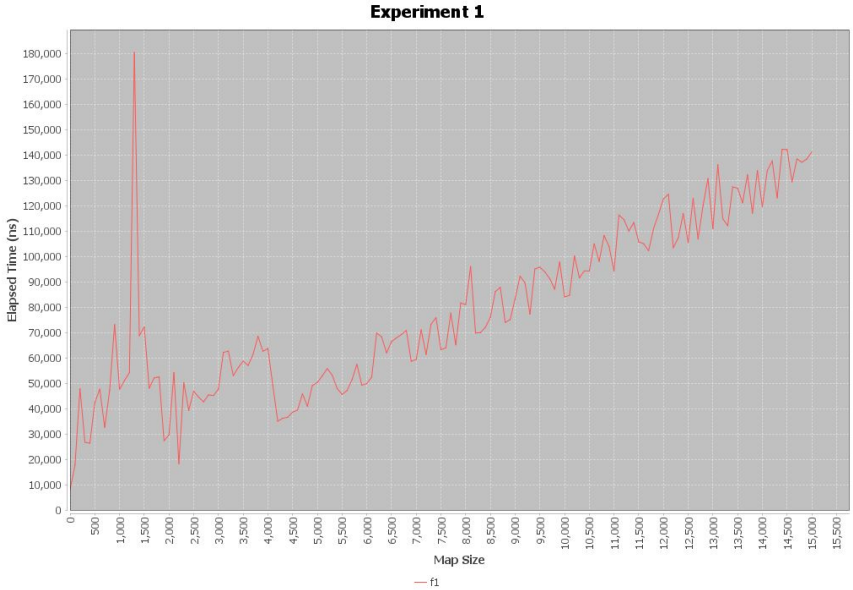
Project 2 Experiments

Experiment 1:

- **Prediction:** f1's runtime order is n and f2's runtime order is n^2, n for the size of input, because f1 is the best case which it only needs constant operations to remove the first entries, while f2 is the worst case which it has to traverse to the last entries for each removal.
- **Discussion:** We are not surprised since the plot is consistent with our prediction.
  f1 removes the first entry and replaces it with the last pair, which allows the next removal to remove the first entry again. Thus, f1 always only needs a constant operation to remove the pair at index0. F2 will always traverse to the last entry to find the index of the desired key for each call of remove(), its operation will depend on n. Therefore, since both f1 and f2 removes n times, f1 would have a runtime order of n * 1 = n while f2 would have runtime order of n*n = n^2, which results in the significant difference between the plot as n increases.



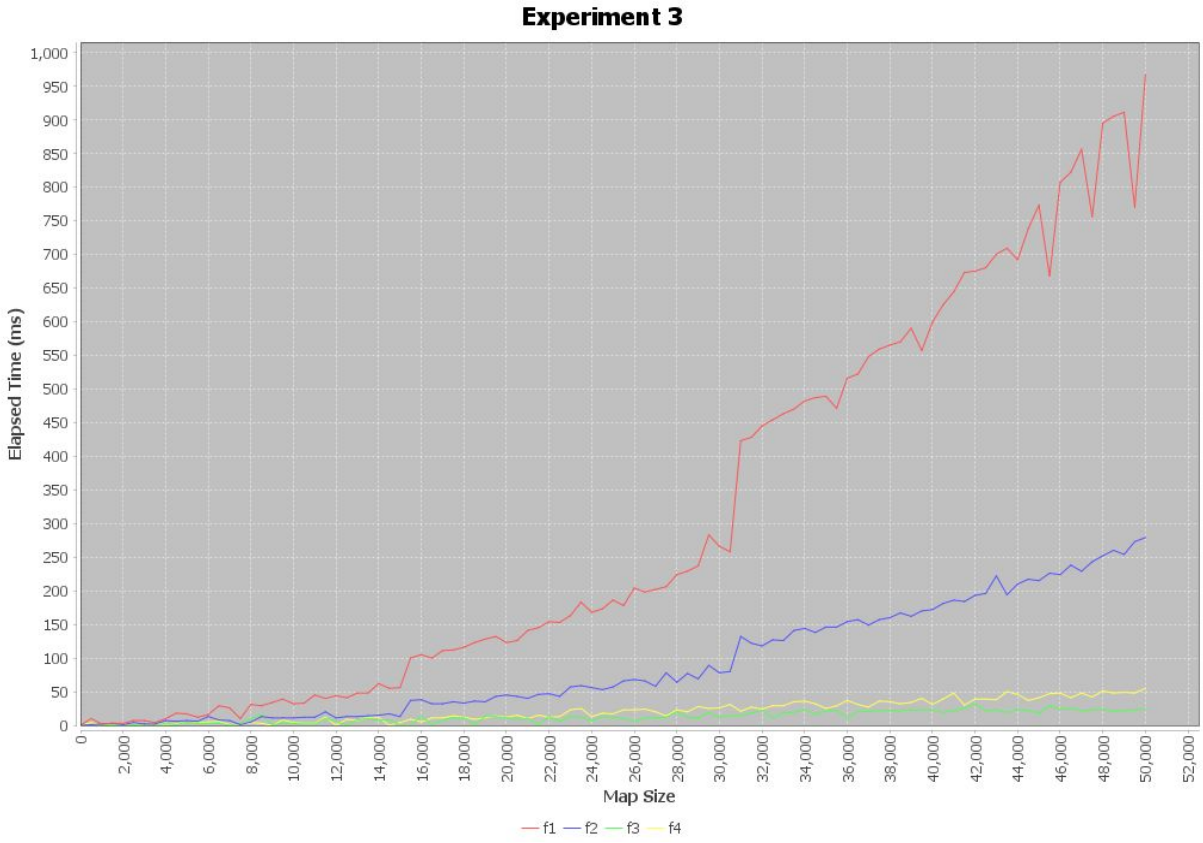


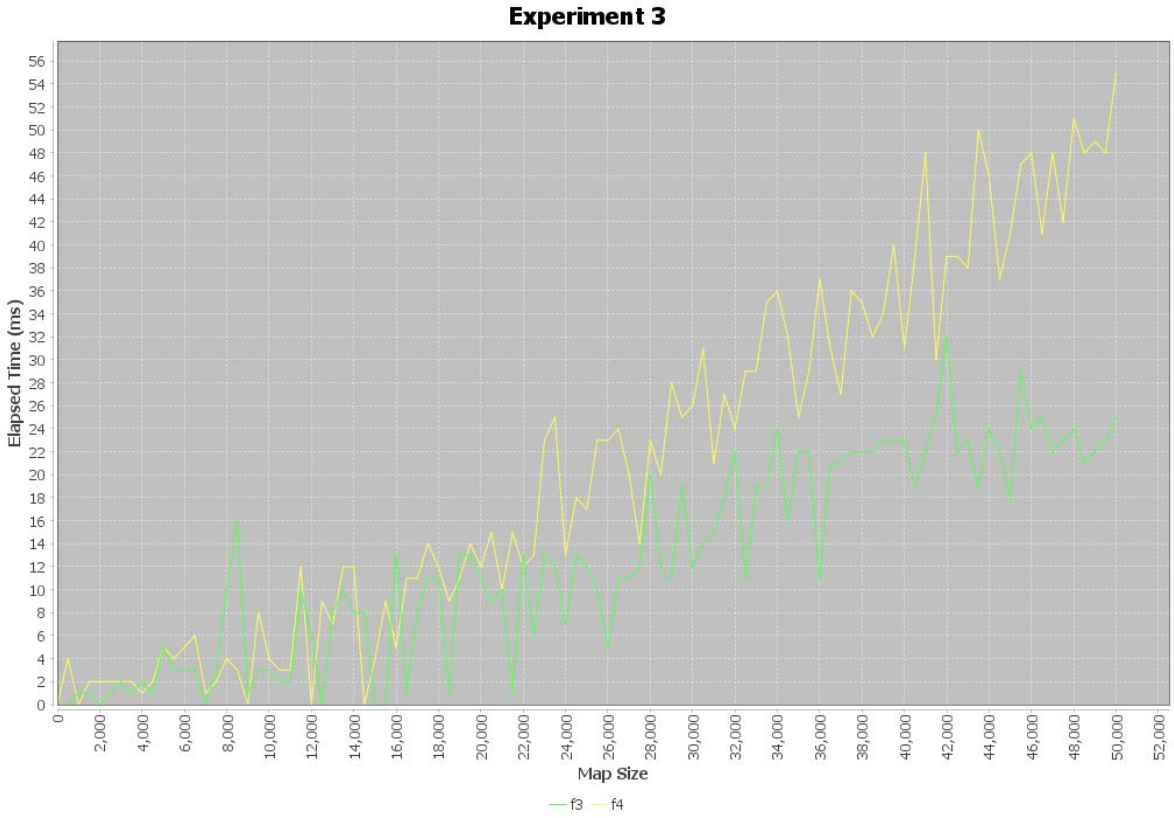
Experiment 2:

Prediction: runtime f1 > f2 > f4 > f3. Thus, f2 will be asymptotically slowest, f3 will be asymptotically fastest.

Discussion:

1. Hashcode for runtime1 uses the sum of the first 4 chars while for runtime2, it uses the sum of all chars. Therefore, the potential values are more spread out(more buckets) for runtime2, which indicates less resizing and that when key pairs are distributed, the middle chains(with the highest counts in the bell-shape histograms) should have a smaller length. This, in turn, reduces the runtime for f2.
2. From the histogram, it is obvious that the output distribution of fakeString3 is about uniform, while the other two have bell-shaped distribution. Therefore, when looking for desired elements in arrayMap associated with middle indexes, fakeString1 and 2 has much more element to look through one by one than fakeString3. Therefore, in practice, fakeString3's runtime order is closer to constant, while fakeString1 and 2's runtime order are closer to n.
3. AVLTreeMap requires to implement comparable interface. This is different because AVLTree has to be able to compare the key object to determine whether it should go through left branch or right branch, while arrayMap and chainedHashMap place key pairs using indexes which is independent of whether the element is "large or small."
4. Asymptotic runtime of methods from fastest to slowest:
   Fastest: FakeString3 > FakeString > FakeString2 > FakeString1 (slowest).
   From 1) and 2) we know that FakeString3 > FakeString2 > FakeString1. Since chainedHashMap uses a resizingLoadFactor closer to 1, its average chain length should be close to 1, while AVL tree requires a greater factor of operations for rotation. Thus, though both fakeString3 and fakeString should be really fast, fakeString3(chainedHashMap with good hashcode) should be slightly faster than fakeString(AVLTreeMap).


Experiment 3
Elapsed Time (ms)
Map Size
f1 f2 f3 f4


Experiment 3
Elapsed Time (ms)
Map Size
f3 f4

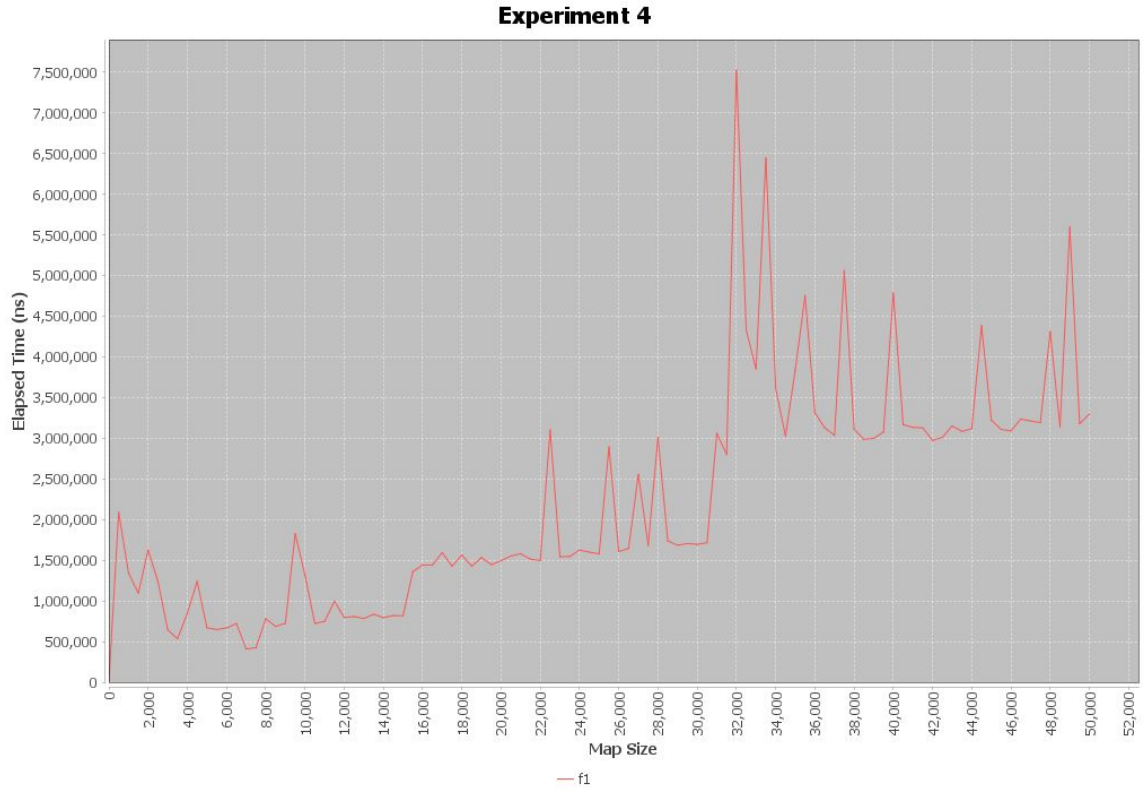
Experiment3:

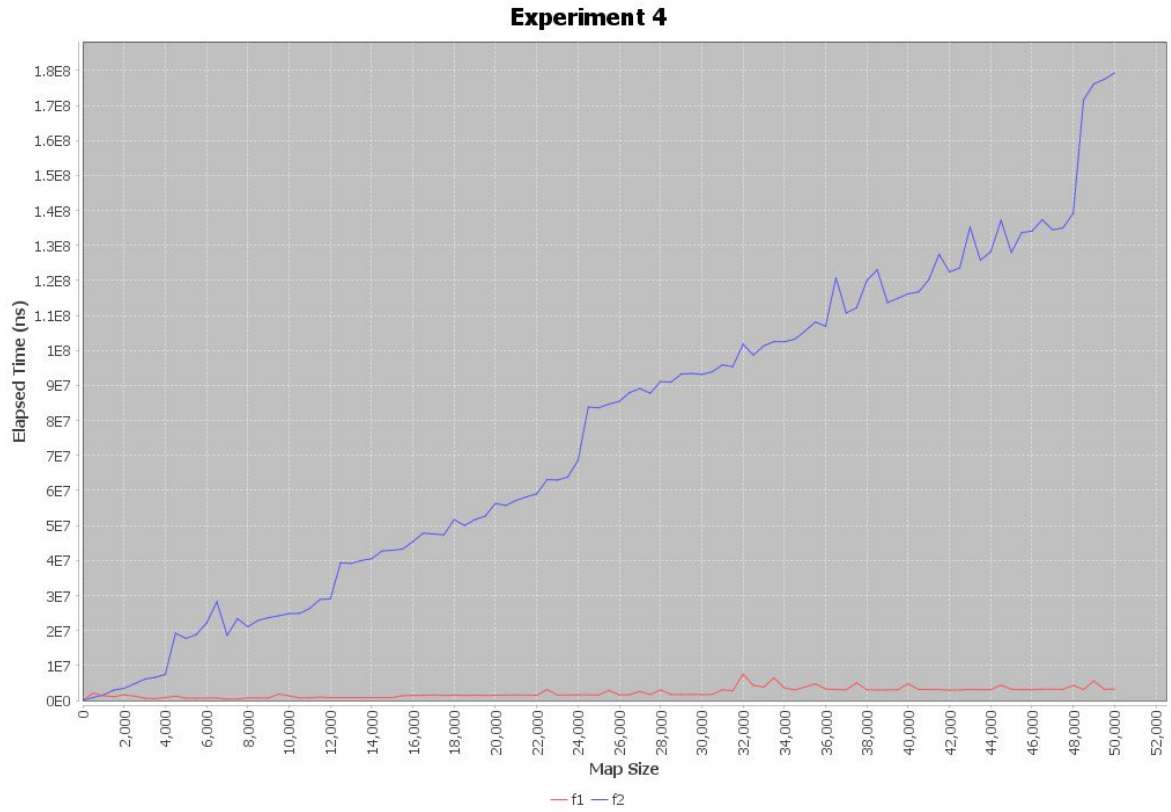
Prompts:

1. Runtime1 and runtime2 differ in its resizingLoadFactor.
2. Runtime1 will run faster. Asymptotically, runtime1 runs on about constantly, while runtime2 runs with about n for its runtime order.

Discussion:

1. Using a resizingLoadFactor of 300 will make the chain length very large and thus requires more frequent resizing of arrayMaps to be able to carry so many elements. Also, when resizing, the put() method runs on n to copy the data, thus requires a longer runtime.
2. An extremely small resizingLoadFactor might help reduce the runtime, but also requires frequent resizing, taking more space for array which has most parts empty or has only a few elements in it.


Experiment 4
Elapsed Time (ns)
7,500,000
7,000,000
6,500,000
6,000,000
5,500,000
5,000,000
4,500,000
4,000,000
3,500,000
3,000,000
2,500,000
2,000,000
1,500,000
1,000,000
500,000
0
0
2,000
4,000
6,000
8,000
10,000
12,000
14,000
16,000
18,000
20,000
22,000
24,000
26,000
28,000
30,000
32,000
34,000
36,000
38,000
40,000
42,000
44,000
46,000
48,000
50,000
52,000
Map Size
f1


Experiment 4
Elapsed Time (ns)
1.8E8
1.7E8
1.6E8
1.5E8
1.4E8
1.3E8
1.2E8
1.1E8
1E8
9E7
8E7
7E7
6E7
5E7
4E7
3E7
2E7
1E7
0E0
0
2,000
4,000
6,000
8,000
10,000
12,000
14,000
16,000
18,000
20,000
22,000
24,000
26,000
28,000
30,000
32,000
34,000
36,000
38,000
40,000
42,000
44,000
46,000
48,000
50,000
52,000
Map Size
f1
f2