HW5
Jia Qiao Lu, Dan Mu
CSE 12
29 October 2017

HW5

Part 1.2: Explanation

11. $n^2 + \frac{n}{2} + 1 = \theta(n^3)$ False

From the definition of Big-Theta,

1. Proof of $n^2 + \frac{n}{2} + 1 = O(n^3)$:

   To determine whether $n^2 + \frac{n}{2} + 1$ is $O(n^3)$, we need to determine whether witnesses C and n_0 exist, so that $n^2 + \frac{n}{2} + 1 \leq C(n^3)$ whenever $n > n_0$.

   If C and n_0 are witnesses, the inequality that $n^2 + \frac{n}{2} + 1 \leq C(n^3)$ holds for all $n > n_0$.

   Observe that the inequality $n^2 + \frac{n}{2} + 1 \leq C(n^3)$ is equivalent to the inequality $\frac{1}{n} + \frac{1}{2n} + \frac{1}{n^3} \leq C$, which follows by dividing both sides by n^3.

   Find constant C that will satisfy:

   When n = 1, $1 + \frac{1}{2} + 1 = \frac{5}{2}$.

   When n = 2, $\frac{1}{2} + \frac{1}{4} + \frac{1}{8} = \frac{7}{8}$.

   When n = 3, $\frac{1}{3} + \frac{1}{6} + \frac{1}{27} = \frac{29}{54}$.

   We choose C = 2 and $n_0 = 2$ to satisfy.

2. Proof of $n^3 = O(n^2 + \frac{n}{2} + 1)$:

   We assume witnesses C and n_0 exist, so that of $n^3 \leq C(n^2 + \frac{n}{2} + 1)$ whenever $n > n_0$.

   If C and n_0 are witnesses, the inequality that $n^3 \leq C(n^2 + \frac{n}{2} + 1)$ holds for all $n > n_0$.

   Observe that $n^2 + \frac{n}{2} + 1 \leq 3n^2$, which $\frac{n}{2} < n^2$ and $1 < n^2$, then $n^3 = n * n^2 > 3n^2$. No matter what C is, n can be made arbitrarily large. It follows that no witnesses C and n_0 for big-O relationship. Hence, n^3 *is not* $O(n^2 + \frac{n}{2} + 1)$.

Conclusion: Therefore, we can prove $n^2 + \frac{n}{2} + 1$ *is not* $\theta(n^3)$ by the definition of Big-Theta.

12. $\frac{1}{n^{100}} + log64 = \theta(1)$ True

From the definition of Big-Theta,

1. Proof of $\frac{1}{n^{100}} + log64 = O(1)$:

   To determine whether $\frac{1}{n^{100}} + log64$ is $O(1)$, we need to determine whether witnesses C and n_0 exist, so that $\frac{1}{n^{100}} + log64 \leq C(1)$ whenever $n > n_0$.

   If C and n_0 are witnesses, the inequality that $\frac{1}{n^{100}} + log64 \leq C(1)$ holds for all $n > n_0$.

   Observe that the inequality $\frac{1}{n^{100}} + log64 \leq C(1)$ is equivalent to the inequality $\frac{1}{n^{100}} + log64 \leq C$, which follows by dividing both sides by 1.

   Find constant C that will satisfy:

   When n = 1, $\frac{1}{1^{100}} + log64 = 7$.

When n = 2, $\frac{1}{2^{100}} + log64 = 6$

When n = 3, $\frac{1}{3^{100}} + log64 = 6$.

C = 8 and $n_0 = 1$

Since when n is approaching infinity, $\frac{1}{n^{100}}$ will be infinitely close to 0.

Therefore $\frac{1}{n^{100}} + log64$ will never be greater than 8.

2. Proof of $1 = O(\frac{1}{n^{100}} + log64)$:

To determine whether 1is $O(\frac{1}{n^{100}} + log64)$, we need to determine whether witnesses C and n_0 exist, so that $1 \leq C(\frac{1}{n^{100}} + log64)$: whenever $n > n_0$.

If C and n_0 are witnesses, the inequality that$1 \leq C(\frac{1}{n^{100}} + log64)$ holds for all $n > n_0$.

Observe that the inequality $1 \leq C(\frac{1}{n^{100}} + log64)$ is equivalent to the inequality $\frac{1}{\frac{1}{n^{100}} + log64} \leq C$, which follows by dividing both sides by $\frac{1}{n^{100}} + log64$.

Find constant C that will satisfy:

When n = 1,$\frac{1}{\frac{1}{1^{100}} + log64} = \frac{1}{7}$.

When n = 2, $\frac{1}{\frac{1}{2^{100}} + log64} = \frac{1}{6}$.

When n = 3, $\frac{1}{\frac{1}{3^{100}} + log64} = \frac{1}{6}$.

We choose C = 1 and $n_0 = 1$ to satisfy.

Conclusion: Therefore, we can prove that $\frac{1}{n^{100}} + log64\ is\ \theta\ (1)$.

13. $nlogn + n = \theta(logn)$ False

From the definition of Big-Theta

Proof of $nlogn + n = O(logn)$:

We assume witnesses C and n_0 exist, so that of $nlogn + n \leq C(logn)$ whenever $n > n_0$.

If C and n_0 are witnesses, the inequality that $nlogn + n \leq C(logn)$ holds for all $n > n_0$.

Observe that the inequality $nlogn + n \leq C(logn)$ is equivalent to the inequality $n + \frac{n}{logn} \leq C$, which follows by dividing both sides by $logn$. No matter what C is, n can be made arbitrarily large. It follows that no witnesses C and n_0 for big-O relationship. Hence, $nlogn + n\ is\ not\ O(logn)$.

Conclusion: Therefore, we can prove $nlogn + n\ is\ not\ \theta(logn)$ by the definition of Big-Theta.

14. $2^n + n = \theta(n)$ False

From the definition of Big-Theta

Proof of $2^n + n = O(n)$:

We assume witnesses C and n_0 exist, so that of $2^n + n \leq C(n)$, whenever $n > n_0$.

If C and n_0 are witnesses, the inequality that $2^n + n \leq C(n)$ holds for all $n > n_0$.

Observe that the inequality $2^n + n \leq C(n)$ is equivalent to the inequality$\frac{2^n}{n} + 1 \leq C$,

which follows by dividing both sides by n. No matter what C is, $\frac{2^n}{n} + 1$ can be made arbitrarily large. It follows that no witnesses C and n_0 for big-O relationship. Hence, $2^n + n \; is \; not \; O(n)$.

Conclusion: Therefore, we can prove $2^n + n \; is \; not \; \theta(n)$ by the definition of Big-Theta.

15. $\frac{n^5}{n^2} + logn = \theta(n^3)$ True

From the definition of Big-Theta,

1. Proof of $\frac{n^5}{n^2} + logn = O(n^3)$:

To determine whether $\frac{n^5}{n^2} + logn$ is $O(n^3)$, we need to determine whether witnesses C and n_0 exist, so that of $\frac{n^5}{n^2} + logn \leq C(n^3)$, whenever $n > n_0$.

If C and n_0 are witnesses, the inequality that $\frac{n^5}{n^2} + logn \leq C(n^3)$ holds for all $n > n_0$.

Observe that the inequality $\frac{n^5}{n^2} + logn \leq C(n^3)$ is equivalent to the inequality $1 + \frac{logn}{n^3} \leq C$, which follows by dividing both sides by n^3 .

Find constant C that will satisfy:

When n = 1, $1 + \frac{log1}{1^3} = 1$.

When n = 2, $1 + \frac{log2}{2^3}$ =1.125.

When n = 3, $1 + \frac{log3}{3^3} == 1.059$.

We choose C = 2 and $n_0 = 2$ to satisfy.

2. Proof of $n^3 = O(\frac{n^5}{n^2} + logn)$:

To determine whether n^3 is $O(\frac{n^5}{n^2} + logn)$, we need to determine whether witnesses C and n_0 exist, so that $n^3 \leq C(\frac{n^5}{n^2} + logn)$:: whenever $n > n_0$.

If C and n_0 are witnesses, the inequality that $n^3 \leq C(\frac{n^5}{n^2} + logn)$ holds for all $n > n_0$.

Observe that the inequality $n^3 \leq C(\frac{n^5}{n^2} + logn)$ is equivalent to the inequality $\frac{n^3}{\frac{n^5}{n^2}+logn} \leq C$, which follows by dividing both sides by $\frac{n^5}{n^2} + logn$.

Find constant C that will satisfy:

When n = 1, $\frac{1^3}{\frac{1^5}{1^2}+log1}$ =1.

When n = 2, $\frac{2^3}{\frac{2^5}{2^2}+log2} = \frac{8}{9}$.

When n = 3, $\frac{3^3}{\frac{3^5}{3^2}+log3} = 0.94$.

We choose C = 1 and $n_0 = 1$ to satisfy.

Conclusion: Therefore, we can prove that $\frac{n^5}{n^2} + logn \; is \; \theta(n^3)$.

Part 2.2: Explanation (Proof)

2. num = 0; → 1
   for (i = 0; i <= n * n; i = i+2) → $\frac{n^2}{2} + 1$
      num = num + 2; → $\frac{n^2}{2}$

Answer: Running time is O(n^2)

Explanation: There is a for loop that runs ($\frac{n^2}{2} + 1$) times. Each time the loop runs it executes 1 instruction in the loop header and 1 instruction in the body of the loop. The total number of instructions is 2 * $\frac{n^2}{2}$ + 1 (for the last loop check) = $2n^2 + 1$ = O(n^2).

3. num = 0; → 1
   for (i = 1; i <= n; i = i*2) → $\log n + 1$
      num++; → $\log n$
   Answer: Running time is O(log(n))
   Explanation: There is a for loop that runs log (n) times. Each time the loop runs it executes 1 instruction in the loop header and 1 instruction in the body of the loop. The total number of instructions is 2 * log (n) + 2 (for the last loop check) = $2 \log n + 2$ = O($\log n$).

5. p = 10; → 1
   num = 0; → 1
   plimit = 100000; → 1
   for (i = p; i <= plimit; i++) → $99990 + 1$
      for (j = 1; j <= i; j++) → $99990(\frac{99990(99990+1)}{2} + 1)$

         num = num + 1; → $99990(\frac{99990(99990+1)}{2})$
   Answer: Running time is O(1)
   Explanation: There is two for loop. The outside for loop runs 100000-10 = 99990 times. Each time the loop runs it executes 1 instruction in the loop header and inside for loop runs i times. Also, it executes 1 instruction in the body of the loop. The total number of instructions is $2(99990) + 2\left(\frac{99990^2(99990+1)}{2}\right) + 4$

9. for (i = 0; i < n; i++) { → $n + 1$
      smallest = i; → n
      for (j = i+1; j <= n; j++) { → $n((n-1) + 1)$
         if (a[j] < a[smallest]) → $n(n-1)$
         smallest = j; → $O(?)$
      }

```
        swap(a, i, smallest); // has three instructions → 3n
    }
```

Answer: Running time is $O(n^2)$

Explanation: There is two for loop. The outside for loop runs n times. Each time the loop runs it executes 1 instruction in the loop header and 1 instruction in the body of the loop. The inside for loop runs n times. Each time the loop run it executes 1 instruction in the loop header and 1 instruction in the body of the loop. The total number of instructions is $6n + 2\big(n(n-1)\big) + 1$= $O(n^2)$.

10.
```
num = 0; → 1
 i = 0; → 1
 while (i < n) { → n + 1
        j = 0; → n
        while (j < 100) { → n(100 + 1)
              // constant time operations
              j++; → 100n
        }
        i++; → n
 }
```

Answer: Running time is O(n)

Explanation: There is two while loop. The outside while loop runs n times. Each time the loop runs it executes 1 instruction in the loop header and 1 instruction in the body of the loop. The inside while loop runs 1 times. The total number of instructions is $204n + 3$ = O(n).

Part 3 - Running time of Sorted Singly Linked List and Sorted Array functions

1. Copying the Singly Linked list:
   Running time: Θ(n). We need to create a new linked list and add all elements to the linked list. Because each time we add one element, we need to go through the linked list to find and get the element then copy it into the new linked list.

2. Copying the Sorted Array list:
   Running time: Θ(n). We need to create a new array list and add all elements to the array list. Because each time we add one element, we need to go through the linked list to find and get the element then copy it into the new linked list.

3. Adding a value to the middle of the list in Sorted Singly Linked List. (where you do not want to destroy the overall order of the elements)
   Running time: Θ(n). We need to go through the linked list in order to find the middle of the list, which has a run time of Θ(n/2). Then, add a new node and set the element inside this node.

4. Adding a value to the middle of the list in Sorted Array. (where you do not want to destroy the overall order of the elements)
   Running time: Θ(n). We need to find the middle of the array and then add the new element. We need to move everything over one cell to the right, which has a run time of Θ(n/2). Therefore, it takes Θ(n).

5. Removing the value from the end of the Sorted Singly Linked List (the last index location).
   Running time: Θ(n). You have a pointer to the tail node in the list, so we can directly remove an element in the tail. However, because it is singly linkedlist, we have to go through all the elements in order to find the new last node and set this node as the new tail, which has a runtime of Θ(n).

6. Removing the value from the end of the Sorted Array (the last index location).
   Running time: Θ(1). We can directly delete the element of the end of the Sorted Array and there is nothing else need to be changed.

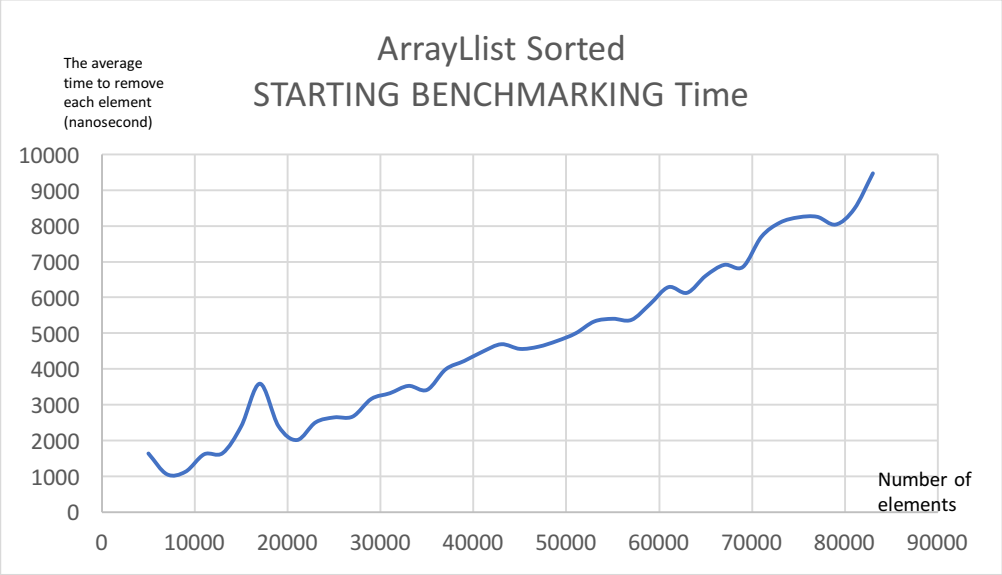
7. Removing the first value from the Sorted Singly Linked List. (Note that the remaining elements will now have an index value that is one lower than before).
   Running time: Θ(1). You have a pointer to the head node in the list, so we can directly remove an element in the head and then set the next node of the list as the new head, which has a runtime of Θ(1).

8. Removing the first value from the Sorted Array. (Note that the remaining elements will now have an index value that is one lower than before).
   Running time: Θ(n). We can directly delete the first element in the index, but we need to move everything one cell to the left in order to resort the array.

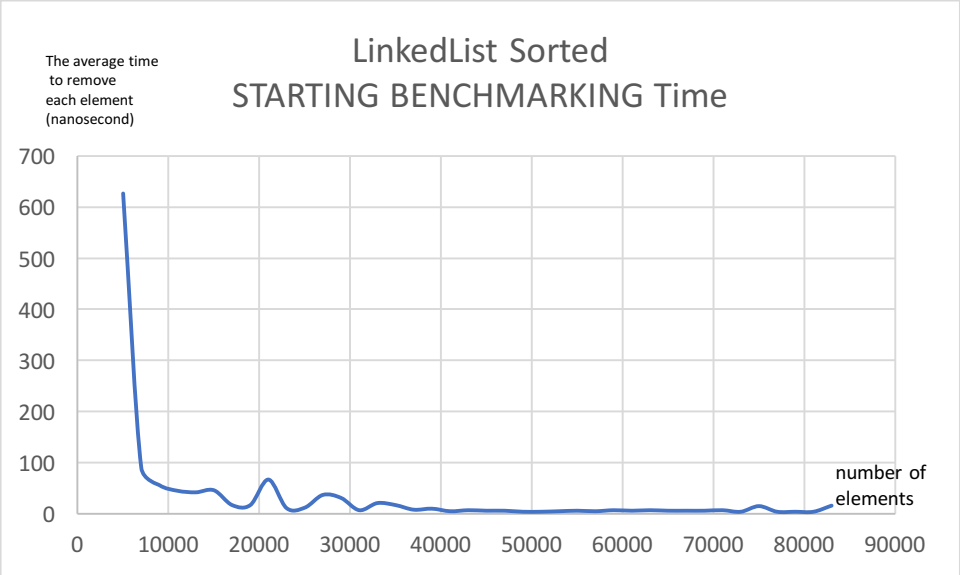
9. Determining whether the Linked List contains some value v.
   Running time: Θ(n). We need to go through the Linked List and check whether it contains the value we want to find. The worst case is that the value is not contained in the list, which has a run time of Θ(n). `

10. Determining whether the Array contains some value v.
    Running time: Θ(n). We need to go through the array and check whether it contains the value we want to find. The best case is that the value is the first element in the index, which has a run time of Θ(1). The worst case is that the value is not contained in the array, which has a run time of Θ(n).

Part 4 - Performing Runtime exploration

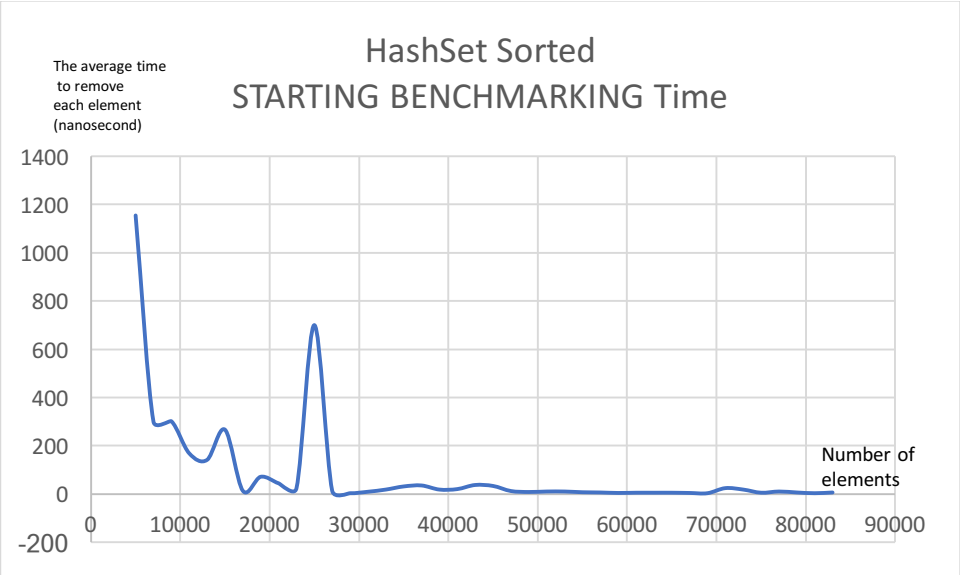


In a Arraylist, when elements are removed in sorted order, the running time we expected is O(n). As mentioning in the instruction, the same order the elements went in, you are always removing at index position 0. Because we reset the index in the list, and move everything one cell to the left in order to resort the array. After deleing each element, it is supposed to be O(n). Also, deleting each elements is O(1). Therefore, we assumed that it has a running time of O(n) + O(1). After checking the data and the graph, and ignoring the small waves, it showed that the observed running times is approaching to linear increasing, which is O(n). Therefore, the graph supported our hypothesis.

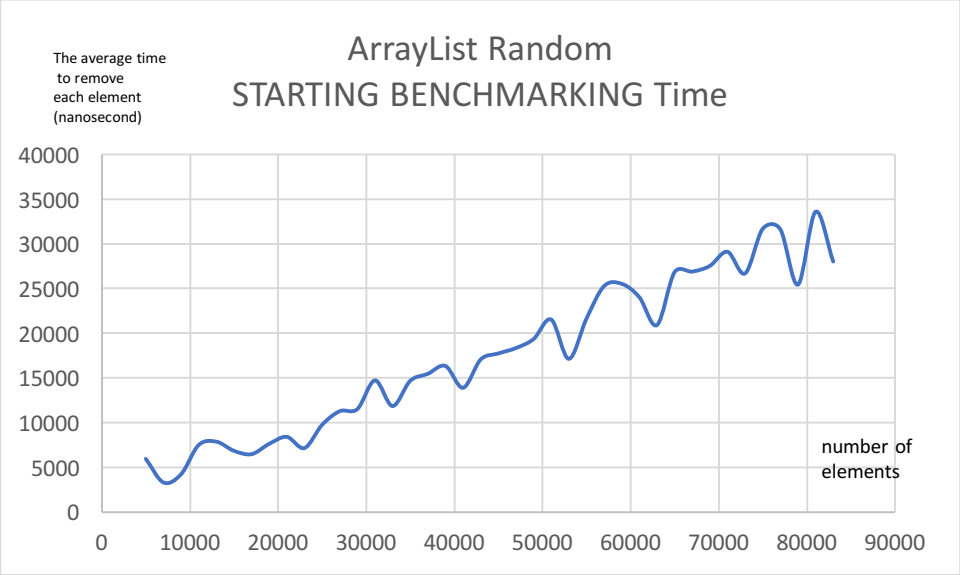


In a LinkedList, when elements are removed in sorted order, the running time we expected is O(1). As mentioning in the instruction, the same order the elements went in, they are always

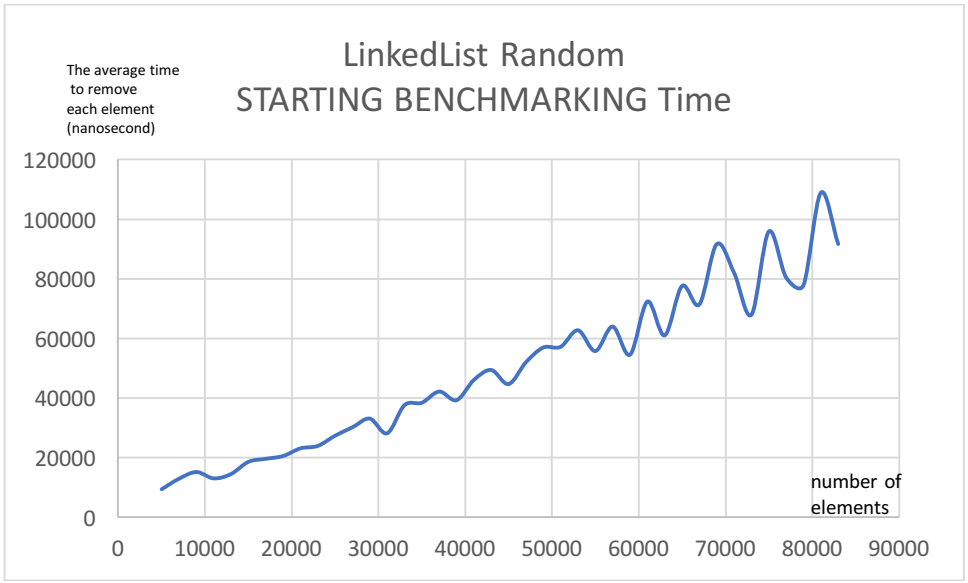
removing at the head of the list. Because we only need to delete the head, it is supposed to be O(1). Also, after deleting each elements, we need to set the next node as the new head, which is O(1). Therefore, we assumed that it has a running time of O(1) + O(1). After checking the data and the graph, and ignoring the small part which has an extreme big wave, it showed that the observed running times is O(1) and supported our hypothesis.



In a HashSet, when elements are removed in sorted order, the running time we expected is O(1). As mentioning in the instruction, the same order the elements went in, elements are removed in sorted order, which was the same order they went in. Because we only need to find the key-value pair and then delete the element directly, it is supposed to be O(1). After checking the data and the graph, and ignoring the small part which has extreme big waves, it showed that it approaching to a constant. Therefore, the observed running times is O(1), which supported our hypothesis.

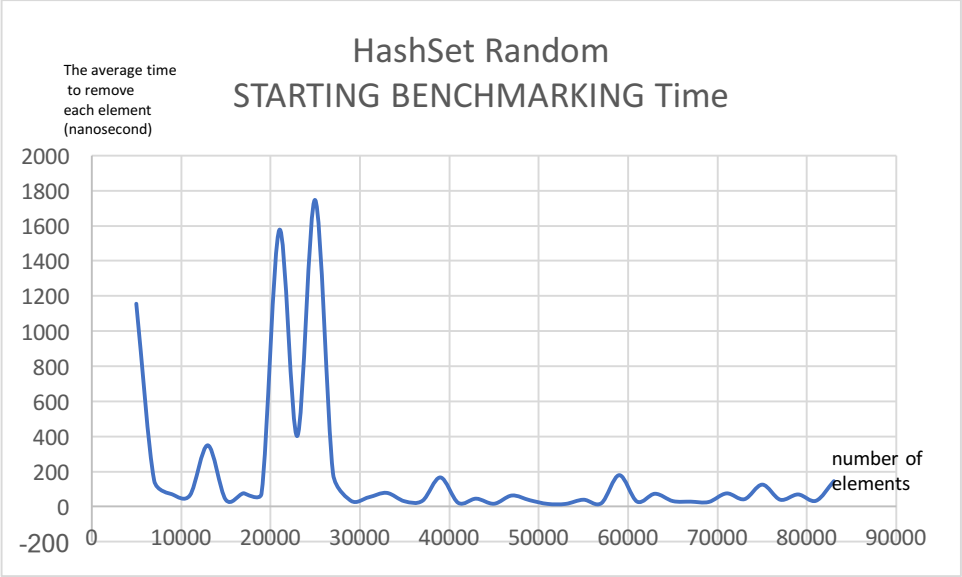


In a Arraylist, when elements are removed in random, the running time we expected is O(n). Because we reset the index in the list, move everything on the right of the deleted elements one cell to the left in order to resort the array. After deleing each element, it is supposed to be O(n). Also, deleting each elements is O(1). Therefore, we assumed that it has a running time of O(n) + O(1). After checking the data and the graph, and ignoring the small waves, it showed that it is approaching to linear increasing. Therefore, the observed running times is O(n), which supported our hypothesis.



In a LinkedList, when elements are removed in the random, the running time we expected is O(n). Because we need to go through the list in order to find the element to delete, it is supposed to be O(n). Also, after deleting each elements, we need to connect the next node and previous node, which each is O(1). Therefore, we assumed that it has a running time of O(n) + O(1) + O(1). After checking the data and the graph, and ignoring the small waves, it showed that it is

approaching to linear increasing. Therefore, the observed running times is O(n), which supported our hypothesis.



In a HashSet, when elements are removed in the random, the running time we expected is O(1). Because we only need to find the key-value pair and then delete the element directly, it is supposed to be O(1). After checking the data and the graph, and ignoring the small part which has extreme big waves, it showed that it is approaching to a constant. Therefore, the observed running times is O(1) and supported our hypothesis.