CSC111 Assignment 1: Linked Lists

Jamie Yi

February 1, 2022

Part 1: Faster Searching in Linked Lists

1. Complete this part in the provided `a1_part1.py` starter file. Do **not** include your solution in this file.
2. Complete this part in the provided `a1_part1_test.py` starter file. Do **not** include your solution in this file.
3. (a) -Let $n \in \mathbb{Z}^+$
   -Let `linky` be a `LinkedList` with length n containing the numbers $0, 1, 2, ..., n-1$ in that order
   -Let $m \in \mathbb{Z}^+$, let m represent the number of times `linky.__contains__(n-1)` is called

   The cost of the assignment statements at the start is constant time, so we'll count that as 1 step.

   Each inner loop iteration also comprises of solely constant time statements, whether or not the if-statement triggers. Therefore, I will also count each loop iteration as one step.
   The loop will iterate exactly n times per call. This is because the early return in the if-statement triggers on the last item of the list. Even if the early return didn't trigger, it would still be the last loop iteration since the next item in the list was `None`. Therefore, the while loop iterates n times per call which is the length of the `linky`. With each iteration taking 1 step in constant time.

   The last return statement is constant time, however since we are always searching for an item in the list it will never be called.

   The method will run $1 + 1 \cdot (n)$ times per call. If it's called m times then the total running time will be $(m)(1 + n)$ steps. Therefore we have a running time of $\Theta(m \cdot n)$

   (b) **Heuristic 1**
   -Let $n \in \mathbb{Z}^+$
   -Let `linky` be a `MoveToFrontLinkedList` with length n containing the numbers $0, 1, 2, ..., n-1$ in that order
   -Let $m \in \mathbb{Z}^+$, let m represent the number of times `linky.__contains__(n-1)` is called

   The cost of the assignment statements at the start is constant time, so we'll count that as 1 step.

   Each inner loop iteration also comprises of solely constant time assignment statements, so I will also count each loop iteration as one step.
   Since we're initially searching for the item at the end of the list (that exists in the list), we will only look at the second condition of the while loop. On the first call the loop will iterate $n-1$ times since after the second last iteration, `curr.item == n-1` which is what we want.
   However, after the first method call the the list is mutated such that $n-1$ is moved to the front. Then for all subsequent loops, after the first two assignment statements, `curr.item == item` so the while loop will never iterate during those calls. It only iterates on the first call.

   The last if-else block is entirely comprised of constant time statements, so I will count the whole block as one step.

On the first call, the loop will run $(1 + (n-1) + 1)$ steps. Then for all subsequent call the loop will only run constant time statements(as described above since the loop no longer runs), we will simplify this to one step per subsequent call. The total running time will be $(1 + (n-1) + 1) + 1 \cdot (m-1)$ steps. Therefore we have a running time of $\Theta(m+n)$

Heuristic 2
-Let $n \in \mathbb{Z}^+$
-Let `linky` be a `SwapLinkedList` with length n containing the numbers $0, 1, 2, ..., n-1$ in that order
-Let $m \in \mathbb{Z}^+$, let m represent the number of times `linky.__contains__(n-1)` is called

The cost of the assignment statements at the start is constant time, so we'll count that as 1 step.

Each inner loop iteration also comprises of solely constant time assignment statements, so I will also count each loop iteration as one step.
If $m < n$ Since $n-1$ swaps places with the item in front of it every time it is searched for, in order for it to reach the front of the list it must be searched for $n-1$ times. So as long as $m < n$, the while loop will run each iteration since `curr.item != item` to begin with. The first call, the loop will iterate $n-1$ times since after the second last iteration, `curr.item == n-1` which is what we want. Then the next call, the loop will iterate one less times since $n-1$ has moved up a spot and this pattern will continue until $n-1$ is the first item. We can represent this as the loop taking a total $\sum_{i=1}^{m}(n-i)$ iterations over the course of m function calls. We know all the terms of the summation will be non-negative since $m < n$.
If $m \geq n$ After the $n-1$th method call we know $n-1$ will be at the front of the list and therefore the while loop will no longer run. So in this case, the loop will iterate exactly $\sum_{i=1}^{n-1}(n-i)$ times before subsequent method calls no longer run the while loop and only constant time statements are executed.

The last if-else block is entirely comprised of constant time statements, so I will count the whole block(and the nested if statements) as one step.
The total running time is as follows:
If $m < n$
$m(1) + \sum_{i=1}^{m}(n-i) + m(1) = m + \sum_{i=1}^{m}(n) - \sum_{i=1}^{m}(i) + m = m + mn - \frac{m(m+1)}{2} + m = m(2+n) - \frac{m(m+1)}{2}$ steps.
The running time in this case is $\Theta(m^2 + n)$

If $m \geq n$
$(n-1)(1) + \sum_{i=1}^{n-1}(n-i) + (n-1)(1) + (m-(n-1))(1) = (n-1) + \sum_{i=1}^{n-1}(n) - \sum_{i=1}^{n-1}(i) + (n-1) + (m-(n-1)) = 2(n-1) + n(n-1) - \frac{(n-1)n}{2} + m - n + 1 = m - n + (2+n)(n-1) - \frac{(n-1)n}{2} + 1$ steps.
The running time in this case is $\Theta(m + n^2)$

Heuristic 3
-Let $n \in \mathbb{Z}^+$
-Let `linky` be a `CountLinkedList` with length n containing the numbers $0, 1, 2, ..., n-1$ in that order
-Let $m \in \mathbb{Z}^+$, let m represent the number of times `linky.__contains__(n-1)` is called

The cost of the assignment statements at the start is constant time, so we'll count that as 1 step.

Each inner loop iteration also comprises of solely constant time assignment statements, so I will also count each loop iteration as one step.
Since we're initially searching for the item at the end of the list (that exists in the list), we will only look at the second condition of the while loop. On the first call the loop will iterate $n-1$ times since after the second last iteration, `curr.item == n-1` which is what we want.
However, after the first method call the the list is mutated such that $n-1$ is moved to the front. Since no other items are ever searched for $n-1$'s `access_count` will always be greater than every other item's `access_count` after the first loop. Then for all subsequent loops, after the first two assignment statements, `curr.item == item` so the while loop will never iterate during those calls. It only iterates on the first call.

The while loop in the if-else block will never run because for other all items in the list, `item.access_count`

$<$ `n-1.access_count` because after the first search, `n-1.access_count` is increased by 1 before the while loop runs and it will keep increasing for all subsequent calls. Thus since all the other statements in the if-else block are constant time, I will count the whole block(and the nested if statements) as one step.

On the first call, the loop will run $(1 + (n-1) + 1)$ steps. Then for all subsequent call the loop will only run constant time statements(as described above since the loop no longer runs), we will simplify this to one step per subsequent call. The total running time will be $(1+(n-1)+1)+1 \cdot (m-1)$ steps. Therefore we have a running time of $\Theta(m+n)$

4. -Let $n \in \mathbb{Z}^+$, assume $n > 2$
-Let `linky1` be a `MoveToFrontLinkedList` with length n containing the numbers $0, 1, 2, ..., n-1$ in that order
-Let `linky2` be a `SwapLinkedList` with length n containing the numbers $0, 1, 2, ..., n-1$ in that order
-(Initially `linky1` and `linky2` should look identical and contain the exact same items
-Let $m \in \mathbb{Z}^+$, where there is some sequence of m search operations to be performed on `linky1` and `linky2`

WTS: If T_1 is the running time for `linky1` and T_2 is the running time for `linky2`, then $T_2 - T_1 \notin \mathcal{O}(1)$
*In this analysis I will only be counting the number of distinct nodes traversed by `__contains__` per operation

When fixing n, it's important that it's greater than two because with MoveToFrontLinkedLists and SwapLinkedList of length 1 and 2, performing search operations will mutate them in identical ways. If the lists each have one element, searching for that one element over and over again will do nothing. If the lists each have 2 elements, either i) searching for the first items will not mutate the lists or ii) searching for the second item will mutate both lists in the same way, in that the two items just swap places.

Answer
Let m be this sequence of search operations:
`-lst.__contains__(n-1)`
`-lst.__contains__(n-2)`
...
`-lst.__contains__(2)`
`-lst.__contains__(0)`
in that order, applied to both `linky1` and `linky2`

For example, if $n = 3$, then the sequence will be just:
`-lst.__contains__(2)`
`-lst.__contains__(0)`

For `linky1`(the MoveToFrontLinkedList), what this sequence does is move all the items starting from the last item to the third item inclusive to the front. Then it will search for 0 which was originally the first item. 0 will always be be the second last item in the list, since 1 never searched for and it comes after 0. Each search operation will be looking for the item that is at the end of the list at that time, so all the statements except the last(quantified as the first $m-1$ statements) will traverse n distinct nodes, or the length of the list. Since the last statement will be looking for the second last item, it will traverse $n-1$ distinct nodes. Therefore this sequence will have a T_1 of $(m-1)(n) + (n-1)$ steps.

For `linky2`(the SwapLinkedList), what this sequence does is to swap around all the items except the first item in the list. In order to move 0 in this list, `-linky2.__contains__(1)` must be called so 0 and 1 switch places, but that statement will always be absent from sequence m so throughout the whole sequence, 0 will remain as the first item. Given how swapping is less striaghtforward to calculate multiple operations of compared to moving to the front, I will make things easier for myself and consider the maximum number of distinct nodes that a given search operation will traverse. Let's say that for all the first $m-1$ operations, a maximum of n distinct nodes are traversed per operation. Since we know that 0 remains unmoved, we know in any given sequence m, `linky2.__contains__(0)` will only traverse one distinct node - the first one. Therefore we see that this sequence will have a maximum T_2 of $(m-1)(n) + 1$ steps.

Now let's subtract T_2 from T_1 knowing that we're using the maximum possible value of T_2, we can safely say that this is the smallest possible difference.
$T_2 - T_1 = ((m-1)(n) + (n-1)) - ((m-1)(n) + 1) = (m-1)(n) - (m-1)(n) + (n-1) - 1 = n - 2$ steps.
$n - 2 \notin \mathcal{O}(1)$ as we wanted.
Linear running time cannot upper bound by a constant and since this is the minimum possible difference, we have shown that there is a sequence m where for any n, $T_2 - T_1 \notin \mathcal{O}(1)$.

Part 2: Linked List Visualization

Complete this part in the provided `a1_part2.py` starter file. Do **not** include your solution in this file.