CSCE 221
Problem Set 8

Jacob Purcell, Texas A&M, Student

I.

For O(N), a linear search will be sufficient. Have an iterator store its starting pointer, then follow each subsequent pointer until it finds a match. This will iterate through every value and requires O(N) time to compute. With O(1) extra space, another iterator may be added but traverses 2 nodes instead of one, eventually the fast iterator will catch up to the slow iterator. While the slow iterator is still testing against whether it makes a circle, the fast iterator can test whether it has caught up to the slow iterator.

Algorithm 1 Check for Circular List

Require: $N\ is\ an\ element\ of\ \mathbb{N}$
  $slow = head, fast = head$
  while $slow! = nullptr \land fast! = nullptr$ **do**
    $slow \rightarrow next(O(N))$
    $fast \rightarrow next \rightarrow next(one\ extra\ calculation)$
    if $slow == head || fast == slow$ **then**
      $Return\ true$
    end if
  end while
  return $false$

II.

A.

Algorithm 2 Selfadjusting List

Require: $Array$
  $temporary\ storage = requested\ value\ O(1)$
  $iterator = requested\ index\ O(N)$
  while $iterator\ is\ greater\ than\ zero$ **do**
    $copy\ previous\ array\ element\ to\ current\ location\ O(1)$
    $decrement\ iterator\ O(1)$
    $(removes\ requested\ value\ from\ array\ and\ creates\ a\ free\ slot\ at\ front)$
  end while$O(N)$
  $assign\ temporary\ storage\ to\ Array\ position\ zero\ O(1)$
  return $temporary\ storage$

B.

Algorithm 3 Selfadjusting List

Require: $Doubly\ Linked\ List$
  $cursor = head\ O(1)$
  while $cursor\ is\ not\ nullptr\ and\ cursor\ data\ is\ not\ requested\ data$ **do**
    $cursor \rightarrow next\ O(1)$
    $(finds\ desired\ value\ )\ O(N)$
  end while
  $cursor\ data = temporary\ storage\ O(1)$
  $cursor\ previous\ next \rightarrow cursor\ next\ O(1)$
  $cursor\ next\ previous\ \rightarrow\ cursor\ previous\ O(1)$
  $(links\ list\ around\ the\ desired\ ptr)$
  $cursor\ next \rightarrow head\ O(1)$
  $cursor\ previous \rightarrow nullptr\ O(1)$
  $head = cursor\ O(1)$
  $(reorder\ list\ so\ last\ accessed\ ptr\ is\ at\ front)$
  return $temprorary\ storage$

C.

The time complexity for both implementations is the same at the asymptote (O(N)), however, the array implementation would be noticably slower for small datasets with a time complexity of O(2N). For each, the time complexity will drop to O(1) if the same element is requested more than once in a row.

D.

Every element has a certain probability (p_i) of being accessed. After being accessed, it is 100% likely that the value will be at the front of the list the next iteration with a probability p_i of being accessed again. Introducing $\hat{e_k}$, the index of the element, the last accessed element is always $\hat{e_0}$. After N accesses, the probability of being accessed n_i times is Np_i. k therefore holds the form $\frac{i}{n_i}$, where the index of the element is inversely related to it's probability. Finally, this means that the chance of finding any specific value in any position is ordered by $\boxed{\Sigma_{i=1}^{Elements} n_i \hat{e_k}}$.