EECS2011 Assignment 01

Araf A Alam
218124347
Sept. 30, 2022

Q1)

Efficiency for each:

A) Array (like the Java array)

(a) Search for an element. -> SLOW

The bigger the array, the more time it will take to search the array. Time complexity of O(n).

(b) Get the middle element, i.e., with the list size being n, return the ⌊n/2⌋-th element in the list. -> FAST

Time complexity of O(1). No loops are needed, always a constant number of operations.

(c) Insert a new element at the beginning of the list. -> SLOW

Need to move all the elements of the array forward by one if the beginning is not null. O(n). However, it's O(1) and extremely fast if the array is empty initially. But we consider the worst-case scenario in this case.

(d) Insert a new element at the end of the list. -> FAST

Time complexity of O(1), just add the new element to the list, arr[arr.length + 1] = new;

(e) Delete an element from the beginning of the list. -> SLOW

Time complexity of O(n). Need to move all the elements of the array down to overwrite the deleted index. But if we were allowed to just leave the first index as NULL, then it would be FAST as we can just set arr[0] to NULL and forget about it.

(f) Delete an element from the end of the list. -> FAST

Time complexity of O(1), just delete the last element, arr[n] = NULL where n is the length of the array.

B) singly-linked list, with "next" only, and with "head" only.

(a) Search for an element. -> SLOW

Time complexity of O(n).

(b) Get the middle element, i.e., with the list size being n, return the ⌊n/2⌋-th element in the list. -> SLOW

Time complexity of O(n).

(c) Insert a new element at the beginning of the list. -> FAST

Time complexity of O(1). A simple linked list insertion, always a constant # of operations.

(d) Insert a new element at the end of the list. -> SLOW

Time complexity of O(n) since we don't have a "tail."

(e) Delete an element from the beginning of the list. -> FAST

Time complexity of O(1). Just set the second element as the head.

(f) Delete an element from the end of the list. -> SLOW

Time complexity of O(n) since we don't have a "tail."

C. singly-linked list, with "next" only, and with "head" and "tail".

(a) Search for an element. -> SLOW

Time complexity of O(n).

(b) Get the middle element, i.e., with the list size being n, return the $\lfloor n/2 \rfloor$-th element in the list. -> SLOW

Time complexity of O(n).

(c) Insert a new element at the beginning of the list. -> FAST

Time complexity of O(1). A simple linked list insertion, always a constant # of operations.

(d) Insert a new element at the end of the list. -> FAST

Time complexity of O(1). Just insert as the next of the tail, then assign as tail.

(e) Delete an element from the beginning of the list. -> FAST

Time complexity of O(1). Just set the second element as the head.

(f) Delete an element from the end of the list. -> SLOW

Time complexity of O(n) since we don't have a "prev" that we can use to assign the tail after.

D. reverse singly-linked list, with "prev" only, and with "tail" only.

(a) Search for an element. -> SLOW

Time complexity of O(n).

(b) Get the middle element, i.e., with the list size being n, return the $\lfloor n/2 \rfloor$-th element in the list. -> SLOW

Time complexity of O(n).

(c) Insert a new element at the beginning of the list. -> SLOW

Assuming the beginning of the list is where head would be. Time complexity of O(n).

(d) Insert a new element at the end of the list. -> FAST

Time complexity of O(1), just set newNode.prev to the node at tail and set newNode as tail.

(e) Delete an element from the beginning of the list. -> SLOW

Assuming the beginning of the list is where head would be. Time complexity of O(n).

(f) Delete an element from the end of the list. -> FAST

Time complexity of O(1), set the second lastNode.prev as the new tail.

E. doubly-linked list with "next" and "prev", and with "head" and "tail".

(a) Search for an element. -> SLOW

Time complexity of O(n).

(b) Get the middle element, i.e., with the list size being n, return the $\lfloor n/2 \rfloor$-th element in the list. -> SLOW

Time complexity of O(n).

(c) Insert a new element at the beginning of the list. -> FAST

Time complexity of O(1). If currentNode is the head, set currentNode.next.prev as newNode. Set the newNode as head. newNode.prev = NULL; newNode.next = currentNode.next;

(d) Insert a new element at the end of the list. -> FAST

Time complexity of O(1), similar process as above, but reversed for the end.

(e) Delete an element from the beginning of the list. -> FAST

Time complexity of O(1). If currentNode is the head, currentNode.next.prev = NULL; currentNode.next = head;

(f) Delete an element from the end of the list. -> FAST

Time complexity of O(1), similar process as above, reversed.

F. circular singly-linked list, with "next" only, and with "tail" only.

(a) Search for an element. -> SLOW

Time complexity of O(n).

(b) Get the middle element, i.e., with the list size being n, return the $\lfloor n/2 \rfloor$-th element in the list. -> SLOW

Time complexity of O(n).

(c) Insert a new element at the beginning of the list. -> FAST

Time complexity of O(1). -> currentNode = tailNode.next; tailNode.next = newNode; newNode.next = currentNode;

(d) Insert a new element at the end of the list. -> FAST

Time complexity of O(1). New node that is set as the new tail and points to the first node in the list. Also, the old tail node's next is the new node.

(e) Delete an element from the beginning of the list. -> FAST

Time complexity of O(1). currentNode = lastNode.next; lastNode.next = currentNode.next.

(f) Delete an element from the end of the list. -> SLOW

Time complexity of O(n) since there's no "prev." We have to loop through the whole array to point to the last node if we delete the last element.

G. circular doubly-linked list, with “next” and “prev”, and with “tail” only.

(a) Search for an element. -> SLOW

Time complexity of O(n).

(b) Get the middle element, i.e., with the list size being n, return the ⌊n/2⌋-th element in the list. -> SLOW

Time complexity of O(n).

(c) Insert a new element at the beginning of the list. -> FAST

Time complexity of O(1). Similar process as with a circular singly-linked list plus assigning the "prev" as it should be assigned. Space complexity is still O(1) but with one more const.

(d) Insert a new element at the end of the list. -> FAST

Time complexity of O(1). Similar process as with a circular singly-linked list plus assigning the "prev" as it should be assigned.

(e) Delete an element from the beginning of the list. -> FAST

Time complexity of O(1). Similar process as with a circular singly-linked list plus assigning the "prev" as it should be assigned.

(f) Delete an element from the end of the list. -> FAST

Time complexity of O(1). If currentNode is the first node in the list: currentNode.prev = lastNode.prev; lastNode.prev.next = currentNode;

Q2) Description of my algorithm

The time complexity of my algorithm is O(n^2) since it consists of a nested loop. In the best case scenario, it's an O(n) but I could make it O(1) in the best case scenario if I checked to see if the 1 is the first index in the array before I do anything else.

We start off by creating a new stack with the already given stack object within our file. Then inside of a for loop, we check whether the current index starting at 0 contains a 1, if not, then we add the value at the current index to the stack and move on to the next index. Upon arriving at 1, we record the index at which 1 is located at and then break the loop.

After that, we check to see if the stack is empty, if so, we simply return the size of the array this means that 1 is at the first index of the array.

Then, inside of a while loop with a condition of (true), we have a for loop that iterates from the index of 1 up until the end of the array length. This loop checks whether the next value in our sorted list shows up in our array excluding the values added to the stack. If so, we set found to true and increment our result by 1, it's initially 0 and it's what keeps track of how many elements we can output in sorted order. If not, we check to see if the value returned by removing the topmost value of the stack matches the value we are looking for, then increment the result if we pass and break the while loop if we don't pass. Also, before we pop the stack value, we check if found is false and the stack size is greater than 0, if not, then we simply stop the while loop right here and then return the final result.

External resourses: None

Plagiarized: No way >_<