Workshop Week 2

Reminder: Big O notation* Recall from prerequisite subjects that big O notation allows us to easily describe and compare algorithm performance. Algorithms in the class $O(n)$ take time linear in the size of their input. $O(logn)$ algorithms run in time proportional to the logarithm of their input, (increasing by the same amount whenever their input doubles in size). $O(1)$ algorithms run in 'constant time' (a fixed amount of time, independent of their input size). We'll have more to say about big O notation this semester, but these basics will help with today's tutorial exercises. 1. **Arrays** Describe how you could perform the following operations on (i) sorted and (ii) un- sorted arrays, and decide if they are $O(1)$, $O(\log n)$, or $O(n)$, where n is the number of elements initially in the array. Assume that there is no need to change the size of the array to complete each operation. - Inserting a new element - sorted: - find key to insert element by binary search $O(\log n)$ - move all elements after insertion index along by one $O(n)$ - insert element at insertion index $O(1)$ - so overall is $O(n)$ - unsorted: - if we are maintaining list as unsorted, then $O(1)$ as you just insert at the end of the array - Searching for a specified element - sorted: binary search $O(\log n)$ - unsorted: traverse list for item $O(n)$ - Deleting the final element - sorted: $O(1)$ - unsorted: $O(1)$ - Deleting a specified element - sorted: $O(n)$ as you have to delete element $O(1)$ and then move elements along to fill the empty place $O(n)$ - unsorted: same as sorted

2. **Linked lists** Describe how you could perform the following operations on (i) singly-linked and (ii) doubly-linked lists, and decide if they are $O(1), O(\log n), O(n)$ where n is the number of elements initially in the linked list. Assume that the lists need to keep track of their final element.

- Inserting an element at the start of the list
  - singly-linked: $O(1)$
    - create new node pointing to head $O(1)$
    - make head point to new node $O(1)$
  - doubly-linked: $O(1)$
    - create new node pointing to head $O(1)$
    - update head.prev to point to new node $O(1)$
    - update head to point to new node $O(1)$
- Deleting an element from the start of the list
  - singly-linked: store deleteNode = head; set head = head.next; deallocate deleteNode ~ $O(1)$
  - doubly-linked: store deleteNode = head; set head = head.next; store head.prev = NULL; deallocate deleteNode ~ $O(1)$
- Inserting an element at the end of the list

- singly-linked: create new node pointing to NULL; point tail point to new node; update tail ~ $O(1)$
- doubly-linked: create new node pointing to NULL and prev pointint to tail; point tail.next to new node; update tail ~ $O(1)$

- Deleting an element from the end of the list
  - singly-linked: traverse list until you find node.next == tail ~$O(n)$; node.next = NULL; deallocate tail; tail = node;
  - doubly-linked: newTail = tail.prev; newTail.next = NULL; deallocate tail; tail = newTail; ~$O(1)$

3. **Stacks** A stack is a collection where elements are removed in the reverse of the order they were inserted; the first element added is the last to be removed (much like a stack of books or plates). A stack provides two basic operations: `push` (to add a new element) and `pop` (to remove and return the top element). Describe how to implement these operations using

- i. an unsorted array
  - `push`: add element to end of list (assuming you know number of elements) $O(1)$; increase length by 1
  - `pop`: remove element from end of list; decrease length by 1; return element; $O(1)$
- ii. a singly-linked list
  - `push`: create new node pointing to NULL; point tail point to new node; update tail ~ $O(1)$
  - `pop`: traverse list until you find node.next == tail ~$O(n)$; node.next = NULL; deallocate tail; tail = node;

4. **Queues** A standard queue is a collection where elements are removed in the order they were inserted; the first element added is the first to be removed (just like lining up to use an ATM). A standard queue provides two basic operations: `enqueue` (to add an element to the end of the queue) and `dequeue` (to remove the element from the front of the queue. Describe how to implement these operations using

- i. an unsorted array
  - `enqueue`: insert item at end of array $O(1)$; increment length
  - `dequeue`: delete item at start of array; shift items along; decrement length $O(n)$
    - * alternatively if you don't need to delete items you could use a queue index that increments on each dequeue. this would go as $O(1)$
- ii. a singly-linked list

  - enqueue: create new node pointing to NULL; point tail point to new node; update tail ~ $O(1)$
  - dequeue: store deleteNode = head; set head = head.next; deallocate deleteNode ~ $O(1)$

Can we perform these operations in constant time? (see solution for each one)

5. **Bonus problem (optional)** Stacks and queues are examples of abstract data types. Their behaviour is defined independently of their implementation - whether they are built using arrays, linked lists, or something else entirely. If you have access only to stacks and stack operations, can you faithfully implement a queue? How about the other way around? You may assume that your stacks and queues also come with a size operation, which returns the number of elements currently stored.

- implementing queue with a stack
  - enqueue: push
  - dequeue:
    * pop all elements
    * keep last element
    * push all remaining elements in reverse order
- implementing stack with a queue
  - pop: dequeue
  - push:
    * dequeue all elements
    * enqueue new element
    * enqueue elements in same order