QUESTION 2

A.1) Inserting a new element at the head of a linked list.
Notation: O(1)
Steps:
- Create a new node assigning it the value you wish to insert.
- Set the next pointer of the new node to point to the current head of the list.
- Set head to be the new node.

All of these operations are of order 1 or constant time, therefore in the worst case it will take O(1) time.

Finding the memory address: Since inserting a new element at the head of a linked list is done by dynamically allocating more memory each time 'new' is called, we don't have any control over where it goes, but by assigning the new node to the head of our list, we can find its address by looking at the head as this stores the memory address of the new node.

A.2) Finding a given element in a linked list.
Notation: O(n)
Steps:
- Iterate through each element in the list until it matches the given element.
- Returning either a bool, index or key value depending on the specific implementation.

Since in the worst case, the given element may not even exist in the list, we would still have to search every element before terminating the search. Therefore for a list of N elements long the worst case would take O(n) time to complete the operation.

Finding the Memory Address: To find the memory address for a given element, we start from the head of the list and iterate through using the 'next' member variable which points to the memory address of the next node in the list until we find the target value. The memory address of the matching node is the address that was stored in the previous node's 'next' pointer or the head if it's the first node.

B.1) Inserting a new element at the tail of a dynamic array.
Notation: If the array needs resizing then O(n), else O(1), assuming the worst case means that the array is at capacity and so would need resizing, I would say O(n)
Steps:
- Check to see if we have room in the array (Since we are assuming the worst case of having to resize) we would not have space.
- Create/allocate memory for a new array with double the previous arrays size.
- Iterate through each element in the original array, adding each element to the new array.
- Then add the new element to the tail.
- Free the old arrays memory.

In the situation where we do have enough room to add a single element in a dynamic array, perhaps we know what the upper bound is or we just have the room, then this operation is far more simple and equates to O(1) as all we would need to do is insert the element in the next index which would be found by looking up the arrays size which is a stored value in c++. It would be O(n) in C though... but now I'm getting too far off topic.

Finding the Memory Address: When an array is created, the specific amount of memory needed for that array is allocated, for example an array of 10 Integers would be stored as 40 bytes (assuming 32 bit integers) with each element occupying 4 bytes next to one another in memory. If the array starts at memory address 0x1000 then the element indexed at array[1] would be 0x1004 in memory, therefore to find the memory address for an element inserted into the tail of an array would be:
base address + (index - 1) * size of element, where size of element is the size in bytes that a value takes up in memory.

B.2) Finding an element with a given index in an array
Notation: O(1)
Steps:
- It's as simple as looking up the element at the given index in the array to locate it. It either exists or your index is out of bounds in which case you'll get an error, but any index from 0 - N-1 will return you the element, therefore it is only O(1) one constant operation.

Finding the Memory Address: Look above for my explanation on arrays in memory, this is essentially the same except we are looking for a given index so the equation is slightly different:
base address + index * size of element, where size of element is the size in bytes that a value takes up in memory.

B.3) Why don't we insert an element at the head of a dynamic array
Inserting an element at the head of a dynamic array would mean shifting each element in the array down by one which would be an order of N operation, compared to what was just mentioned above, inserting at the tail is an order of 1 or constant operation, therefore it is far less complex and far more efficient to insert at the tail over the head.

C.1) Which data structure would you choose for operations that frequently add and remove data from a dynamic set?

Singly/Doubly Linked lists - Our above analysis shows that linked lists are better for this operation as they are more efficient with O(1) time complexity.

C.2) Which data structure would you choose for operations that do frequent item lookups?

Dynamic arrays - Our above analysis shows that dynamic arrays are best for this operation as they are more efficient with O(1) time complexity.

Bonus answer: Hash tables for key lookups are faster than both linked lists and arrays.