CS 120: Intro to Computer Programming II

In-Class Activity - 03 Linked Lists - Day 1

Activity 1 - Turn in this one

In this ICA, we will replicate the activity that we did in the PlayPosit video.)

In this activity, choose 3 random numbers. Put them into an array, but **in order.** Then, choose 5 other random numbers, **not in order.** Insert them into the array, one at a time; at all times, **keep the array sorted.**

Draw the array after each insertion step. For each insertion step, show how many "shift" operations you had to perform to make it work.

Activity 2 - Turn in this one

In the video, after I had you perform the activity, I discussed the performance of this operation. In this activity, I want you to restate what I said, in your own words. **Make sure to discuss this in your group - there will probably be some students who don't understand this very well, yet. Help them out!**

Answer the following questions in your own words:

- When you inserted a new value into a sorted array, what was the minimum number of values that you had to shift over, and when did that happen?
- How about the maximum?
- In the video, I claimed that **on average** you would shift roughly half of values. Why was this the case? (Be sure to talk about "typical" insertions - not just the min and max.)
- What does $O(n)$ mean? If something takes $O(n)$ time, how does the time grow as the size of the array gets larger?
- Argue that inserting one value, into a sorted array, takes $O(n)$ time.

Bonus question (not required for the ICA, but please discuss it in group):

- Suppose we start with an empty array, and then insert many values, keeping them sorted as we go. Why would that be an $O(n^2)$ operation, to insert them all?

Solution:

- The minimum number of shifts was 0; this happened when the new value to insert was greater than everything in the array.
- The maximum number of shifts was n; this happned when the new value was less than everything in the array.
- All possible values, from 0 to n, are equally likely. If we average them all out, we'll notice that 0 balances n, and 1 balances $(n-1)$, and so on - so everything averages out to the middle, $(n/2)$.
- $O(n)$ means "linear time" - that is, the operation takes an amount of time that grows proportionally to the size of the input.

- Since, on average, we expect insertion to take time equal to half the size of the data (assuming, of course, that the input data is random), the time taken is certainly proportional to the size of the array.
- We are doing $O(n)$ work each time we insert, and we are inserting n times - so the total amount of work is proportional to the **square** of the size.

Activity 3 - Turn in this one

This activity has more questions about the structure of a linked list (sorry!) Please **discuss these questions with your group,** and make sure that everybody agrees and understands - and turn in your answers.

- Why is it important to have a head pointer, when we want to access a linked list? That is, what would be impossible if we didn't have one?
- In a literal, physical sense, does the head of the list contain all of the data of the list. (Kind of like how an array contains all of the elements inside it?) Explain your answer.
- When we want to pass a linked list as a parameter to a function, should we pass many references? Or maybe an array full of references? Or is there some better way to do it?

Solution:

- The `head` pointer gives us the location of the first node in the list; without it, we wouldn't be able to find any of the nodes!
- No, the head node does **not** store all of the values. It only is a reference to the first, and all of the rest are accessed indirectly through that first node.
- In parameters (and variables), we use the head pointer to represent the entire list; we generally do **not** send all of the values as separate references.

Activity 4 - Optional

OPTIONAL. Complete this if you have time, and turn it in. If you don't have time, you may report to your TA that you ran out of time.

I've pointed out that every object (a.k.a "node") in a linked list needs two standard variables. What are they? Explain the purpose of each of these two variables.

Solution: Every node needs both a `val` field - which stores the value for that node - and a `next` pointer - which points to the next node in the list.

Activity 5 - Optional

OPTIONAL. Complete this if you have time, and turn it in. If you don't have time, you may report to your TA that you ran out of time.

Draw a picture of a linked list with 10 elements in it, as a reference diagram. In this picture, represent each node in the list with a box; put the value of that node inside the box, and represent the `next` pointer with an arrow going from that box, to the next.

Make sure to include a head pointer.

Use **random, unordered** values in your list (lists don't have to be sorted, just like arrays don't have to be sorted!).

Make sure that your boxes are not touching each other.

Finally, write down all of the values from the linked list, in the same order they were in the list, separated by commas, like this:

```
10, 13, -7, 2, 0, -16, 1000, 43, 17, 256
```

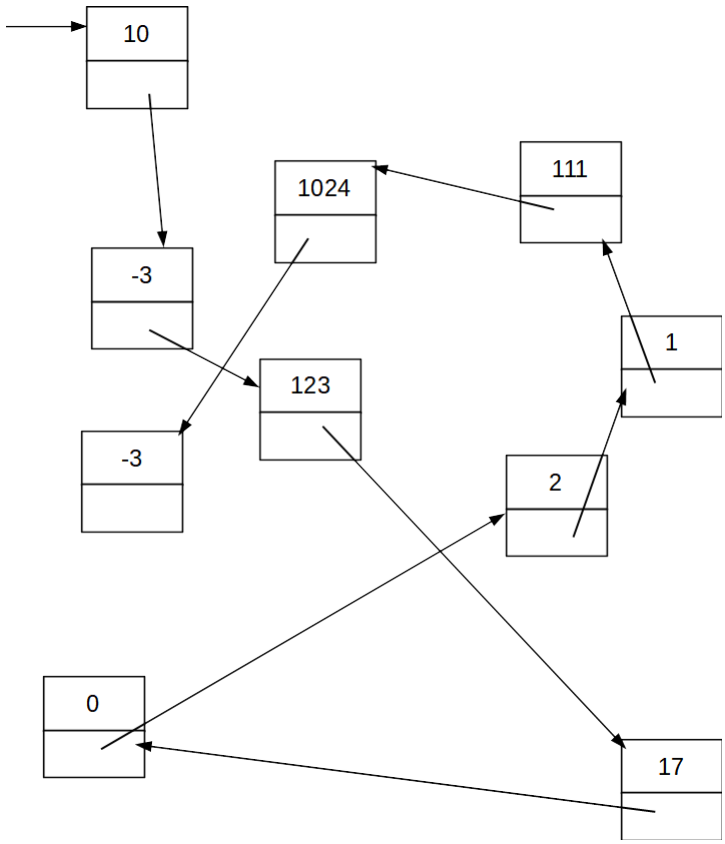
Challenge Activity - Do not turn in this one

We haven't shown you how to write classes in Python yet. But one way to represent a linked list is with arrays.

Draw a picture of a 5-node linked list, using arrays. For each node, use element `[0]` of the array to store the value, and element `[1]` of the array to store the `next` pointer.

Make sure to include a head pointer.

Finally, write a function, which takes as its only parameter, the head pointer to the list we've built. (Assume that it has **exactly** 5 nodes.) Read the values out of that list, and return a newly-created array object (length 5) with those values stored inside it.



Solution: