Algorithms & Data Structures I Week 8 Lecture Note

Notebook:	Algorithms & Data Structures I		
Created:	2020-10-21 4:14 PM	**Updated:**	2020-11-01 8:02 PM
Author:	SUKHJIT MANN		

Cornell Notes	**Topic:** Data Structures and Searching, part 2	Course: BSc Computer Science
		Class: CM1035 Algorithms & Data Structures I [Lecture]
		Date: November 01, 2020

Essential Question:

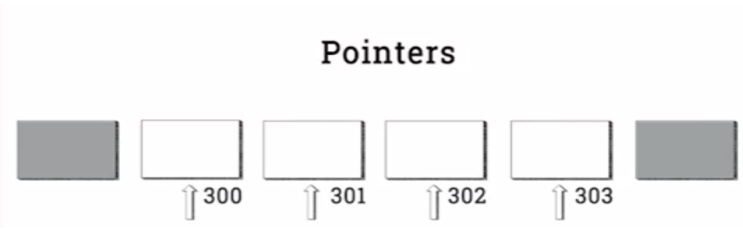
What are arrays, dynamic arrays, linked lists and the linear search algorithm?

Questions/Cues:

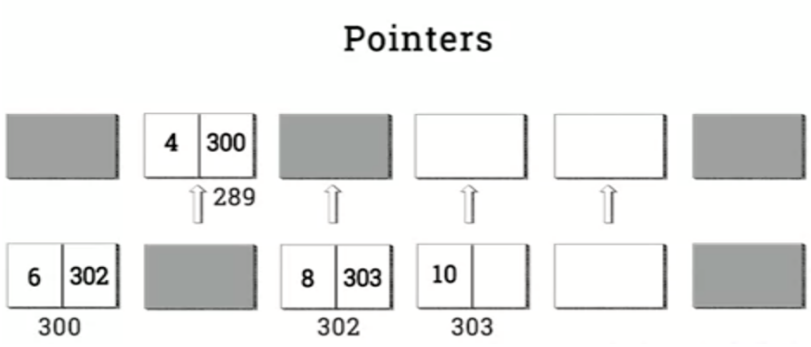
- What is a pointer?
- What is a node?
- What is a linked list?
- How we read and write to and from a linked list?
- How to add or remove nodes from an existing linked list?
- How do we search a linked list?

Notes

- Pointer = Recall that piece of data stored in memory has an address as well as the value being stored. A pointer is used to store the memory address. Think of it as to where we should look in memory; it doesn't store the value at that address, it just points to it
    - To find the value in the memory address to which the pointer is pointing, we need dereference the pointer. This dereferencing of a pointers is the like the act of looking at the words on a page for a particular page number.
    - The point is the like the page number telling us where to read and analogously we dereference the page number to read the words on the page

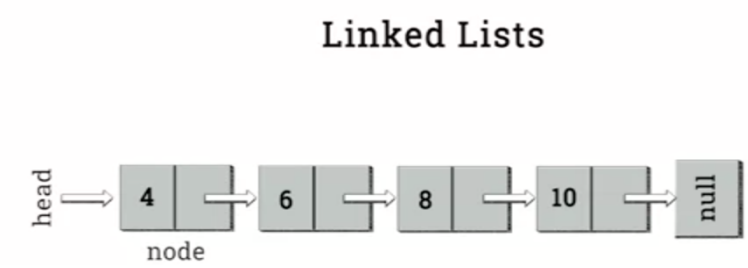


    - An important point to note is that we can have several pieces of data distributed in a computer's memory but we want to collect the data together into something more meaningful
        - Instead of physically moving the data to be sequential or contiguous like an array, we could store not only the data we care about, but we can store a pointer telling us where to look next. To look at the value at which the pointer is pointing to, we just dereference that pointer



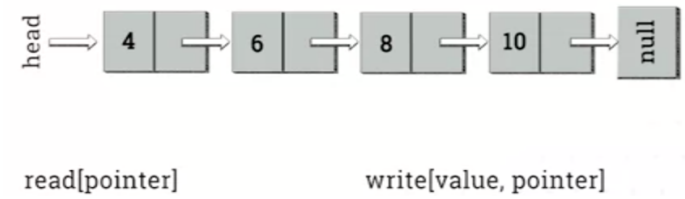
- In the diagram above, we can see several blocks of data where integers and references to locations are stored.
  - The collection of the value and a pointer will be called a **node**, and this is the structure of a linked list, which is just another sequential collection of data

- Linked list = can be broken down into nodes with pointers between them
  - Each node contains a field in which we store a particular value and a link, which is the pointer to the next node
  - The linked list is a concrete data structure and much like the other data structures like a vector or array, a linked list is linear in nature or in a line



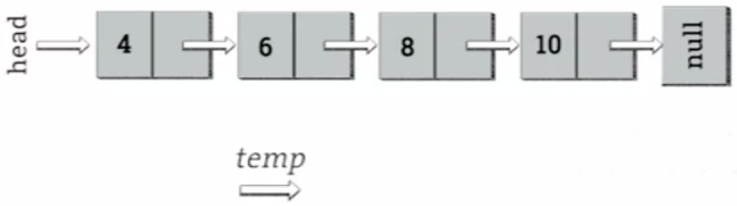
  - The linked list shown above is a singly linked list, where each node only has one pointer to the next node, we call this the next pointer. There is also a pointer to the first node of the list and we reserve a special name for the address associated with that particular pointer, we call it head.
    - So if we have a pointer telling us where to look in our memory and it's set to head, if we dereference this pointer we get the value in the field of the very first node of our linked list
    - Also, the final node has a pointer to another special address in memory; null, which is just a Computer science way of saying nothing. If we have a pointer to null, that means no further parts of the memory are being pointed at and we have the end of the linked list
    - In this way, we can see how we have a sequential collection of data where one node is ordered one after another, with the pointers telling us where to go next
  - Just we can have empty vectors, queue, stacks and etc, we can also have empty linked lists
    - These lists just consist of our head pointer referencing the null address in memory
  - One of the advantages the linked list has to an array is that it's easy to insert or remove nodes at arbitrary positions; we just need to introduce some extra pointers or remove some pointers
    - Recall that with an array, we have the read and write operations, with a linked list we can't just select arbitrary elements, we must follow the pointers until we get to the node that we want. We can thus dereference a pointer to read the value stored at a node.

Linked Lists

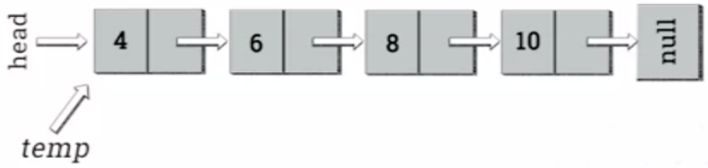


read[pointer] write[value, pointer]

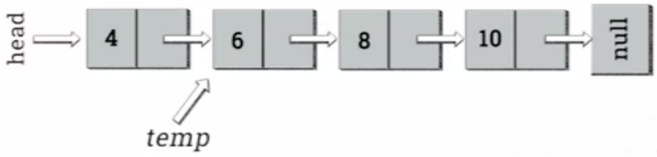
- We can also write things into the fields of a node, if we're just pointed to a particular node, just as we can write to an array element once we have the index of that element
- In an array we kept track of the index of a particular element with an integer, so we knew where we were writing and reading elements. With a linked list, we need to keep track of the pointer references, so that we know where we are manipulating data within the list.
  - To do this, we create a temporary pointer, let call it *temp*, which will store a pointer at any one time, just like an integer can store the index of an array. In this way, we can see that the pointer is itself an abstract data type that can take certain values



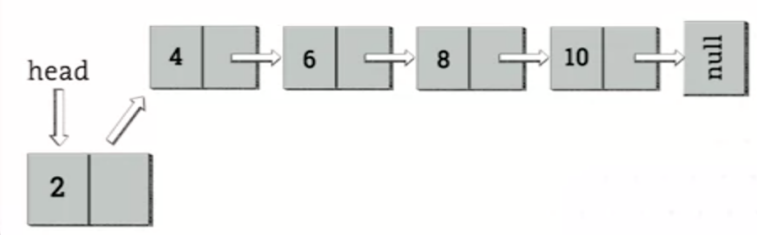
- Once the temporary pointer is created, we can initially assign it to the value head. Temp is now pointing at the first node in the linked list. So we know where we can read values and write new values to the field of the node.



  - A node comes equipped with a data value and a pointer. So we can also read the value of this pointer as well as assigning it to our temporary pointer, *temp*. *Temp* will now be telling us where to look next in the linked list. We cam repeat the process by signing to temp the pointer value in the second node and so on

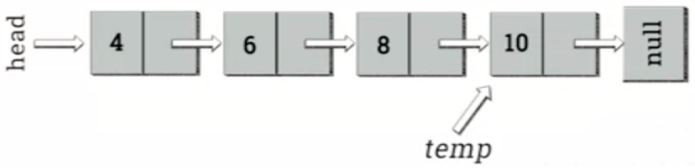


  - In this way, we can traverse a linked list by changing our temporary pointer value, so the computer knows where to look next
- To add new nodes to an existing list, we have different approaches depending on where we want to add the node:
  - If we wish to add a node to the front of the list, we need to first create the node and assign a value to its field and then create a pointer from that node to the head of the old list, then finishing with a new head pointer pointing to the new node

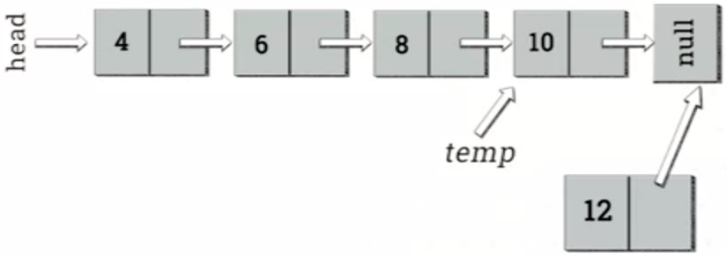


- We can use this method to create new linked lists from the empty list. These four steps are enough to build larger and larger lists.

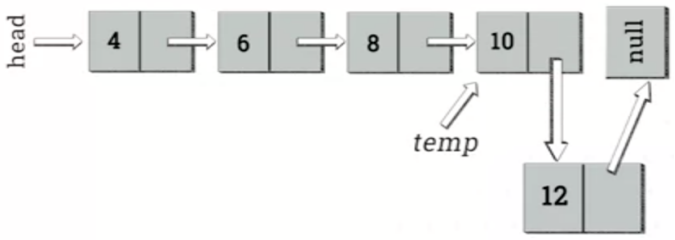
- If we want to put a new node at the end or in the middle of a non-empty list, well first we would need to traverse the list until our temporary pointer, *temp*, is pointing at the correct node. The correct node is the node before the location where we want to put our new node



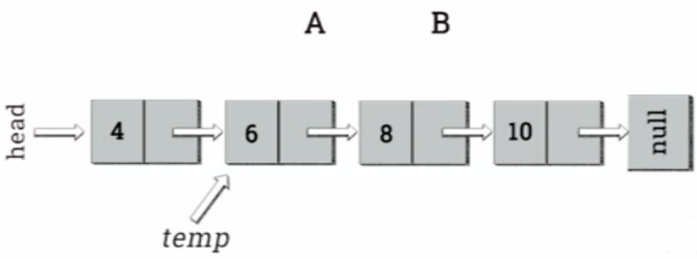
- We can now read where our temp pointer is pointing to see that it's pointing to null
  - From this, let's say you want to put the new node at the end of the list, so this new node would have to have a pointer pointing to null and the last node in the old list will now point to this new node
    - First, we create a new node with its own pointer, put data into its field and then set its next pointer to be the null pointer.



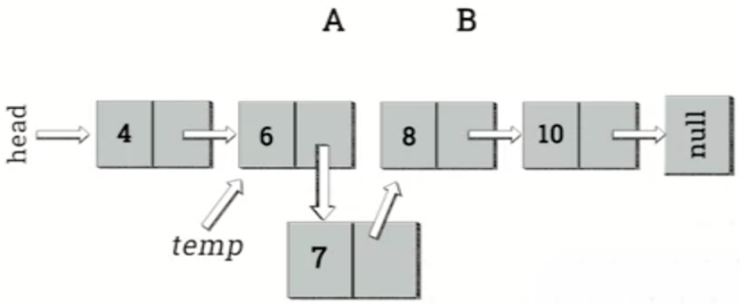
    - Now, we change the pointer of the last node in the old linked list which was previously pointing to null, to now be a next pointer to our new node



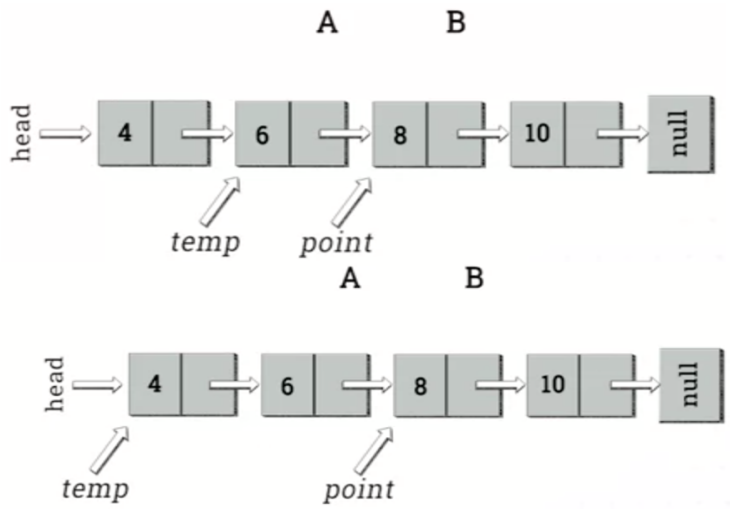
- To add a new node to the middle of a non-empty list,
  - First, we need to check need to check that the previous node to where we want to add a new node isn't pointing to null. This can't happen since it means we're not adding something to the middle but actually to the end



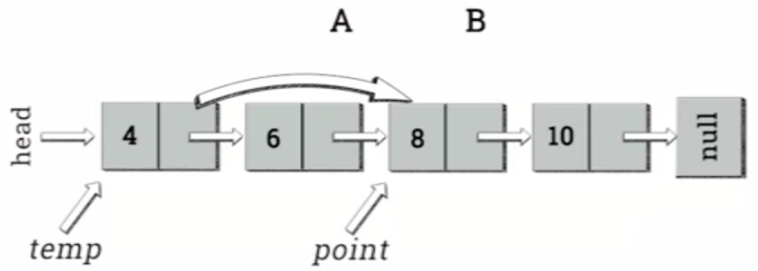
        - Then, we create the node and assign it a value in its field and now we change the next pointer of our new node to be the previous pointer of node B and also the previous pointer of our new node to be the next pointer of node A



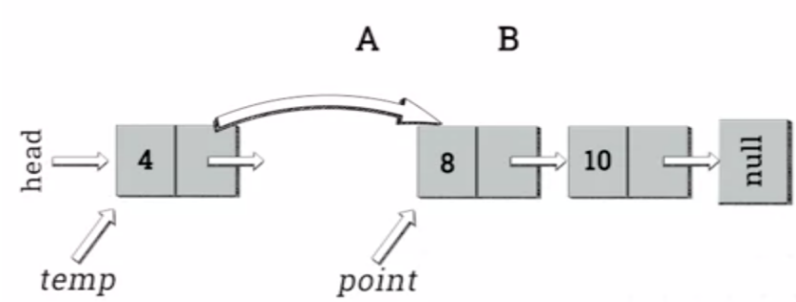
- In summary, in each of these cases of adding a new node in our list, there are only a few steps involved:
    1. Traverse the list
    2. Generate a new node and change a couple of pointers
- If only add nodes to the beginning of the list, we would never need to traverse the list in the first place, leaving us just four operations:
    1. Allocate a new node
    2. Fill its field with data
    3. Create a new next pointer for it
    4. Then move the head pointer from the beginning of the old list to the new node
- To remove a node, the operation needed to delete a node depend on where it is
    - If the node is at the beginning of the list, we can simply change the previous pointer of the first node to be the head pointer, instead of the next pointer of the first node. The first is now orphaned from the rest of the list and we can simply free up the memory that this node is occupying by deleting its values
    - Similarly, if the node to be deleted is at the end of the list, then after traversing the list using a temporary pointer, the previous pointer of the node being deleted will now become a null pointer and we free up the space that this end node is occupying
    - To delete a node in the middle of a list, we first traverse the list to get to the node that is to be deleted. We now want the previous pointer to this node to be merged with the next pointer of this node, (ie. they become the same pointer).
        - From here, first we must store the next pointer of the node to be deleted. We will store this as *point*. Now we go back one node in the list



        - So our temporary pointer is pointing at the previous node to the one being deleted.



- We now reassign the next pointer of this node to have the value *point*. Now we can just free up our memory by deleting the node completely



- In terms of searching, we can't just select nodes at will- we need to traverse the list using the pointers. So, any search algorithm for a linked list would traverse the linked list just as we would traverse an array. In this way, we can modify the linear search algorithm to the case of the linked list, by creating a temporary pointer, *temp*, and just use it to look at each individual node until we find the value we're looking for or not

Summary

In this week, we learned about what a pointer is, what a node is, what a linked list is, how we can read from and write to a linked list, how we can either add or remove node from a existing linked list, how we can search a linked list and so much more.