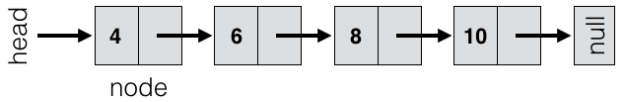
Queues and Stacks with Linked Lists

Algorithms and Data Structure I

In these notes we will go over how we can implement stacks and queues with the linked list data structure. We will quickly review linked lists, in particular how elements can be added and removed from the front node (or first node) of the linked list, as well as adding elements at the end of a linked list. After this, we will show how stacks can be implemented with linked lists, and finally how queues can be implemented.

1 Linked lists

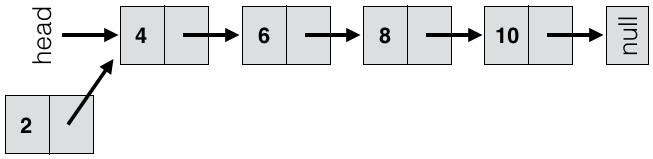
Recall that with linked lists we have nodes, and in each node we have a data field and a field that stores a *next* pointer that points to the next node. In addition to this we have a head pointer and a pointer to null at the beginning and end of the list respectively. Here's the picture we have of a linked list:



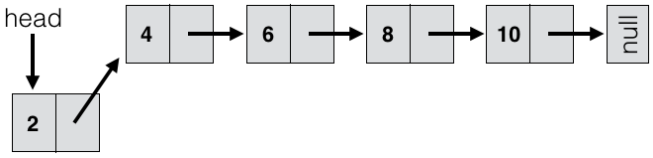
In addition to the list, we are able to create new pointers as variables and have them point to the nodes of linked lists. This will allow us to keep track of certain elements of a linked list, and we can dereference pointers to read (and write) the values stored at where the pointer is pointing. When we come to implement queues, having extra pointers will be very useful.

1.1 Adding and removing elements at the front of a linked list

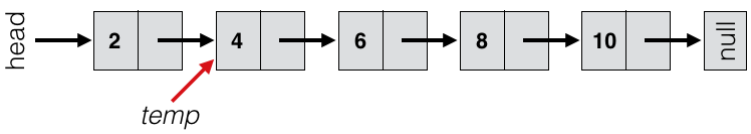
If a node is to be added at the front of the linked list, we first create a node which stores the required value (in the data field) as well as a pointer to the first node of the list, if the list is empty then it will point to null. The situation will look like the example above:



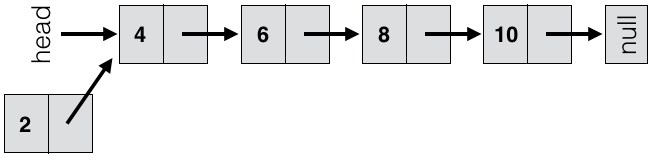
Then after this, the head pointer will be moved to now point at this new node to arrive at the following situation:



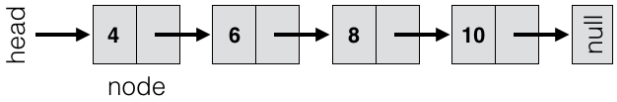
To remove a node from the front of the linked list, we can just, in some sense, reverse this process by first getting the head pointer to point at second node (or null if there is only one element in the list). We can assign the head pointer this value by first creating a new pointer called *temp* and give it the value of the next pointer of the first node. The situation will look like this:



We can then give the head pointer the same value as *temp* to recover this situation:



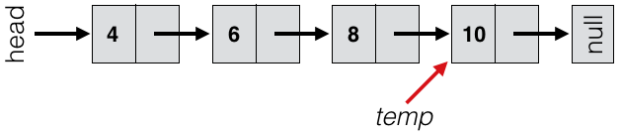
Then it remains to remove what was the first node in the linked list to obtain the following linked list:



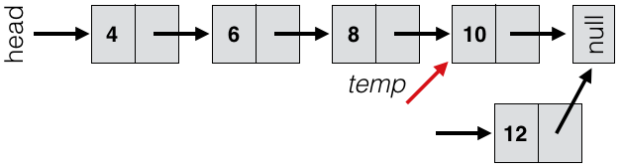
Thus the first node has been removed from the linked list.

1.2 Adding elements at the end of a linked list

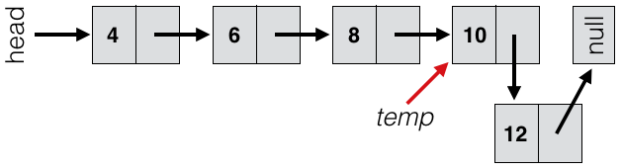
To add an element at the end of a list we need to introduce an extra pointer in addition to the linked list called *temp*. This pointer will traverse the list until it reaches the end of the list. We can check whether we have reached the end by checking the next pointer of each node to see if it points to null, and if it does we do not alter the value of *temp*. Once this pointer has traversed the list to the end we end up with this situation:



To add a new node to the end of the list, we first create a new node with the relevant data in its data field and we keep track of its location with a pointer. Then the next pointer of this new node will point to null to get this:



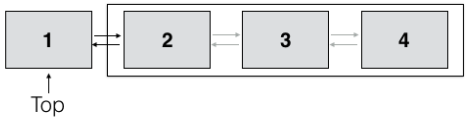
In the next step we take the next pointer of the final node in the original linked list, and give it the same value at the pointer to this new node:

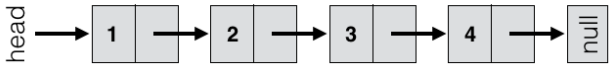


We have now added a new node to the end of the list. If we wanted to keep track of where the end of the list is we would just update *temp* to have the same value as the next pointer of the node at which it is currently pointed. Therefore, instead of having to traverse the list every time we wanted to add a new node at the end, we just update the value of *temp*; this will be useful when implementing queues.

2 Stacks

To implement a stack with a linked list, we just make the elements of the stack be the nodes in a linked list; every element storing a value will correspond to a node storing the same value in its data field. The top of the stack is purely the first node in the linked list. In essence the head pointer in the linked list will point to the top of our stack. The order of elements in a stack will then just be reflected by the order of the nodes in the linked list: if an element is pushed before another in the stack then there will be a pointer from the node corresponding to latter to the node corresponding to the former. So we have the following picture:





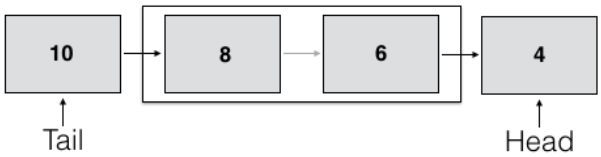
If our stack is empty then the head pointer will point to null, just as the top operation applied to an empty stack will have to return null, or nothing, or something similar. Now, we will show how each of the stack operations can be implemented by the linked list:

- top: The head pointer will be dereferenced and the associated value returned
- empty?: The head pointer will be dereferenced and return TRUE if and only if it points to null, and otherwise return FALSE
- push![o]: A new node storing the value o in its data field will be added to the front of the linked list, and the head pointer will now point to this new node
- pop!: The front node of the linked list will be removed as outlined above, with the head pointer now pointing to what was the second node (or null if there was only one element)

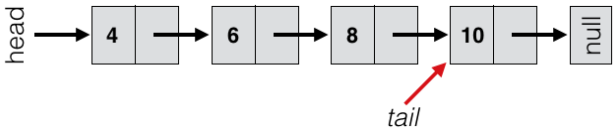
This is why linked lists are an excellent data structure for implementing stacks. The relative ease with which we can add and remove nodes to the front of the linked list without having to assign large amounts of new memory for each operation is a real advantage.

3 Queues

We can see that it is relatively straightforward to implement a stack with a linked list, but what about queues? Well, we can see that the elements in a queue will just correspond to the nodes in the linked list, much as with a stack. However, because a queue has a head and a tail, we need two pointers: the head pointer will point to the node corresponding to the head of the queue, and for the tail of the queue there will be a tail pointer that points to the final node in the linked list. If we are given the following queue:



Then the corresponding linked list will be:



To keep track of the tail of the queue we just keep track of the *tail* pointer. If the queue has just one element then the head and tail pointer will coincide, and for an empty queue, the head and tail pointer will both point to null. Here are the implementations of the queue in full:

- head: The head pointer will be dereferenced and the associated value returned
- empty?: The head pointer will be dereferenced and return TRUE if and only if it points to null, and otherwise return FALSE
- enqueue![o]: A new node storing the value o in its data field will be added to the end of the linked list as outlined above, with the *tail* pointer indicating where the end is. After the node is added, the *tail* pointer will be updated to be the next pointer of the current node, so that it is now pointing to the end node of the linked list

- dequeue!: The front node of the linked list will be removed as outlined above, with the head pointer now pointing to what was the second node (or null if there was only one element)

Again, we can see how linked lists are a natural data structure for implementing queues. The ease with which we can add and remove nodes at the beginning and end of the linked list suits the enqueuing and dequeuing operations of a queue. The only extra technicality is the need for a pointer to keep track of the node corresponding to the tail of the queue.

Exercise: Have a think about how dynamic arrays can be implemented with the linked list data structure. Go through each of the operations of the dynamic array and see if you can describe how it would be implemented with a linked list.