CSCI-2270-Data Structures
Instructor-Hoenigman/Jacobson
Recitation 5

In this recitation, you will be learning the basic manipulations of linked list such as, building a singly linked list and deleting nodes from it.

Linked Lists

As we mentioned last week, one of the limitations of arrays is that they have a fixed size. As studied in the previous recitation, allocating memory to store additional data, once the array is full, could be addressed with an array-doubling algorithm. Even if only one or two additional elements need to be added, array doubling still allocates double the memory. This can be computationally expensive. A list is a data structure that allows for individual elements to be added and removed as needed. In a typical list implementation, called a linked list, memory is allocated for individual elements, and then pointers link those individual elements together.

Singly and Doubly Linked Lists

There are two types of linked lists, singly linked lists and doubly linked lists. In a singly linked list, each element, also called a node, contains the *data stored* in the node and *a pointer* to the next node in the list (shown in Figure 1).

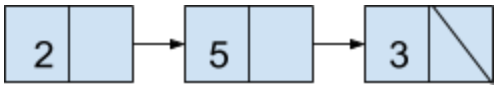

Figure 1. Singly linked list with four elements, called nodes. In this example, each node has an integer key value and a pointer to the next node in the list.

In the Figure 1 example, the node *data is the integer key*. The first node has a key value of 2, the second node has a key value of 5 and the third node has a key value of 3. The next pointer for the final node in the list is set to NULL, which is shown by the slanted line.

In a doubly linked list, each node in the list contains the *node data*, a *pointer to the next* node in the list, and a *pointer to the previous* node in the list. In Figure 2, each node has three properties: an integer key, a pointer to the next node in the list, and a pointer to the previous node in the list. For the first node in the list, the previous pointer is set to NULL, and for the last node in the list, the next pointer is set to NULL. Nodes in a linked list can also be much more complex than these simple examples.

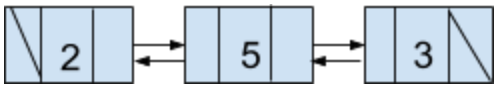

Figure 2. Doubly linked list with three elements.

The node at the beginning of the list is called the head of the list. When implementing a linked list, a separate pointer should be stored to this node as it is the only entrance to the list. The last node in the list is called the tail of the list.

All variables stored in memory have a memory location that can be accessed using a pointer variable. A linked list node can be implemented in C++ using a class or a struct and the next and previous pointers in the node reference another instance of the node.

Let's remind ourselves the node implementation of both singly and doubly linked list

Singly Linked List

```
struct node{
        int key;
        node *next;
};
```

Doubly Linked List

```
struct node{
        int key;
        node *next;
        node *previous;
};
```

Now that we refreshed our memory with the basics of what is a linked list we will dive into the construction of a list and how to search values and delete nodes from it.

Searching for Nodes from a Singly Linked List:

Whenever you want to look for a specific node in the linked list you need to traverse it. This process is similar to traversing an array: you start from the first element in the list and loop over until you find the value you are looking for. How do you know where to start though? Well this is why we set a **head node**. The head node always points to the first element in the linked list and this guarantees us to retrieve the linked list created. Great we have a way to always find our list and start traversing it:

```
tmp = head
while (search_value != tmp -> key)
        tmp = tmp -> next
```

As you can see the search is conceptually identical to the search of an element in an array but for the fact that when dealing with linked lists we always start from the head because that is the only information we know about the list.

Inserting Nodes in a Singly Linked List

Traversing the list is not only fundamental when you have to look for a value but also when you need to insert/add a node to the list. You can add a node in front of the **head**, after the **tail**, or anywhere in between head and tail.

Head

Adding a node at the start of the linked list is the most straightforward insertion we can think of when dealing with linked lists. If you want to add a node in front of the head you need to remember to link your newly added node (set the *next pointer of the new node) to the head of the existing linked list first. Once you are done with it you can safely set the head of the list to point to the node you just added to the left of the old list.

```
node *node_to_add = new node
node_to_add -> next = head
head = node_to_add                    //set head to new head (new added node)
```

Tail

We saw this in our last recitation. First we need to create a new node which will store the value we want to add. The new node will point to NULL because we are going to add it at the end of the linked list. We then need to traverse the linked list to find its last element. We need to do so because, in order to add a node at the tail, the last element of the linked list needs to link to the future new element of the list (the node you are about to add).

```
node *node_to_add = new node
node_to_add -> key = value
node_to_add -> next = NULL

tmp = head
while(tmp->next != NULL)
        tmp = tmp->next

tmp->next = node_to_add               //point old tail to new tail
```

Middle

Adding a node in between the head and tail requires more work. We first need to traverse the list until the position where we want to add our node. We could also use the key value of the previous node to find that position. We then need to make sure the new node points to the node that will follow and that the new node will be linked with the node it will be preceded by.

```
node *node_to_add = new node
node_to_add -> key = value
```

```
tmp = head
index = 0
while(index != position-1)          //position - 1 is the index of the previous node
        tmp = tmp ->next            //find the node that will precede the new node to
add
        index ++
node *tmp1 = tmp ->next
node_to_add->next = tmp1            //point new node to node that will follow
tmp ->next = node_to_add            //point to that precede new node to new node
```

Deleting for Nodes from a Singly Linked List

Head

Deleting the head node requires us to keep track of the second node to make sure that we set the head to point to the new first node.

```
tmp = head
head = head -> next                 //set new head to second node in list
delete tmp
```

Tail

Deleting the tail node is an easy task. We just need to reset the tail to point to the next-to-last node in the list.

```
tmp = head
while(tmp -> next != NULL)
        tmp = tmp -> next           //find second to last node in list

tmp1 =tmp  -> next                  //pointer that points to tail
tmp -> next = NULL                          //make second to last node to be new tail
delete tmp1
```

Middle

Deleting a node in the middle of a linked list requires traversing the list and making sure that we link the remaining nodes. If we find the node we want to delete and we delete it, without linking the remaining nodes together, we create a memory leak. It is important to link the preceding and following node with one another **before** deleting the desired node.

In class work:

We did not write the pseudocode in the previous section because your job for this week's recitation is to write the pseudo code for deleting a node in the middle of the linked list. Once you are done with the pseudo code, use the singly list 3->5->1->10->2 and draw the steps that your pseudo code will go through in order to delete the node at position 4. Show the work to your TA.

Recitation 5 Programming Assignment

There is a link on Moodle to a Recitation 5 Programming exercise. In the quiz, you are asked to write a C++ function to delete nodes in a doubly linked list. You have until Sunday at 5pm to complete the exercise.