BYU IDAHO

CSE 212 | Programming with Data Structures

W07 Prepare: Reading

Table of Contents

Linked Lists

I. LINKED LIST STRUCTURE

When we learned about the dynamic array, we saw that it was characterized by contiguous memory. Each item in a dynamic array is right next to the next item in memory. This allowed for very quick access to items in the dynamic array because memory addresses of each item could be calculated from a formula (address = starting_address + (index * item_size)). The queue, stack, set, and map were all based on memory organized in this way.

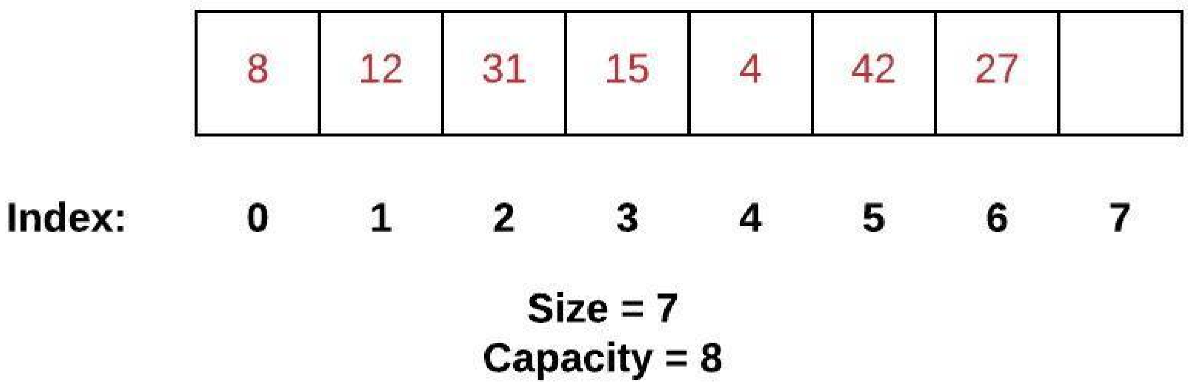

Figure 1 - Dynamic Array

A collection of data can also be stored in a random way within memory. A **linked list** is organized in this way. With a linked list, each element in the list is at some location in memory. There is no guarantee that one element will be next to another element. In order to keep our list together, each element (which we will call a **node**) will contain both the **value** and a link to the **next node** in the list. Specifically, the link will be a **pointer** to the location in memory that contains our next element.

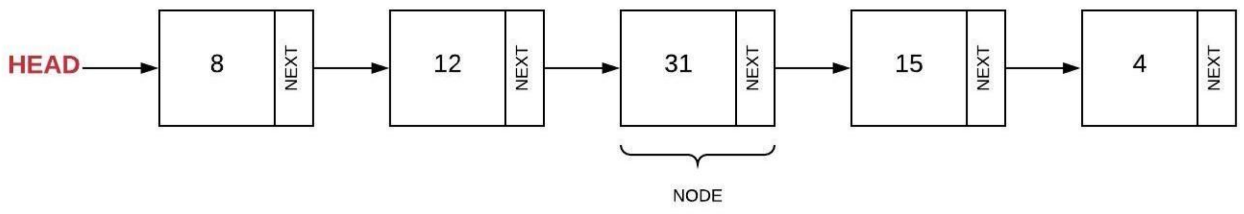

Figure 2 - Linked List

In the linked list shown above, the first node is called the **head**. If you know where the head is, then you can traverse the entire linked list by following the pointers. Most linked lists maintain a bi-directional linking between nodes. Bi-directional means that each node will maintain a pointer to both the next node and **previous**

node. The **doubly-linked list** shown below has both a head and a **tail**.

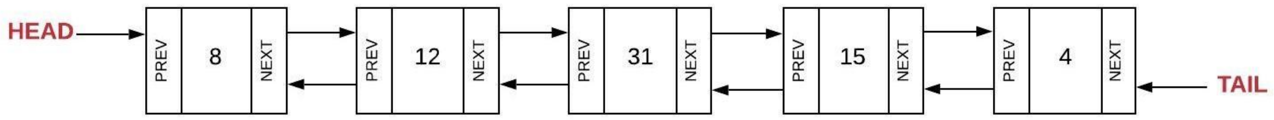

Figure 3 - Doubly-Linked List

Inserting into a Linked List

Unlike inserting into a dynamic array where we had to worry about moving items over towards the end to maintain contiguous memory, the act of inserting into a linked list only has an effect on its neighboring elements. Additionally, since we are going to use pointers to connect the nodes together, there is no need to think about things such as capacity or growing the list like we did with a dynamic array. There are three scenarios to consider: insert at the head, insert at the tail, and insert at the middle.

Inserting at the head usually requires a four step process:

01. Create a new node (we will call it **`new_node`**)
02. Set the "next" of the new node to the current head (**`new_node.next = self.head`**)
03. Set the "prev" of the current head to the new node (**`self.head.prev = new_node`**)
04. Set the head equal to the new node (**`self.head = new_node`**)

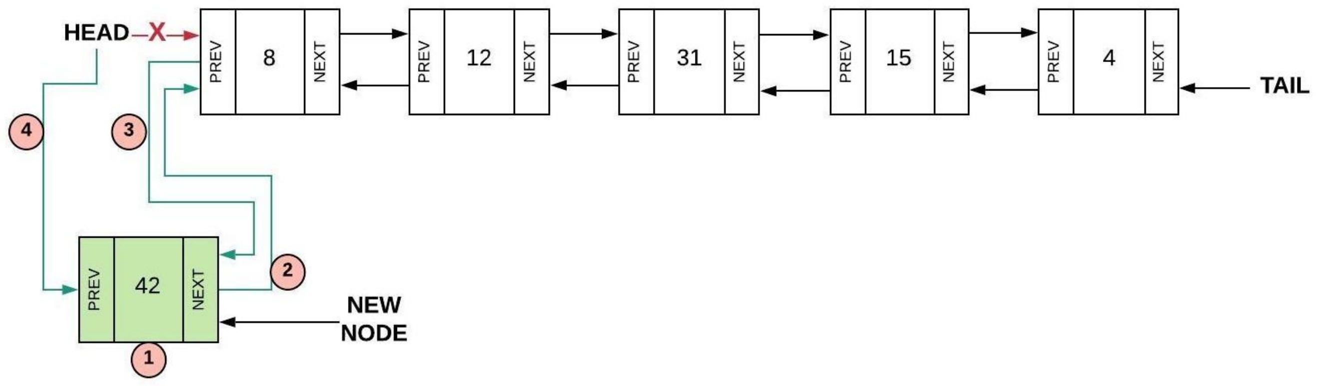

Figure 4 - Inserting at the Head of a Linked List

There is a special case that exists for inserting at the head (and also inserting at the tail). If the linked list is empty (**`self.head is None)`**) then all we have to do is set both the head and the tail to the new node we created.

Inserting at the tail is similar to inserting at the head. The same four steps are followed but with respect to the tail the following should happen:

01. Create a new node (we will call it **`new_node`**)
02. Set the "prev" of the new node to the current tail (**`new_node.prev = self.tail`**)
03. Set the "next" of the current tail to the new node (**`self.tail.next = new_node`**)
04. Set the tail equal to the new node (**`self.tail = new_node`**)

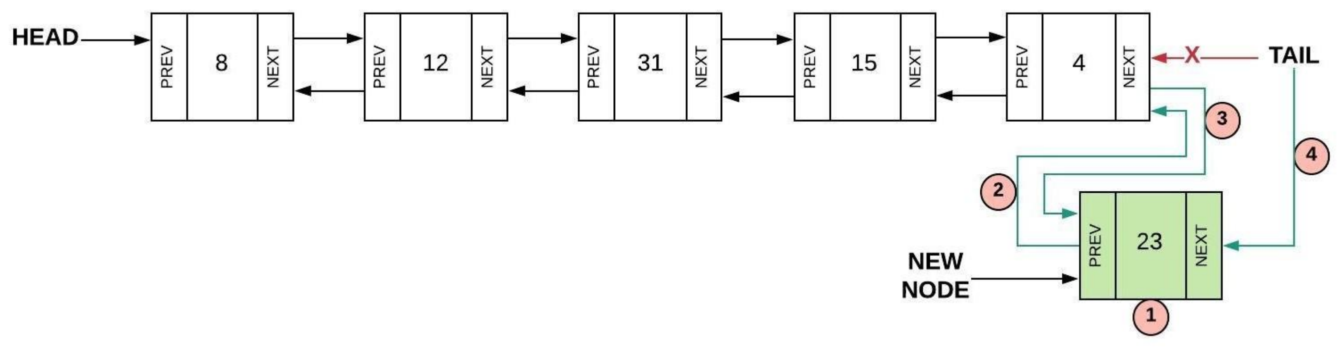

Figure 5 - Inserting at the Tail of a Linked List

The process for inserting into the middle is a little more complicated. In the picture below, we are trying to insert after node **`current`**. The process involves five steps as follows:

01. Create a new node (we will call it **`new_node`**)
02. Set the "prev" of the new node to the current node (**`new_node.prev = current)`**

"

03. Set the "next" of the new node to the next node after current (**`new_node.next = current.next)`**
04. Set the "prev" of the "next" node after current to the new node (**`current.next.prev = new_node)`**
05. Set the next of the current node to the new node (**`current.next = new_node)`**

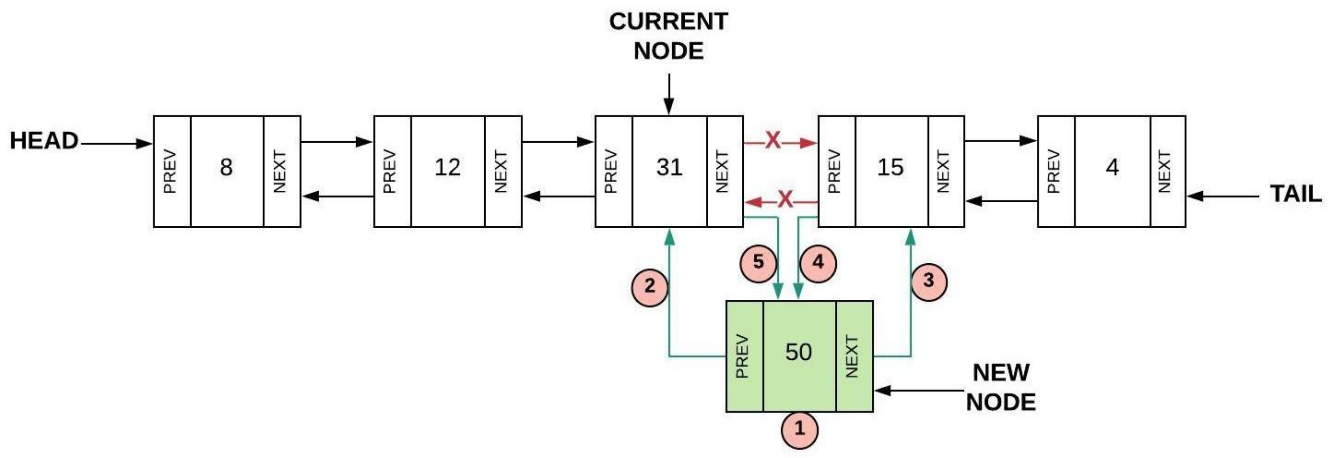

Figure 6 - Inserting in the Middle of a Linked List

Removing from a Linked List

Removing the first (the head) or the last (this tail) node from a linked list is similar and involve setting updating the second node (or the second to last node in the case of removing from the tail). The process for removing the first node is as follows:

01. Set the "prev" of the second node (**`self.head.next`**) to nothing (**`self.head.next.prev = None`**)
02. Set the head to be the second node (**`self.head = self.head.next`**)

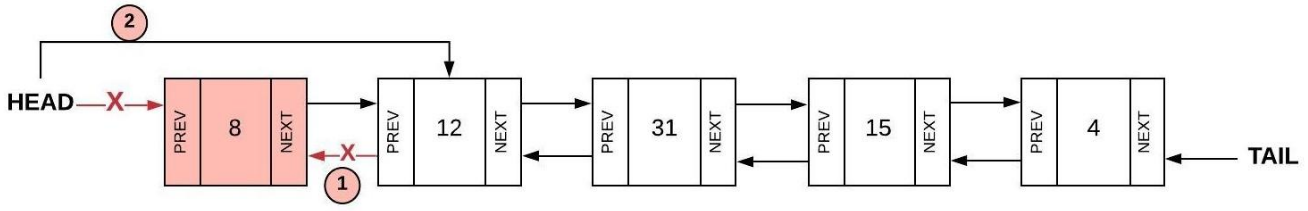

Figure 7 - Remove the Head from the Linked List

As a special case, if there was only one node in the linked list, the head and tail would need to be set to None thus creating an empty linked list.

The process for removing the last node is as follows:

01. Set the "next" of the second to last node (**`self.tail.prev`**) to nothing (**`self.tail.prev.next = None`**)

02. Set the tail to be the second to last node (**`self.tail = self.tail.prev`**)

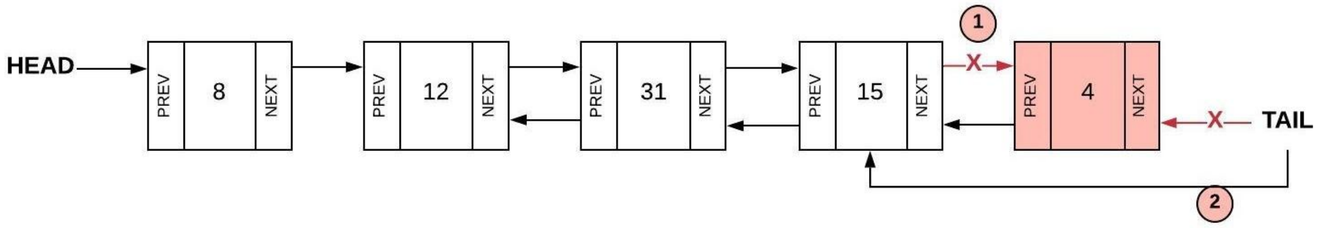

Figure 8 - Remove the Tail from the Linked List

The process to remove from the middle is not as complicated as inserting from the middle. In the picture below, we are trying to remove the node **`current`**. The process involves two steps:

01. Set the prev of the node after current to the node before current (**`current.next.prev = current.prev`**)

02. Set the next of the node before current to the node after current (**`current.prev.next = current.next`**)

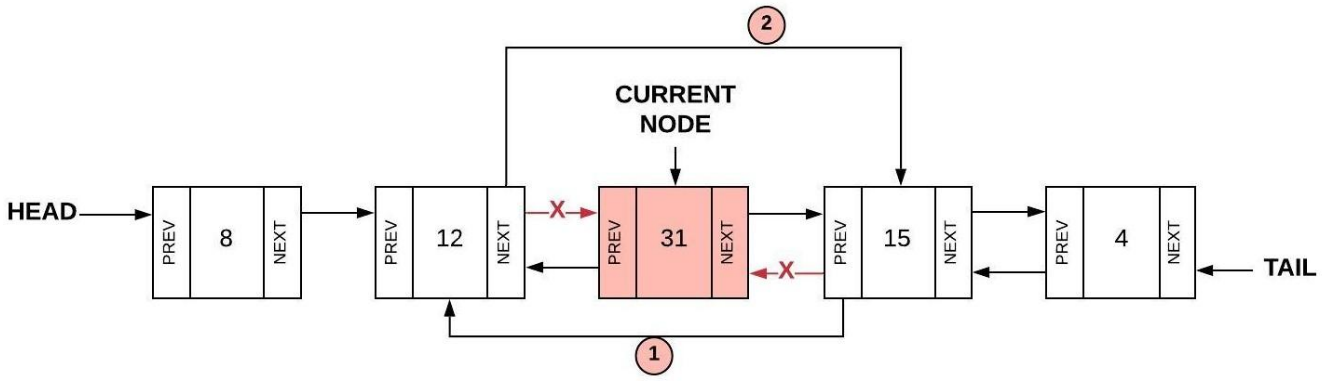

Figure 9 - Remove from the Middle from the Linked List

Accessing from a Linked List

If we want to find a value in the linked list or if we want to find a specific node (e.g. the 3rd node or the 10th node), we are required to loop through the linked list. We can start at either the head (if we want to go forward through the list) or we can start at the tail (if we want to go backward through the list). To loop through the list, we will follow the "next" (or the "prev" if going backwards from tail) links until we get to the end. The following code is a basic traversal through a linked list:

```
def go_forward(self):
        # Start at the beginning (the head)
        current = self.head

        # Loop until we have reached the end (None)
        while current is not None:

                # Do something with the current node
                print(current.data)

                # Follow the pointer to the next node
                current = current.next
```

Linked Lists in Python

In your assignment this week you will be writing your own linked list class. However, Python does have a linked list available for use called the **`deque`**. To create an empty linked list, the following code is used: **`link_list = deque()`**. You will need to include the following import statement as well: **`import deque`**. The table below shows the common functions in the **`deque`**.

Common Linked List Operation	Description	Python Code	Performance
insert_head(value)	Adds "value" before the head	my_deque.appendleft(value)	O(1) - Just need to adjust the pointers near the head.
insert_tail(value)	Adds "value" after the tail	my_deque.append(value)	O(1) - Just need to adjust

			the pointers near the tail.
insert(i, value)	Adds "value" after node "i".	my_deque.insert(i, value)	O(n) - It's not complicated to adjust the pointers to insert, but it takes a loop to find the node to insert after.
remove_head()	Removes the first item (the head)	value = my_deque.popleft()	O(1) - Just need to adjust the pointers near the head.
remove_tail(index)	Removes the last item (the tail)	value = my_deque.pop()	O(1) - Just need to adjust the pointers near the head.
remove(i)	Removes node "i".	del my_deque[i]	O(n) - It's not complicated to adjust the pointers to remove, but it takes a loop to find the node to remove.
size()	Return the size of the linked list	length = len(my_deque)	O(1) - The size is maintained within the linked list class.
empty()	Returns true if the length of the linked list is zero.	if len(my_deque) == 0:	O(1) - The comparison of the length with 0 is all that is needed.

Comparing Dynamic Array and Linked List

The dynamic array and linked list look the same to the user but because the memory is managed differently, the performance numbers are different. The table below compares these two data structures:

Operation	Dynamic Array	Linked List
Insert Front	O(n)	O(1)
Insert Middle	O(n)	O(n)
Insert End	O(1)	O(1)
Remove Front	O(n)	O(1)
Remove Middle	O(n)	O(n)
Remove End	O(1)	O(1)

We can conclude the following:

- The dynamic array has good performance at the end.
- The linked list has good performance at the beginning and the end.

Thinking back about stacks and queues, the stack did the push and pop operations from the end. Therefore, a stack can use either a dynamic array or linked list (a Python `list` or a Python `deque`) equally well. However, since a queue did the enqueue and dequeue operations from the end and the front, a queue should always be implemented with a linked list (a Python `deque`). This strong statement only applies when the size of the data is "large" (recall discussions earlier about big O). If we only have a small number of items in my queue, then both a dynamic array and linked list will run fast enough.

Key Terms

» **doubly-linked list** - A linked list that is bi-directional. The linked list will maintain both a head and a tail. To traverse in either direction, the node will have both a pointer to the next node and the previous node. Access to both the head and tail is O(1).

» **head** - A pointer to the first node in the linked list.

» **linked list** - A data structure that keeps data in order but is not in contiguous memory. To get to the next (or previous) item in the list, pointers are maintained and followed. Access to the head is O(1).

» **next** - A pointer in each node of the linked list that points to the next node.

» **node** - The combination of the value and the pointers representing one item in the linked list.

» **pointer** - Refers to an address in the computer memory. Also called a reference.

» **tail** - A pointer to the last node in the linked list. If the list has only one item, then the head and tail are the same.

» **value** - The actual data stored in the linked list as part of the node.

» **previous** - A pointer in each of the linked list nodes that points to the previous node.