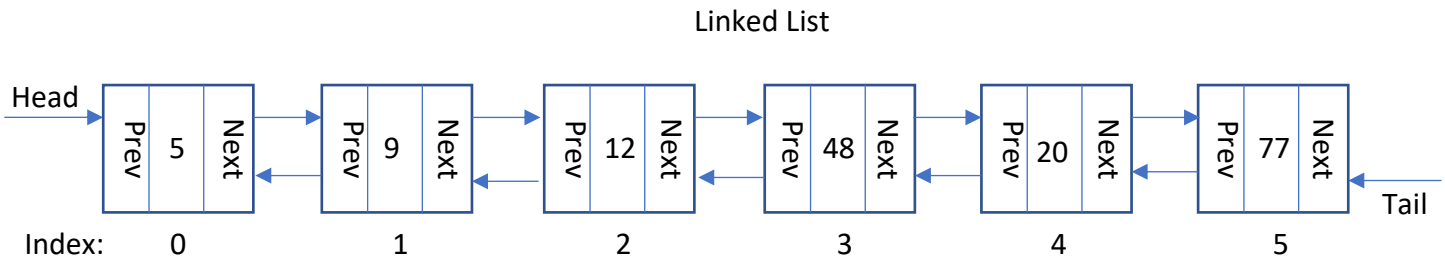
Linked List Tutorial

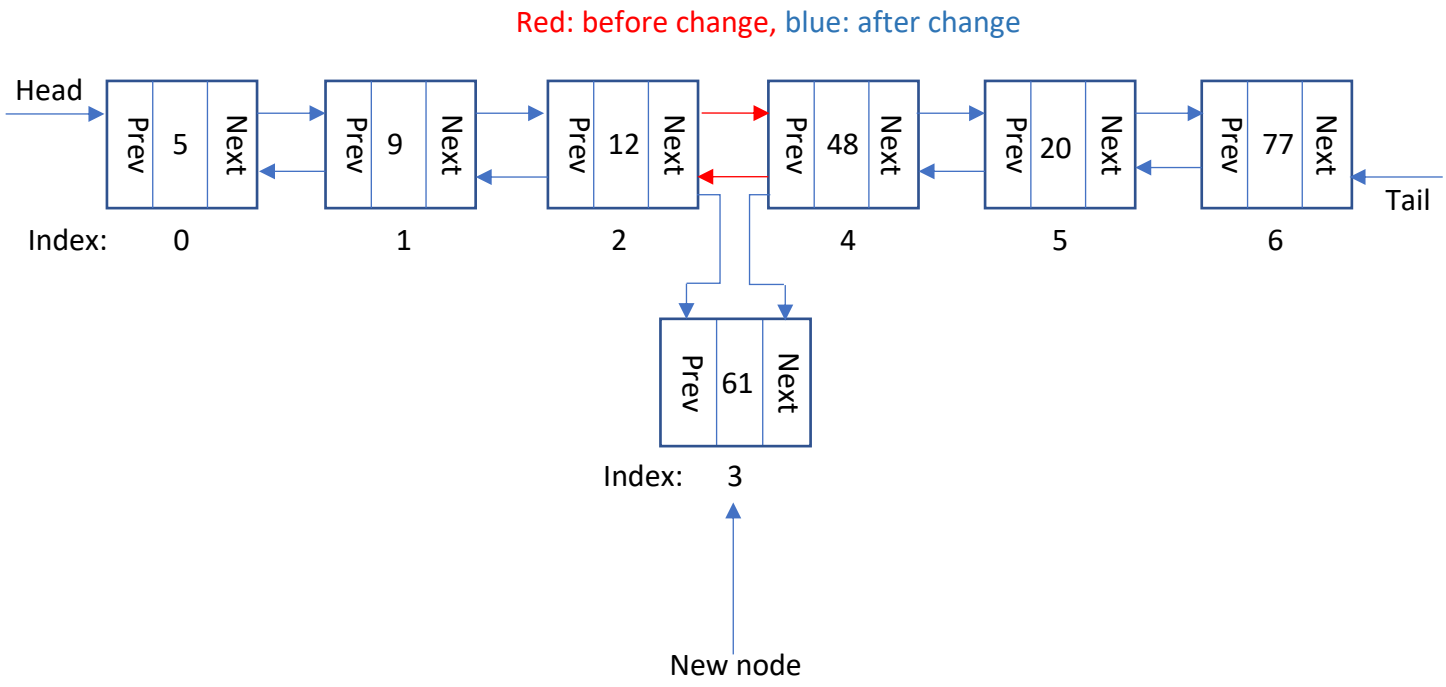
- Introduction
  - Another data structure comparable to an array or list is a **Linked List.** From an outside perspective, a linked list operates the same as an array, but the way it works is somewhat different than how an array works. When memory is allocated for an array, the memory locations are consecutive to one another and thus, consecutive array memory locations can be found through a simple calculation (address = start_address + (index * item_size)). On the other hand, a linked list is also a collection of locations in memory which stores information, except the memory locations are not necessarily consecutive, and thus cannot be deduced mathematically.
  - Each item in a linked list is called a **node**. A node has 3 parts: a value, a handle to the next item's location in memory, and a handle to the previous item's location in memory. In this way, each item is linked to 2 other items although dispersed randomly throughout memory. The first node of a linked list is called the **head,** and the last is called the **tail.**



- Inserting data into a linked list
  - When inserting a new node into a linked list, if the order of the item in the list doesn't matter then the easiest way to do this is to make the new node the new head of the linked list. However, if the linked list has, say, 6 items, like the one above, and the new node needs to be inserted into the slot with index = 2, then the process becomes a little more complicated.
  - Inserting a node at the head is a two-step process.
    - Step 1: Change the current's head's "prev" handle to the new node.
    - Step 2: Change the "head" handle to be the new head's handle.
  - Inserting a node at a particular location in the linked list is like inserting it at the head, but it requires a couple extra steps. This is because we must edit the "next" and "prev" handles of its adjacent nodes to match the new order.
    - Step 1: Change the previous node's "next" handle to the new node

- Step 2: Set the new node's "prev" handle to the previous node mentioned in step 1
- Step 3: Set the new node's "next" handle to the next node
- Step 4: Change the next node's "next" handle to the new node

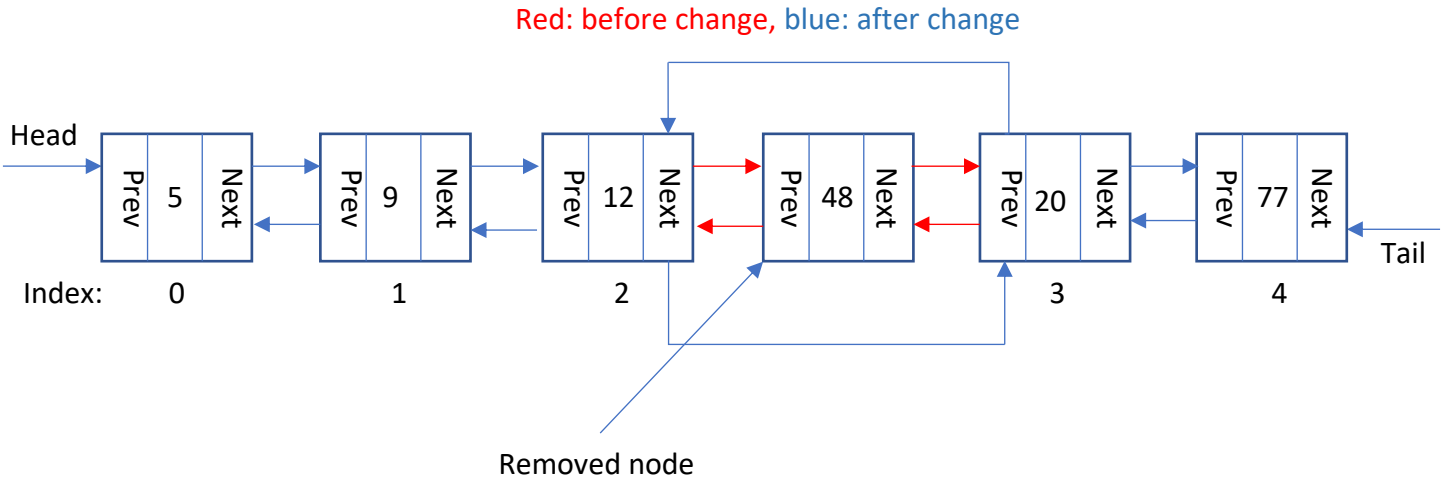
- Inserting a node at the tail is like inserting it at the head.
  - Step 1: Change the current tail's "next" handle to the new node.
  - Step 2: Change the handle to the tail to the new node.



- Removing data from a linked list
  - Removing data from a linked list is like the process of inserting data, except in reverse.
  - Removing a node from the head is a two-step process.
    - Step 1: Change the handle for the head to the node next to the current head
    - Step 2: Change the "prev" handle for the new head to none, as there is not a node which precedes the head
  - Removing a node from a specific location in the linked list is a four-step process.
    - Step 1: Change the "next" handle of the node before the node which will be removed to point to the node next to the node which will be removed.

In other words, if the node to be removed is in index 4, change the “next” handle of the node in index 3 to point to index 5.

- Step 2: Change the “prev” handle of the node next to the node which will be removed to point to the node before the node which will be removed. In other words, if the node to be removed is in index 4, change the “prev” handle of the node in index 5 to point to index 3.



- Accessing data from a linked list
  - Accessing data from a linked list requires a loop starting at the head or tail. Since nodes’ memory location is only known to its neighboring nodes, we gain instant access using index notation (i.e. linkedList[5]), and thus we must iterate through the linked list to arrive at data item we want. If item #5 in the list is wanted, we start with the head and move forward 4 times. If we want to find a particular value, we must iterate through the list and compare against list values until we find it.
- Example with Python Code
- Problem to solve