Chapter 3 – Linked List:

Introduction:

When studying the array, we were quite annoyed by the limitations of these arrays:

- **Fixed capacity:** Once created, an array cannot be resized. The only way to "resize" is to create a larger new array, copy the elements from the original array into the new one, and then change the reference to the new one.
- **Shifting elements in insert:** Since we do not allow gaps in the array, inserting a new element requires that we shift all the subsequent elements right to create a hole where the new element is placed. In the worst case, we "disturb" every slot in the array when we insert it at the beginning of the array!
- **Shifting elements in removal:** Removing may also require shifting to plug the hole left by the removed element. If the ordering of the elements does not matter, we can avoid shifting by replacing the removed element with the element in the last slot (and then treating the last slot as empty).

The answer to these problems is the **Linked List** data structure. It is another primary data structure which is a sequence of nodes connected by links. The links allow the insertion of new nodes anywhere in the list or to remove of an existing node from the list without having to disturb the rest of the list (the only nodes affected are the ones adjacent to the node being inserted or removed). Since we can extend the list one node at a time, we can also resize a list until we run out of resources. However, we'll find out soon enough what we lose in the process — random access, and space! A linked list is a sequence container that supports sequential access only and requires additional space for at least n references for the links.

We maintain a linked list by referring to the first node in the list, conventionally called the **head** reference. All the other nodes can then be reached by traversing the list, starting from the head. An empty list is represented by setting the head reference to None. Given a head reference to a list, how do you count the number of nodes in the list? You have to iterate over the nodes, starting from the head, and count until you reach the end.

As mentioned earlier, a linked list is a sequence of nodes. To put anything (primitive type or object reference) in the list, we must first put this "thing" in a node, and then put the node in the list. Compare this with an array – we simply put the "thing" (our primitive type or object reference) directly into the array slot, without the need for any extra scaffolding. Now to put the “thing” in our list and to store the next item’s information, we need to design our own data type by designing a **Node** class. Our Node class is quite simple — it contains an element, and a reference to the next node in the list. Below is a code snippet of a node class given.

```
# Node class design
class Node:
  def __init__(self, e, n):
    self.elem = e
    self.next = n

```

This allows you a create a singly-linked list, and as a result, we can only move forward in the list by following the next links. To be able to move backward as well, we'll have to add another reference to the previous node, increasing the space usage even more. If you are wondering how can we do that, give it a thought (We will cover it later).

Operations of Linked List:

To understand the operations, first, we will need to understand how to create a linked list. After creation, we can perform different operations on it like an array (e.g: iteration, insertion, removal, and so on).

Creation:

In order to create a linked list, we just need to create objects of the Node class and then connect one node to another using that next variable. We will only store the first Node-type object and call it head. Whenever we want to add a new node-type object, we just need to go to the last node and add the new node after that.

In the example code below, we are taking an array and converting it to a Linked List. To keep things easier, we are using the tail variable so that we do not need to go to the end of the list every time we want to add a new item.

```
# Creating a list
def createList(a):
  head = Node(a[0], None)
  tail = head
  for i in range(1, len(a)):
    n = Node(a[i], None)
    tail.next = n
    tail = tail.next
  return head
```

Iteration:

Iteration is one of the most important aspects of every data structure. We will use an iterative approach to traverse a linked list. The important thing is that to do this traversal we are using a variable (temp in this case) to store the location of node and so we are updating this temp for every iteration. Below is an example of an iteration function that takes the head of a linked list as a parameter.

```
# Iteration over a linked list
def iteration(head):
  temp = head
  while temp != None:
    print(temp.element)
    temp = temp.next
```

Count:

In order to count the number of nodes present in a linked list, we just need to count how many times the iteration process is running during a traversal.

```
# Counting number of element in the list
def count(head):
  count = 0
  temp = head
  while temp != None:
    count += 1
    temp = temp.next
  return count
```

Retrieving an element from an index:

In the memory level, a linked list is not indexed as it does not book any memory block. However, the concept of indexing is very beneficial in order to access values, insert a value, and remove a value (we have seen it in the array part). For this reason, we will implement indexing in our linked list where the head will be a value at index 0 and the next node will be 1, and so on (just like an array). Below an example code is shown where the head of the elemAt function is taking a head of the list and an index (idx). The function is returning the corresponding element at that index. If such an index is not found the function will print an error message.

```python
# Getting element of an specific index
def elemAt(head, idx):
  count = 0
  temp = head
  obj = None
  while temp != None:
    if count == idx:
      obj = temp.element
      break
    temp = temp.next
    count += 1
  if obj == None:
    print("Invalid index")
  return obj
```

Retrieving a node from an index:

Similar to getting the element, we can also get the node of that index. Below the example, nodeAt function is given which is similar to the elemAt.

```python
# Getting node of an specific index
def nodeAt(head, idx):
  count = 0
  temp = head
  obj = None
  while temp != None:
    if count == idx:
      obj = temp
      break
    temp = temp.next
    count += 1
  if obj == None:
    print("Invalid index")
  return obj
```

Update value of specific index:

By tuning the elemAt function, we can also update the element of a linked list using an index.

```
# Setting new element of an specific index
def set(head, idx, elem):
  count = 0
  temp = head
  isUpdated = False
  while temp != None:
    if count == idx:
      temp.elem = elem
      isUpdated = True
      break
    temp = temp.next
    count += 1
  if isUpdated:
    print("Value successfully updated!!!!")
  else:
    print("Invalid index")
```

Searching an element in the list:

Searching for an element in a list can be done by sequentially searching through the list. There are two typical variants: return the index of the given element (indexOf), or return true if the element exists

```
# Getting index of an specific element
def indexOf(head, elem):
  temp = head
  count = 0
  while temp != None:
    if elem == temp.elem:
      return count
    count += 1
    temp = temp.next
  return -1 # Here -1 represents the absence of element in the list
```

Second approach:

```
def contains(head, elem):
  temp = head
  while temp != None:
    if elem == temp.elem:
      return True
    temp = temp.next
  return False
```

Inserting an element into a list:

There are three places in a list where we can insert a new element: in the beginning ("head" changes), in the middle, and at the end. To insert a new node in the list, you need the reference to the predecessor to link in the new node. There is one "special" case however — inserting a new node at the beginning of the list because the head node does not have a predecessor. Inserting, in the beginning, has an important side effect — it changes the head reference! Inserting in the middle or at the end is the same — we first find the predecessor node and link in the new node with the given element. To find the specific node, we can take the help of the nodeAt function mentioned above. Below is the example code of insertion. However, this code can be made more efficient!!! Try that.

```
def insert(head, elem, idx):
  total_nodes = count(head)
  if idx == 0: # Inserting at the beginning
    n = Node(elem, head)
    head = n
  elif idx >= 1 and idx < total_nodes: # Inserting at the middle
    n = Node(elem, head)
    n1 = nodeAt(head, idx - 1)
    n2 = nodeAt(head, idx)
    n.next = n2
    n1.next = n
  elif idx == total_nodes: # Inserting at the end
    n = Node(elem, None)
    n1 = nodeAt(head, total_nodes - 1)
    n1.next = n
  else:
    print("Invalid Index")
  return head
```

Simulation Example: Example-Add-Remove.pdf

Removing an element from a list:

Removing an element from the list is done by removing the node that contains the element. If we only know the element that we want to remove, then we have to sequentially search the list to find the node which contains the element (if it exists in the list that is); or else, we may already have a reference to the node which contains the element to remove; or, we have an index of the element in the list. Just like inserting a new node in a list, removing requires that you have the reference to the predecessor node. And just like in insertion, removing the 1st node in the list is a "special" case — it does not have a predecessor, and removing it has the side-effect of changing the head. You can try to make this more efficient too!!!

```
def remove(head, idx):
  if idx == 0: # Removing first element
    head = head.next
  elif idx >= 1 and idx < count(head): # Removing middle element
    n1 = nodeAt(head, idx - 1)
    removed_node = n1.next
    n1.next = removed_node.next
  else:
    print("Invalid Index")
  return head
```

Copying a list:

Copying the elements of a source list to a destination list is simply a matter of iterating over the elements of the source list, and inserting these elements at the end of the destination list.

```
def copyList(source):
  copy_head = None
  copy_tail = None
  temp = source
  while temp != None:
    n = Node(temp.elem, None)
    if copy_head == None:
      copy_head = n
      copy_tail = copy_head
    else:
      copy_tail.next = n
      copy_tail = copy_tail.next
    temp = temp.next
  return copy_head
```

Reversing a list:

Since a linked list does not support random access, it is difficult to reverse a list in place without changing the head reference. Instead, we'll create a new list with its own head reference, and copy the elements in the reverse order. This method does not modify the original list, so we can call it an out-of-place method.

```
1 def reverse_out_of_place(head):
2   new_head = Node(head.elem, None)
3   temp = head.next
4   while temp != None:
5     n = Node(temp.elem, new_head)
6     new_head = n
7     temp = temp.next
8   return new_head
```

The problem with this approach is that it creates a copy of the whole list, just in reverse order. We would like an in-place approach instead — re-order the links instead of copying the nodes! That would of course change the original list and would have a new head reference (which is the reference to the tail node in the original list).

```
1 def reverse_in_place(head):
2   new_head = None
3   temp = head
4   while temp != None:
5     n = temp.next
6     temp.next = new_head
7     new_head = temp
8     temp = n
9   return new_head
```

Rotating a list left:

Rotating a list left is much simpler than rotating an array — the 2nd node becomes the new head, and the 1st becomes the new tail node. We don't have to actually move the elements, it's just a matter of rearranging a few links

```
def rotate_left(head):
    new_head = head.next
    temp = new_head
    while temp.next != None:
        temp = temp.next
    temp.next = head
    head.next = None
    head = new_head
    return head
```

Rotating a list right:

Rotating a list right is almost the same as rotating left — the current last node becomes the new head and the current second last node becomes the last node.

```
def rotate_right(head):
    last_node = head.next
    second_last_node = head
    while last_node.next != None:
        last_node = last_node.next
        second_last_node = second_last_node.next
    last_node.next = head
    second_last_node.next = None
    head = last_node
    return head
```

Simulation Example: **Example-Part-1-Rotation.pdf**