Linked Lists

Chapter No.	15
Status	Completed

- Contiguous Lists
  - A contiguous list refers to a list of entries that is stored in memory cells that are in sequence with consecutive addresses
  - Limitations of Contiguous Lists
    - We need to determine how much space to allocate to each entry, as well as the number of entries, to determine how much memory to allocate to the entire list
    - If the list is a static list that is not going to change, then a contiguous list is a suitable data structure
    - However, if the list is a dynamic list which changes frequently, such as when entries are added, deleted or rearranged, this would cause a lot of shuffling and reallocation of the computer's memory
    - In the worst-case scenario, we might add so many entries that the entire list might need to be shifted to a new location in the computer's memory to ensure that there are sufficient contiguous memory cells to store all the entries

- Linked Lists
  - A linked list is a linear data structure which stores data in nodes which do not need to be located in contiguous memory cells
  - Each node in a linked list can be stored in a different area of the computer's memory, instead of one large, contiguous block of memory
    - To do this, each node, consists of the data and a pointer to indicate the location of the next node in the linked list
  - The start pointer/head pointer indicates the location of the first node in the linked list
  - The last node in the linked list has a null pointer which does not point to any other node
  - Benefits of Linked Lists
    - When entries in the list need to be added, deleted or reordered, only the pointers need to be changed, which makes such operations much more efficient, especially when the data stored in each entry is large
    - Being a dynamic data structure, linked lists only use as much memory as they require as additional memory is allocated to the linked list only when it is required

▼ Implementing Linked Lists in Python

```
class Node():

  def __init__(self, data):
    self.data = data
    self.next = None

class LinkedList():

  def __init__(self):
    self.head = None

  def display(self): #prints all the data stored in a linked list in order
    temp = self.head
    while temp: #example of iterating through a linked list
      print(temp.data)
      temp = temp.next

  def push(self, data): #adds a node to the beginning of the linked list
    new_node = Node(data)
    new_node.next = self.head
    self.head = new_node

  def append(self, data) #adds a node to the end of the linked list
    new_node = Node(data)

    if self.head is None: #if linked list is empty
      self.head = new_node

    else:
      temp = self.head

      while temp.next: #example of iterating through a linked list
        temp = temp.next

      temp.next = new_node

  def insert_after(self, prev_node, data): #inserts a node into the linked list after prev_node
      if prev_node is None: #if prev_node does not exist
        print("The given previous node must be in the linked list!")

      else:
        new_node = Node(data)
        new_node.next = prev_node.next
        prev_node.next = new_node

  def delete_Node(self, to_delete): #deletes the node containing the data to_delete
      temp = self.head

      if temp is None: #if the linked list is empty
        return None

      if temp.data == to_delete: #if first node is to be deleted
        self.head = temp.next
        return None

      else:
        while temp: #example of iterating through a linked list
          if temp.data == to_delete: #checks if the current node contains the data to_delete
            break

          prev = temp
          temp = temp.next

        prev.next = temp.next
```

- A linked list can also be represented using an array (table)
- A free list/free space list is an array or linked list that contains a list of free nodes
    - This is makes it easy to find unused nodes which can be added to a linked list