Namal University, Mianwali
Department of Electrical Engineering
Course Title: EE-253 Data Structures and Algorithms

LAB MANUAL

Lab 5 Lists in Python

In previous lab, we have studied the concept of arrays in python. We have studied types of arrays (referential array, compact array, dynamic array, etc.) how to make an array and perform different operations (traversing, updating, removing, etc.) on arrays.

1. Lab Objectives

The objective of this lab is to introduce students to the concept of lists in python. In this lab students will enable the concepts of making a linked list, types of linked list, accessing the elements of linked list and operations which can be performed on linked list. Students will be provided with examples, followed by performing lab tasks.

2. Lab Outcomes

- CLO:1 Recognize the usage of fundamental Data structure using Python Programming Language.
- CLO:3 Demonstrate solution to real life problems using appropriate data structures.

3. Equipment

- Software
    - IDLE (Python 3.11)

4. Instructions

1. This is an individual lab. You will perform the tasks individually and submit a report.
2. Some of these tasks (marked as ‘Example’) are for practice purposes only while others (marked as ‘Task’) have to be answered in the report.
3. When asked to display an output in the task, either save it as jpeg or take a screenshot, in order to insert it in the report.
4. The report should be submitted on the given template, including:
    a. Code (copy and pasted, NOT a screenshot)
    b. Output figure (as instructed in 3)
    c. Explanation where required
5. The report should be properly formatted, with easy to read code and easy to see figures.
6. Plagiarism or any hint thereof will be dealt with strictly. Any incident where plagiarism is caught, both (or all) students involved will be given zero marks, regardless of who copied whom. Multiple such incidents will result in disciplinary action being taken.
7. Late submission of report is allowed within 03 days after lab with 20% deduction of marks every day.
8. You have to submit report in pdf format (Reg.X_DSA_LabReportX.pdf).

5. Background

Linked List

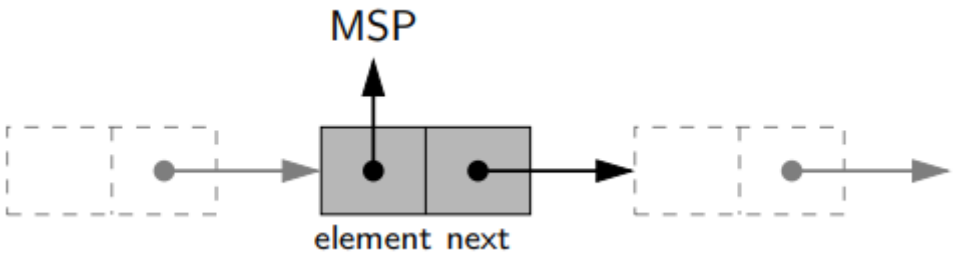
A linked list relies on a more distributed representation in which a lightweight object, known as a node, is allocated for each element. Each node maintains a reference to its element and one or more references to neighboring nodes in order to collectively represent the linear order of the sequence.

Types of Linked List:

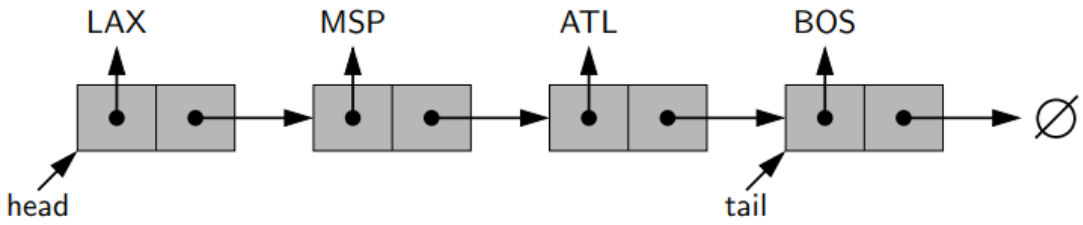
- Singly Linked Lists
- Circular Linked Lists
- Doubly Linked Lists

Singly Linked Lists:

A ***singly linked list***, in its simplest form, is a collection of ***nodes*** that collectively form a linear sequence. Each node stores a reference to an object that is an element of the sequence, as well as a reference to the next node of the list.



Above example of a node instance that forms part of a singly linked list. The node's element member references an arbitrary object that is an element of the sequence (the airport code MSP, in this example), while the next member references the subsequent node of the linked list (or None if there is no further node).



Above example of a singly linked list whose elements are strings indicating airport codes. The list instance maintains a member named head that identifies the first node of the list, and in some applications another member named tail that identifies the last node of the list. The None object is denoted as Ø.

The first and last node of a linked list are known as the ***head*** and ***tail*** of the list, respectively. By starting at the head, and moving from one node to another by following each node's next reference, we can reach the tail of the list. We can identify the tail as the node having None as its next reference. This process is commonly known as ***traversing*** the linked list. Because the next reference of anode can be

viewed as a ***link*** or ***pointer*** to another node, the process of traversing list is also known as ***link hopping*** or ***pointer hopping***.

Example 1: In this example, the Node class represents each node in the linked list, with a piece of data and a reference to the next node. The LinkedList class manages the singly linked list itself, with methods for adding and printing nodes in the list.

Source code:

```
# Define a Node class to represent each node in the linked list
class Node:
    def __init__(self, data):
        # Each node has a piece of data and a reference to the next node
        self.data = data
        self.next = None

# Define a LinkedList class to manage the linked list
class LinkedList:
    def __init__(self):
        # Initialize the head of the list to None
        self.head = None

    def append(self, data):
        # Create a new node with the given data
        new_node = Node(data)

        # If the list is empty, set the new node as the head of the list
        if self.head is None:
            self.head = new_node
            return

        # If the list is not empty, traverse to the end of the list and add the new node
        last_node = self.head
        while last_node.next:
            last_node = last_node.next
        last_node.next = new_node

    def print_list(self):
        # Traverse the list starting from the head and print each node's data
        current_node = self.head
        while current_node:
            print(current_node.data)
            current_node = current_node.next

# Create a new linked list and append some nodes
my_list = LinkedList()
my_list.append(10)
my_list.append(20)
```

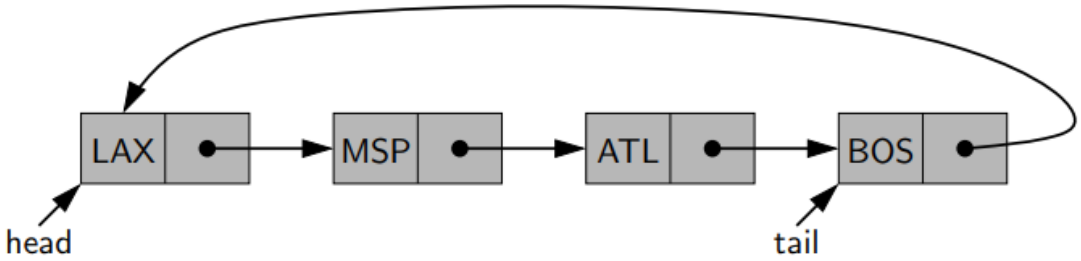
```
my_list.append(30)
my_list.append(40)

# Print the contents of the list
my_list.print_list()
```

Output:

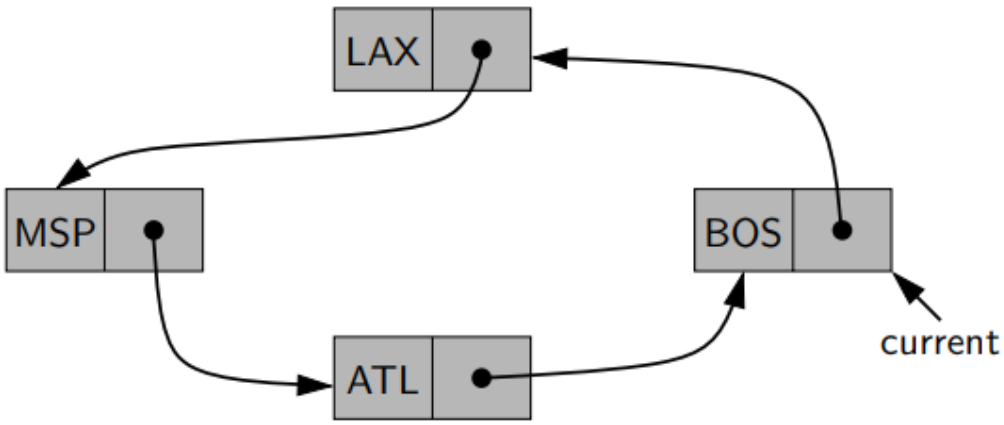
```
10
20
30
40
```

Circular Linked Lists:

In the case of linked lists, there is a more tangible notion of a circularly linked list, as we can have the tail of the list use its next reference to point back to the head of the list, as shown in below Figure. We call such a structure a ***circularly linked list***.



A circularly linked list provides a more general model than a standard linked list for data sets that are cyclic, that is, which do not have any particular notion of a beginning and end.



A circular view similar to above Figure could be used, for example, to describe the order of train stops in the Chicago loop, or the order in which players take turns during a game. Even though a circularly linked list has no beginning or end, per se, we must maintain a reference to a particular node in order to make use of the list. We use the identifier current to describe such a designated node. By setting current = current.next, we can effectively advance through the nodes of the list.

Example 2: In this example, the Node class represents each node in the linked list, with a piece of data and a reference to the next node. The CircularLinkedList class manages the circular linked list itself, with methods for adding and printing nodes in the list.

Source code:

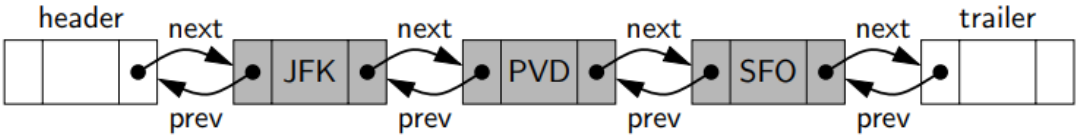
```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None

class CircularLinkedList:
    def __init__(self):
        self.head = None

    # Function to add node to the end of the list
    def append(self, data):
        new_node = Node(data)
        if self.head is None:
            self.head = new_node
            new_node.next = self.head
            return

        current_node = self.head
        while current_node.next != self.head:
            current_node = current_node.next

        current_node.next = new_node
        new_node.next = self.head

    # Function to display the contents of the list
    def display_list(self):
        if self.head is None:
            print("List is empty")
            return

        current_node = self.head
        while True:
            print(current_node.data, end=' ')
            current_node = current_node.next
            if current_node == self.head:
                break
        print()

# Create a new circular linked list
clist = CircularLinkedList()
```

clist.display_list()
 # Append some nodes to the list
clist.append(1)
clist.append(2)
clist.append(3)

 # Display the contents of the list
clist.display_list() # Output:

Output:
List is empty
1 2 3

Doubly Liked Lists:
To provide greater symmetry, we define a linked list in which each node keeps an explicit reference to the node before it and a reference to the node after it. Such a structure is known as a ***doubly linked list***. These lists allow a greater variety of $O(1)$-time update operations, including insertions and deletions at arbitrary positions within the list. We continue to use the term "next" for the reference to the node that follows another, and we introduce the term "prev" for the reference to the node that precedes it.
In order to avoid some special cases when operating near the boundaries of a doubly linked list, it helps to add special nodes at both ends of the list: a ***header*** node at the beginning of the list, and a ***trailer*** node at the end of the list. These "dummy" nodes are known as ***sentinels*** (or guards), and they do not store elements of the primary sequence. A doubly linked list with such sentinels is shown in Figure.



When using sentinel nodes, an empty list is initialized so that the next field of the header points to the trailer, and the prev field of the trailer points to the header; the remaining fields of the sentinels are irrelevant (presumably None, in Python). For a nonempty list, the header's next will refer to a node containing the first real element of a sequence, just as the trailer's prev references the node containing the last element of a sequence.

Example 3: In this example, the Node_of_doubly_linked_list class represents each node in the linked list, with a piece of data and a reference to the next node. The DoublyLinkedList class manages the doubly linked list itself, with methods for adding and printing nodes in the list.

Source code:
 # Doubly Linked List

```
# Define a class for the doubly linked list node
class Node_of_doubly_linked_list:
    def __init__(self, data):
        self.data = data
        self.next = None
        self.prev = None

# Define a class for the doubly linked list
class a_Doubly_Linked_List:
    def __init__(self):
        self.head = None

    # Function to add a new node at the end of the list
    def append(self, data):
        new_node = Node_of_doubly_linked_list(data)
        if self.head is None:
            self.head = new_node
        else:
            current_node = self.head
            while current_node.next is not None:
                current_node = current_node.next
            current_node.next = new_node
            new_node.prev = current_node

    # Function to display the contents of the list
    def display(self):
        current_node = self.head
        while current_node is not None:
            print(current_node.data)
            current_node = current_node.next

# Example usage
my_list = a_Doubly_Linked_List()
my_list.append(1)
my_list.append(2)
my_list.append(3)
my_list.display()
```

Output:

1
2
3

6. Lab Tasks:

Task 1: Write a Python program to make a singly linked list using above class in example 1. Make a method (add_node) to insert a data of your choice after 1st node of list. Also make a method (delete_node) to delete the data from tail of list. Also make a method (print_list) to print the list after every step.

Task 2: Write a Python program to make a circular linked list using above class in example 2. Make a method (manipulate_node) to manipulate a data of your choice at central node of list. Also make a method (delete_current_node) to delete the data from current node of list. Also make a method (print_list) to print the list after every step.

Task 3: Write a Python program to make a doubly linked list using above class in example 3. Make a method (next_of_node) to add a new node containing data to the front of the list and a method (prev_of_node) to add a new node containing data to the back of the list. Also make a method (remove_next) to remove and return the data from the first node in the list and a method (remove_prev) to remove and return the data from the last node in the list. Also make a method (print_list) to print the list after every step.

Lab Evaluation Rubrics							
Domain	**CLOs/ Rubric**	**Performance Indicator**	**Unsatisfactory 0-2**	**Marginal 3-5**	**Satisfactory 6-8**	**Exemplary 9-10**	**Allocated Marks**
Psychomotor	**CLO:1 R2**	Implementation with Results **(P)**	Does not try to solve problems. Many mistakes in code and difficult to comprehend for the instructor. There is not result of the problem.	Does not suggests or refine solutions but is willing to try out solutions suggested by others. Few mistakes in code, but done along with comments, and easy to comprehend for the instructor. Few mistake in result.	Refines solutions suggested by others. Complete and error-free code is done. No comments in the code, but easy to comprehend for the instructor. Results are correctly produced.	Actively looks for and suggests solution to problems. Complete and error free code is done, easy to comprehend for the instructor. Results are correctly produced. Student incorporated comments in the code.	
Affective	**CLO:3 R3**	Lab Report **(A)**	Code of the problem is not given. Outputs are not provided. Explanation of the solution is not stated.	Code of the problem is not given. Output is not complete. Explanation of the solution is not satisfactory.	Code of the problem is not given. Output is completely given. Explanation of the solution is not satisfactory.	Code of the problem is not given. Output is completely given. Explanation of the solution is satisfactory.	
	CLO:1 R5	Discipline and Behavior **(A)**	Got and wandered around. More than two incidents of talking non-lab related stuff in laband/or any talk with other groups, voice level exceeding the appropriate level, use of cell phones and involvement in any non-lab activity.	Got out of seat and wander around for some time. No more than two incidents of talking non-lab related stuff in lab Voice level exceeding theappropriate level, use of cell phones and involvement in any non-lab related activity.	Stayed in seat and got up for a specific lab related reason, but took more time than required to do the job. No more than one incidents of talking non-lab related stuff in lab. Voice level exceedingthe appropriate level, use of cell phones and involvementin any non-lab related activity.	Stayed in seat and got up for a specific lab related reason. Tookcare of lab related business and sat down right away. Voice level kept appropriate. Not used cell phones or involved in any non- lab related activity.	