

# BridgeLabz

Employability Delivered

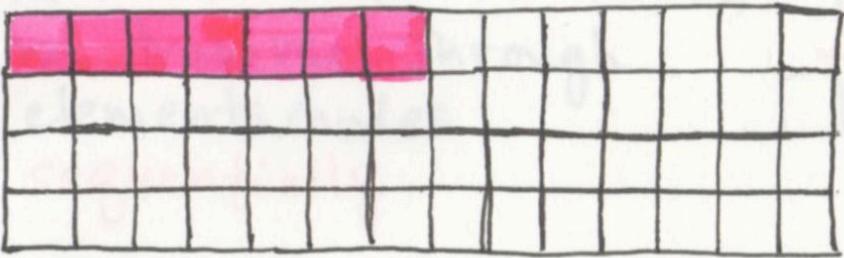
## C# Core Concepts – Data Structures using Generics

# Data Structures

- Means of **Organizing and Storing Data** in a way **Operations** can be performed Efficiently.
- Here are the commonly used Data Structures
  - **Arrays** – An array is a continuous structure of fixed size. Array Operations include Traversal, Search, Update but not Insert or Delete.
  - **Linked Lists** – It is a sequence of items linked in a linear order with each other. Hence Random Access to Data is not possible
  - **Stack & Queues** – Stack is a Last in First Out Structure (**LIFO**) while Queues are First in First Out Structure (**FIFO**)
  - **Hash Tables** – Stores values which have keys associated with each of them, It's a **(K,V) pair**.
  - **Trees** – Is a structure where data is organized **Hierarchically** and linked together.

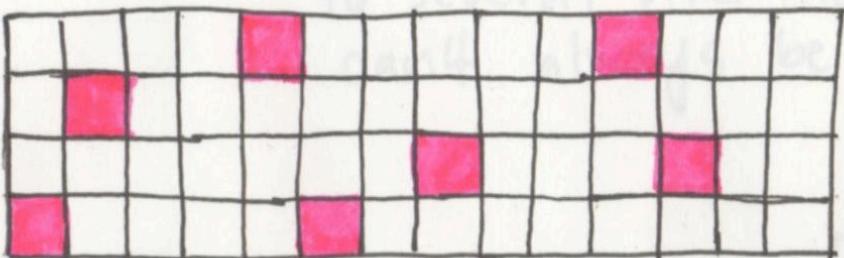
# Memory Allocation

STATIC



Arrays need a contiguous block of memory.

DYNAMIC C-BASIC

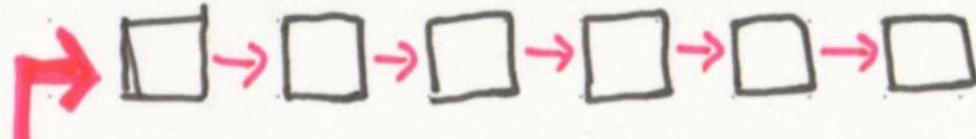


Linked lists don't need to be contiguous in memory; they can grow dynamically.

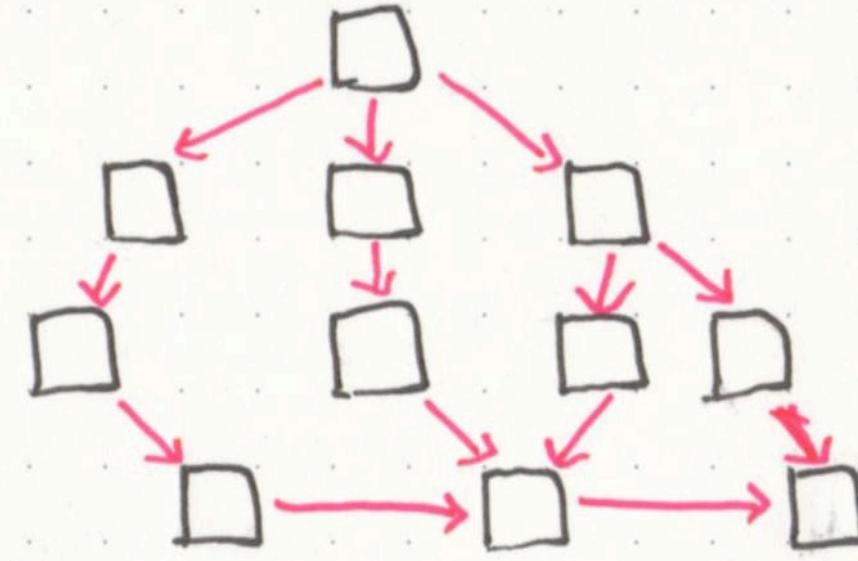
■ = one byte of used memory

## Arrays & Linked List

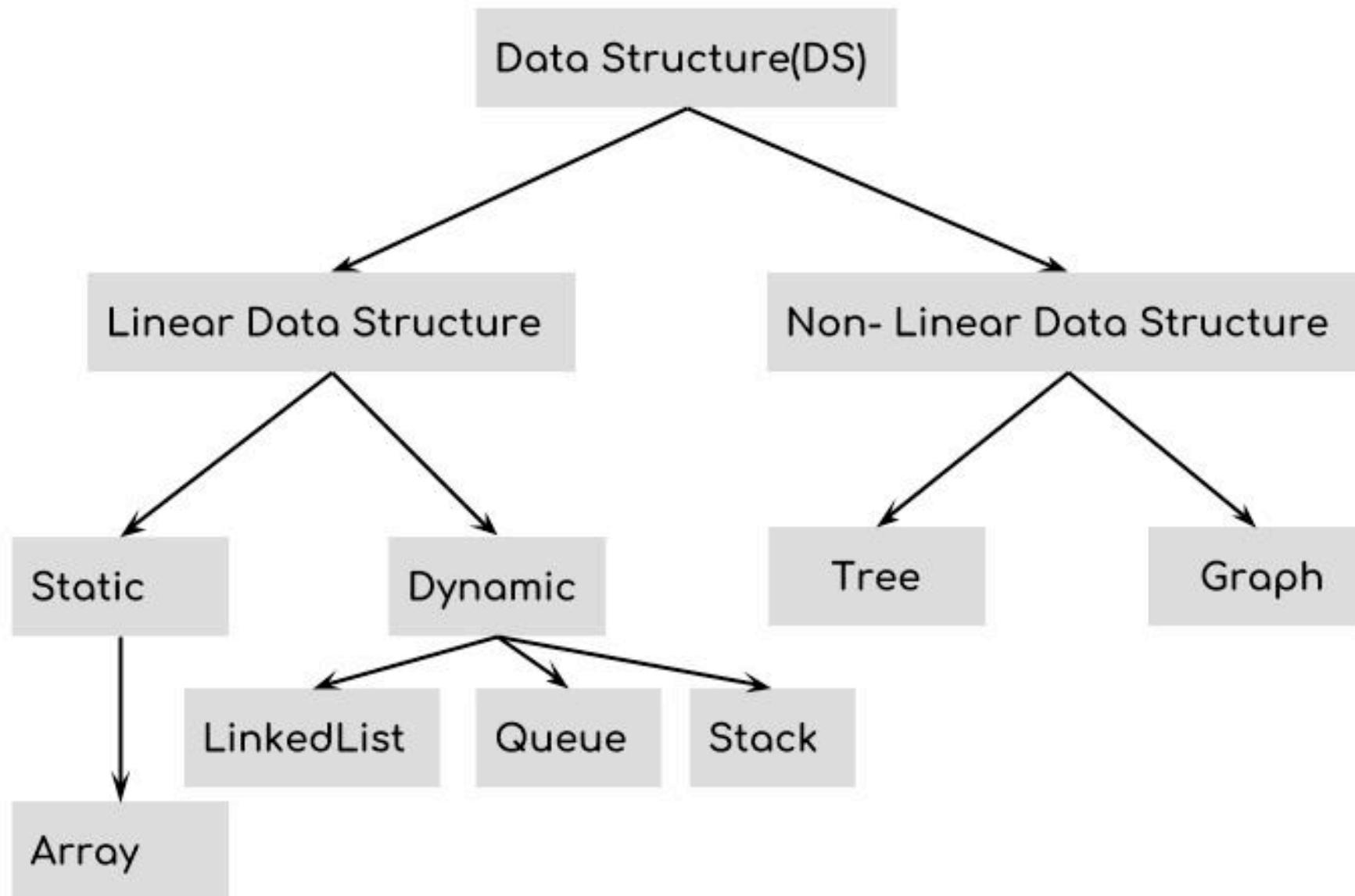
## Linear vs. Non-linear structures



In **linear** data structures,  
we traverse through  
elements/nodes  
**sequentially**.



In **non-linear** data structures,  
one node might be connected  
to several other nodes, so it  
can't always be traversed sequentially.

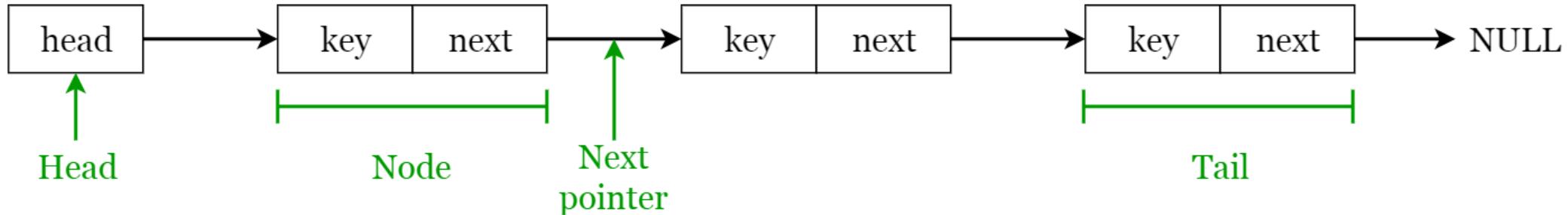


# Data Structures

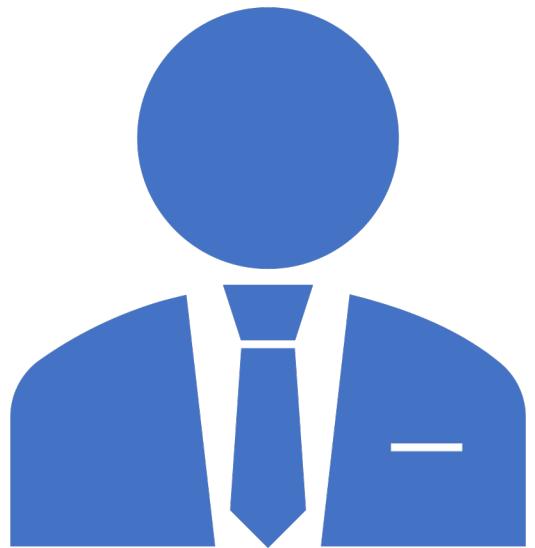
# Section 1: LinkedList

---

# LinkedList

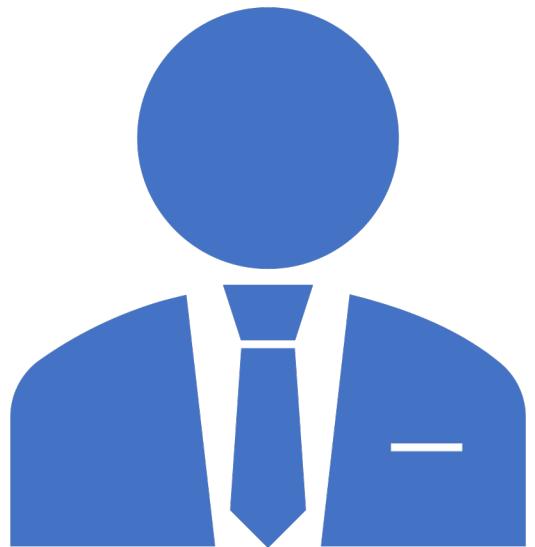


- Elements in a linked list are known as nodes.
- Each **Node** contains a **key** and a pointer to its successor node, known as **next**.
- The attribute named **head** points to the first element of the linked list.
- The last element of the linked list is known as the **tail**.



**UC 1**

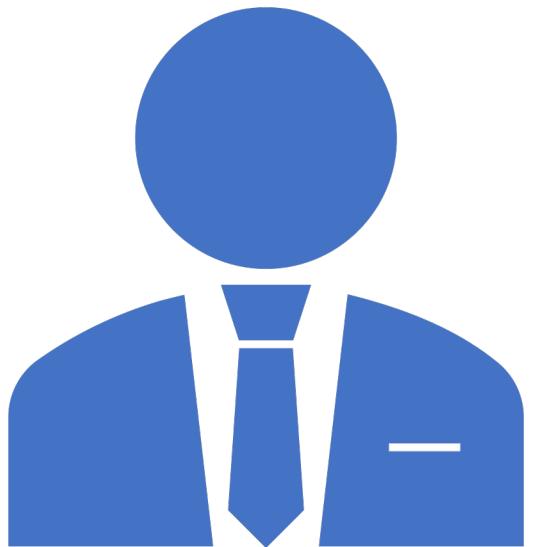
Lets create a  
simple Linked List  
of 56, 30 and 70



**UC 2**

**Ability to create Linked List by adding 30 and 56 to 70**

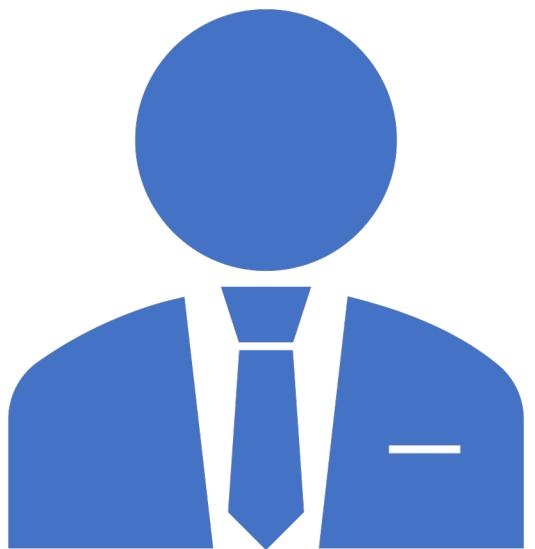
- Node with data 70 is First Created
- Next 30 is added to 70
- Finally 56 is added to 30
- LinkedList Sequence: 56->30->70



**UC 3**

Ability to create Linked List by appending 30 and 70 to 56

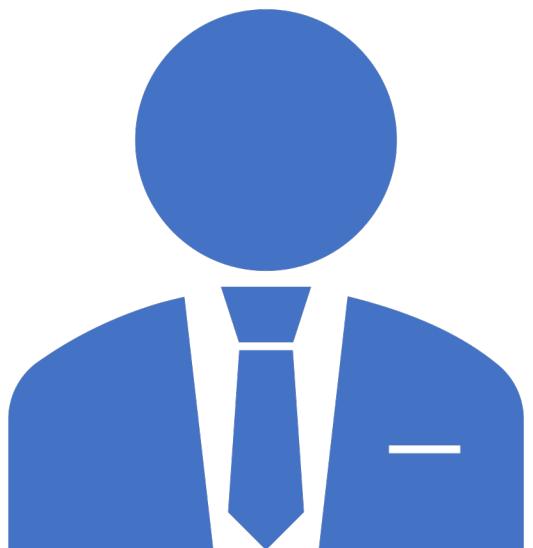
- Node with data 56 is First Created
- Next Append 30 to 56
- Finally Append 70 to 30
- LinkedList Sequence: 56->30->70



**UC 4**

**Ability to insert  
30 between 56  
and 70**

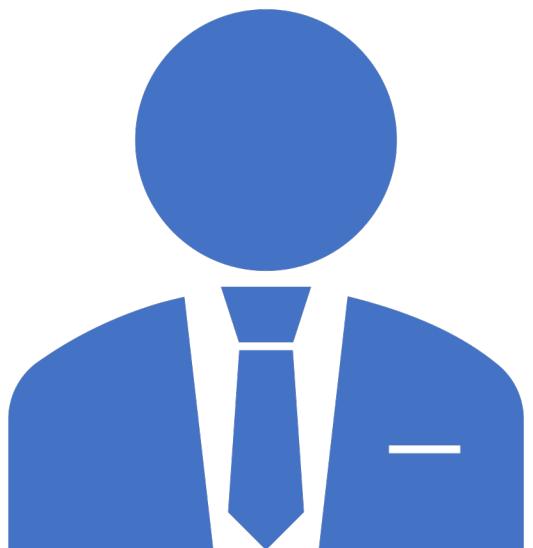
- Final Sequence: 56->30->70



**UC 5**

Ability to delete the first element in the LinkedList of sequence 56->30->70

- Write pop method.
- Note there is new head
- Final Sequence: 30->70



UC 6

Ability to delete the last element in the LinkedList  
of sequence 56->30->70

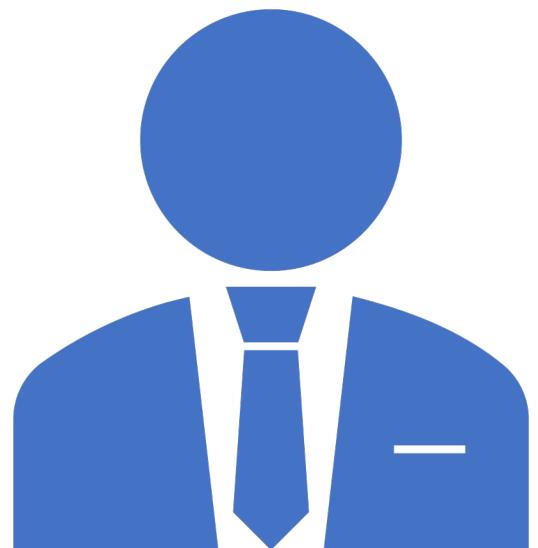
- Write popLast method
- Note there is new tail
- Final Sequence: 56->**30**

# The Unordered List Abstract Data Type

The structure of an unordered list, as described above, is a collection of items where each item holds a relative position with respect to the others. Some possible unordered list operations are given below.

- `List()` creates a new list that is empty. It needs no parameters and returns an empty list.
- `add(item)` adds a new item to the list. It needs the item and returns nothing. Assume the item is not already in the list.
- `remove(item)` removes the item from the list. It needs the item and modifies the list. Assume the item is present in the list.
- `search(item)` searches for the item in the list. It needs the item and returns a boolean value.
- `isEmpty()` tests to see whether the list is empty. It needs no parameters and returns a boolean value.
- `size()` returns the number of items in the list. It needs no parameters and returns an integer.
- `append(item)` adds a new item to the end of the list making it the last item in the collection. It needs the item and returns nothing. Assume the item is not already in the list.
- `index(item)` returns the position of item in the list. It needs the item and returns the index. Assume the item is in the list.
- `insert(pos, item)` adds a new item to the list at position pos. It needs the item and returns nothing. Assume the item is not already in the list and there are enough existing items to have position pos.
- `pop()` removes and returns the last item in the list. It needs nothing and returns an item. Assume the list has at least one item.

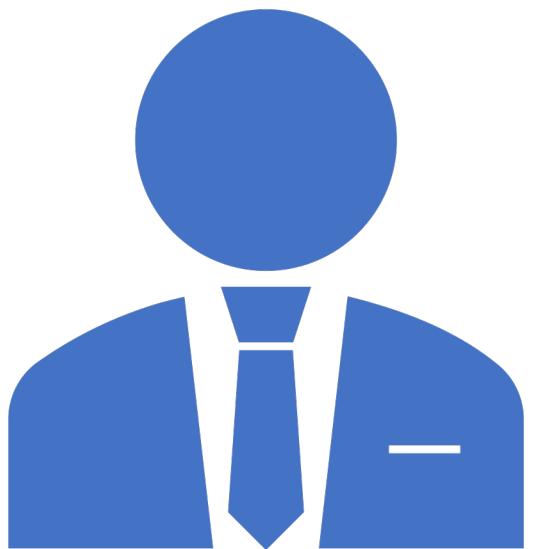
UnOrdered  
LinkedList  
API



UC 7

Ability to search  
LinkedList to find Node  
with value 30

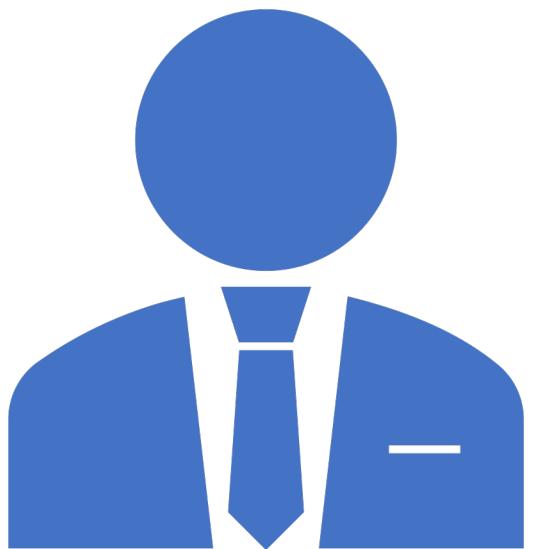
- Write MSTest Test Case as demonstrated in class
- Loop through LinkedList to find node with key 30



**UC 8**

Ability to insert 40 after 30 to  
the Linked List sequence of  
 $56 \rightarrow 30 \rightarrow 70$

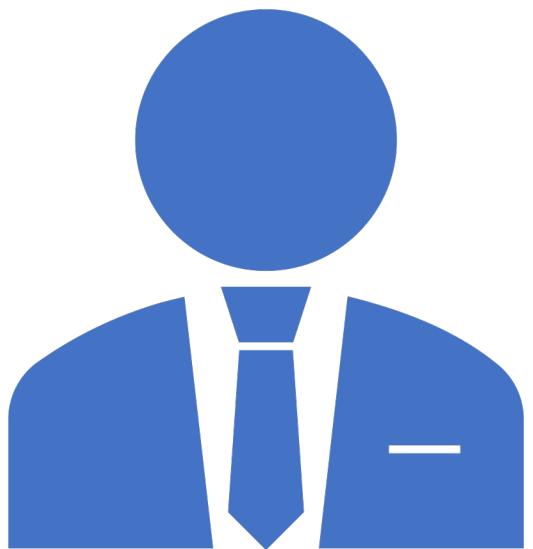
- Write MSTest Test Case as demonstrated in class
- Search LinkedList to find Node with key value 30
- Then Insert 40 to 30
- Final Sequence:  $56 \rightarrow 30 \rightarrow 40 \rightarrow 70$



**UC 9**

Ability to delete 40 from the Linked List sequence of 56->30->40->70 and show the size of LinkedList is 3

- Write MSTest Test Case as demonstrated in class
- Search LinkedList to find node with key value 40
- Delete the node
- Implement size() and show the Linked List size is 3
- Final Sequence: 56->30->70



**UC 10**

Ability to create Ordered Linked List  
in ascending order of data entered  
in following sequence 56, 30, 40,  
and 70

- Refactor the code to create SortedLinkedList Class
- Create Node that takes data that is Comparable
- Perform Sorting during the add method call
- Final Sequence: 30->40->56->70

# The Ordered List Abstract Data Type

We will now consider a type of list known as an ordered list. For example, if the list of integers shown above were an ordered list (ascending order), then it could be written as 17, 26, 31, 54, 77, and 93. Since 17 is the smallest item, it occupies the first position in the list. Likewise, since 93 is the largest, it occupies the last position.

The structure of an ordered list is a collection of items where each item holds a relative position that is based upon some underlying characteristic of the item. The ordering is typically either ascending or descending and we assume that list items have a meaningful comparison operation that is already defined. Many of the ordered list operations are the same as those of the unordered list.

- `OrderedList()` creates a new ordered list that is empty. It needs no parameters and returns an empty list.
- `add(item)` adds a new item to the list making sure that the order is preserved. It needs the item and returns nothing. Assume the item is not already in the list.
- `remove(item)` removes the item from the list. It needs the item and modifies the list. Assume the item is present in the list.
- `search(item)` searches for the item in the list. It needs the item and returns a boolean value.
- `isEmpty()` tests to see whether the list is empty. It needs no parameters and returns a boolean value.
- `size()` returns the number of items in the list. It needs no parameters and returns an integer.
- `index(item)` returns the position of item in the list. It needs the item and returns the index. Assume the item is in the list.
- `pop()` removes and returns the last item in the list. It needs nothing and returns an item. Assume the list has at least one item.
- `pop(pos)` removes and returns the item at position pos. It needs the position and returns the item. Assume the item is in the list.

## Ordered LinkedList API

# MyLinkedList Code Snippet

```
using System;
using System.Collections.Generic;
using System.Text;

namespace LinkedListProblem
{
    public class LinkedList
    {
        internal Node head;
        internal void Add(int data)
        {
            Node node = new Node(data);
            if (this.head == null)
            {
                this.head = node;
            }
            else
            {
                Node temp = head;
                while (temp.next != null)
                {
                    temp = temp.next;
                }
                temp.next = node;
            }
            Console.WriteLine("{0} inserted into linked list", node.data);
        }
        internal void Display()
        {
            Node temp = this.head;
            if (temp == null)
            {
                Console.WriteLine("LinkedList is empty");
                return;
            }
            while (temp != null)
            {
                Console.Write(temp.data + " ");
                temp = temp.next;
            }
        }
    }
}
```

```
internal Node RemoveFirstNode()
{
    if (this.head == null)
    {
        return null;
    }
    this.head = this.head.next;
    return this.head;
}
internal Node RemoveLastNode()
{
    if (head == null)
    {
        return null;
    }
    if (head.next == null)
    {
        return null;
    }
    Node NewNode = head;
    while (NewNode.next.next != null)
    {
        NewNode = NewNode.next;
    }
    NewNode.next = null;
    return head;
}
internal Node Search(int value)
{
    while (this.head != null)
    {
        if (this.head.data == value)
        {
            return this.head;
        }
        this.head = this.head.next;
    }
    return null;
}
```

# MSTest Test Cases

```
public class MyLinkedListTest {  
  
    @Test  
    public void given3NumbersWhenAddedToLinkedListShouldBeAddedToTop() {...}  
  
    @Test  
    public void given3NumbersWhenAppendedShouldBeAddedToLast() {...}  
  
    @Test  
    public void given3NumbersWhenInsertingSecondInBetweenShouldPassLinkedListRes...  
  
    @Test  
    public void givenFirstElementWhenDeletedShouldPassLinkedListResult() {...}  
  
    @Test  
    public void givenLastElementWhenDeletedShouldPassLinkedListResult() {...}  
}
```

# Types of LinkedList

- **Singly linked list** — Traversal of items can be done in the forward direction only using `next` pointer, pointing to the next node.
- **Doubly linked list** — Traversal of items can be done in both forward and backward directions. Nodes consist of an additional pointer known as `prev`, pointing to the previous node.
- **Circular linked lists** — Linked lists where the `prev` pointer of the head points to the tail and the `next` pointer of the tail points to the head e.g. Alt + Tab



# BridgeLabz

Employability Delivered

Thank  
You

