Lists

A list is a popular data structure to store data in sequential order. For example, a list of students, a list of available rooms, a list of cities, and a list of books, etc. can be stored using lists. The common operations on a list are usually the following:

- Retrieve an element from this list.
- Insert a new element to this list.
- Delete an element from this list.
- Find how many elements in this list.
- Find if this list is full.
- Find if this list is empty.

Two Ways to Implement Lists

There are two ways to implement a list:

One is to use an array to store the elements.

Two is to use a linked structure. A linked structure consists of nodes. Each node is dynamically created to hold an element. All the nodes are linked together to form a list.(linked list)

1- Array Lists

Array is a fixed-size data structure. Once an array is created, its size cannot be changed. Nevertheless, you can still use array to implement dynamic data structures. The trick is to create a new larger array to replace the current array if the current array cannot hold new elements in the list.

Initially, an array, say data of Object[] type, is created with a default size. When inserting a new element into the array, first ensure there is enough room in the array. If not, create a new array with the size as twice as the current one. Copy the elements from the current array to the new array. The new array now becomes the current array.

The ArrayLists Abstract Data Type

Class specification :

List
-MaxSize:int **- items[]:object;** **-count:int**
+List(int) **+ insert (int,object):void** **+ remove(int):void** **+retrieve(int):object** **+IsFull():bool** **+IsEmpty():bool** **+Size():int** **.** **.**

The List abstract data type (ADT) supports the following:

ADT : List
{
Data: a non zero positive integer number representing MaxSize . and array of object elements represent the items.

Operations:
A constructor(List) :initialize the data to some Data object certain value.

insert (pos,element): Insert object **element** into **items** at position **pos**.

Input : Position **p**os and Object **element** Output: None.

remove (pos): remove from **items** the object at position **pos**.

Input : Position **p**os; Output: None.

retrieve (pos): Return, but do not remove, from **items** the object at position **p**os.

Input : Position **pos**; Output: object.

IsFull() : Return a Boolean value indicating if the list is full.

Input : None; Output: Boolean.

IsEmpty() : Return a Boolean value indicating if the list is empty.

Input : None; Output: Boolean.

Size(): Return the number of objects in the list .

Input : None; Output: Integer.

End ADT List

We illustrate the operation in the list ADT in the following example:

Example: the following table shows a series of list operation and their effect on an initially empty list items of integer and MaxSize = 5

Operation	**Output**	items
insert (0,5)	-	(5)
insert(1,3)	-	(5, 3)
insert(2,7)	-	(5,3,7)
insert(1,8)	-	(5,8,3, 7)
insert(0,9)	-	(9,5,8,3, 7)
insert(3,5)	**Error**	(9, 5,8,3, 7)
remove (1)		(9,8,3, 7)
remove (0)		(8,3, 7)
retreve (1)	**3**	(8,3, 7)
Size()	**3**	(8,3, 7)

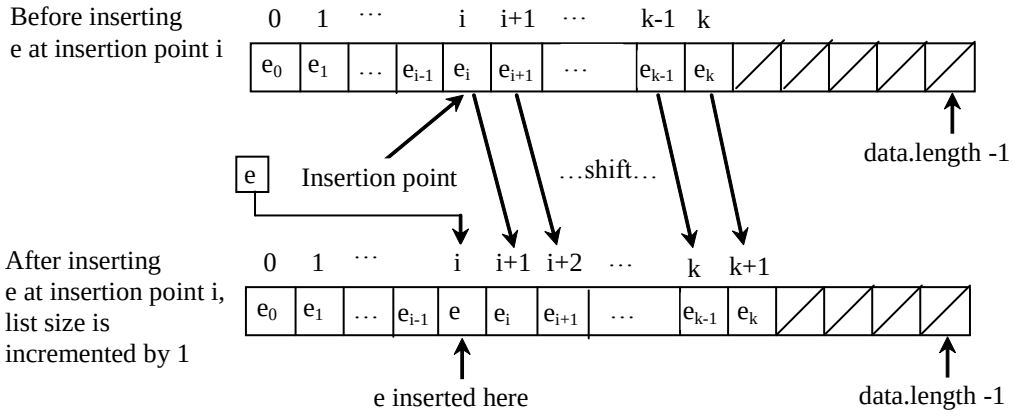
```
class List
  {
// data member  or data value
      private int MaxSize, count;
      private  object[] items;
// Constructer    or default   Constructer
        public List(int n)
      {
        MaxSize = n;
        items = new object[MaxSize];
        count=0;
      }
```

List Operations

We describe how to use this method to implement a list in code :

Pseudocode insert(pos, element)

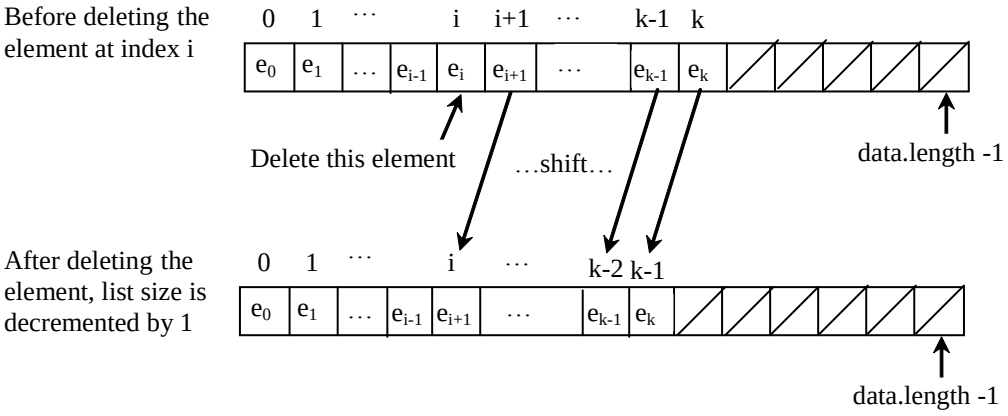
Before inserting a new element at a specified index, shift all the elements after the index to the right and increase the list size (count) by 1.




```
if  IsFull()  then
        print error  a listFull
else
     if (pos greater than or equal   0  and pos Less than or equal  count)
      {
           for (i= count-1 to  i>= pos , i--)        // shift elements right
                    items[i+1] ← items[i]
        items[pos] ← element          // insert  object in the specified pos
        count←count+1
     }
    else     print error out of range
```

Pseudocode remove(pos)

To remove an element at a specified index, shift all the elements after the index to the left by one position and decrease the list size (count) by 1.



```
if  IsEmpty()  then
        print  error  a listEmpty
else
      {
          if (pos greater than or equal  0  and pos  less than  count)
             {
                  for (i= pos , i< count-1 , i++)        // shift elements left
                    items[i] ←items[i+1]
                count← count-1
             }
           else     print   error out of range
         }
```

Pseudocode retrieve (pos)

```
if  IsEmpty()  then
        print  error  a listEmpty
else
    if (pos greater than or equal  0  and pos less than  count)
       return (items[pos])
     else     print   error out of range
```

Pseudocode Size():

```
return count
```

Pseudocode IsEmpty():

```
if ( count equal  0)   return true
else    return    false
```

Pseudocode IsFull():

```
if ( count  equal  Maxsize)  return  true
else   return  false
```

The table show the running times of methods in realization of a list by an array

Method	Worst case	Best case	Average case
insert	**O(n)** if position of element in the first of the list and count of elements in list not equal zero.	O(1) if position of element in the end of the list. or the count of elements in list equal zero.	O(log n) If position of element in the middle of the list
Remove	**O(n)** if position of element in the first of the list and count of elements in list not equal 1.	O(1) if position of element in the end of the list. or the count of elements in list equal 1	O(log n) If position of element in the middle of the list
Retrieve	O(1)	O(1)	O(1)
IsFull()	O(1)	O(1)	O(1)
IsEmpty	O(1)	O(1)	O(1)

Exercises

Q1/ Consider the following operations on a List data structure that stores integer values?

List L1=new List(7);

L1.insert(0,6);

L1.insert(1,21);

L1.insert(2,3);

L1.remove(1);

L1.insert(0,7);

L1.remove(2);

L1.insert(1,15);

L1.remove(1);

what s will look like after the code above executes?

Q2/ given the following list:

Write a segment of code of c# program to perform the followings:

a-Create a list of 10 objects.

b-Add the element 70, 80, 90,100 in the list.

c- Delete the first object in the list.

d- Insert one element at position p in a list.

e- Delete all the elements of even position in the list.

f- Count the number of elements in the list.

Q3/ let a=(a1,a2….a10)and b=(b1,b2,….b8)Are two lists. Write C# method to merge the first five elements of the list b at the end of the list a .

Q4/ Define a new method of ArrayList class : void Display () that display all elements of any list .

Q5/ Define a new method of ArrayList class : void DisplayExcept (value) that display all elements of any list except the element of data value44

Q6/ what is the output of the following code ?

a- for (int i=0; i< 5; i++)
Ll.insert(0,i);

b- for (int i=0; i< 5; i++)
Ll.insert(i,0);

c- for (int i=0; i< 5; i++)
Ll.insert(i,i);

Q7: Define a new method of ArrayList class : int Search (element) that search element in the list.

Q8: Define a new method of ArrayList class : void Sort ( ) that sorts elements in the list.

Q9: Define a new method of ArrayList class : int Search1(element) that search element in the sorted list.

2- Linked List :

A Linked List (or one way list) is a linear collection of data elements called (Node) , where "Linear " order is given by means of "pointer" .

-A Linked List is a series of connected (linked) elements.

-Each node contains at least two fields :

-data of any type

-pointer to the next node in the list

-Each linked list must have a Head which is a pointer to the first node.

-The last points to null.

Linked List in C# :

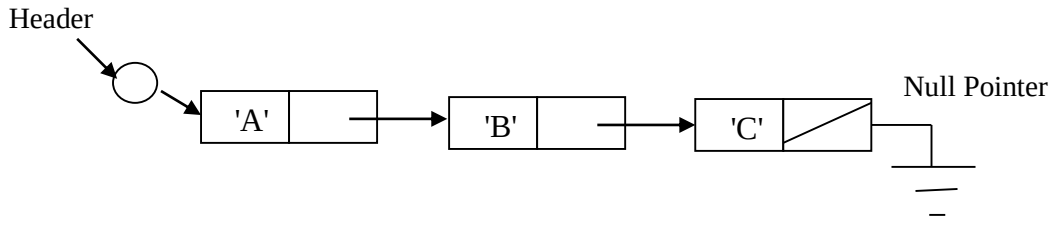
1-A linked list is a collection of class objects called nodes.

2- Each node is linked to its successor node in the list using a reference to the successor node.

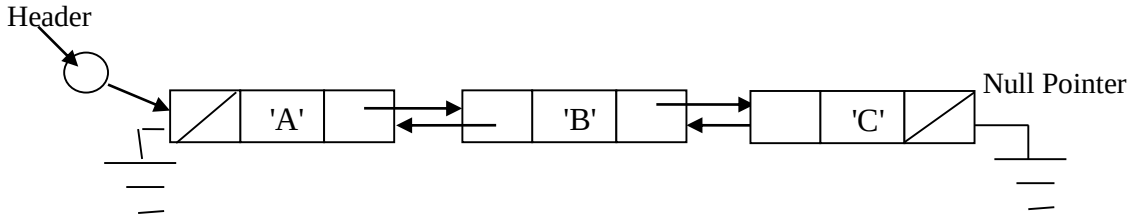
3-A node is made up of a field for storing data and the field for the node reference.

4-The reference to another node is called a link (next).

An example linked list is shown in Figure below.



Single Linked List



Double Linked List

AN OBJECT-ORIENTED LINKED LIST DESIGN

1-Linked list will involve at least two classes.

2-We'll create a Node class and instantiate a Node object each time we add a node to the list.

3-The nodes in the list are connected via references to other nodes.

4-These references are set using methods created in a separate LinkedList class.

5- Let's start by looking at the design of the Node class.

Traversing a Linked List:

-Traversal means "visiting " or examining each node .

-Algorithm :

- start at the beginning
- Go one node at a time until the end .

Types of Linked List

1-Single Linked List (S.L.L)

The single linked list is the most basic of all the dynamic data structures . a single linked list is simply a sequence of dynamically allocated storage elements , each containing a pointer to its successor and have a pointer to the head of the list called Head .

The Single Linked List Abstract Data Type

The Node Class

Class **Node** specification :

Node
+data:object **+ next:Node**
+Node() **+ Node (object)**

The Node abstract data type (ADT) supports the following:

ADT : Node

{

Data: A node is made up of two data members: data, which stores the node's data; and next, which stores a reference to the next node in the list.

Constructors:

we need at least two constructor methods. We definitely want a

A constructor(Node()) : default constructor that creates an empty Node, with both the data and next members set to null.

A constructor(Node(object)) : assigns data to the data member and sets the next member to null.

} ADT Node

Here's the code for the Node class:

```
public class Node
  {
   // data member or data value
      public object data;
       public Node next;
    // Constructer    or default   Constructer
          public Node()
               {
                  data=null;
                  next=null;
                }
         public Node(object item)
           {
             data=item;
             next=null;
           }
} end of class Node
```

The linkedlist class :

1-The Linked List class is used to create the linkage for the nodes of our linked list.

2-The class includes several methods for adding nodes to the list, removing nodes from the list, traversing the list, and finding a node in the list etc.

3-We also need a constructor method that instantiates a list.

4-The only data member in the class is the head node.

Class **LinkedList** specification :

LinkedList
-Head:Node
+LinkedList() **+ AddAtEnd(object):void** **+ AddAtFront(object):void** **+ DelFront():void** **+ DelEnd():void** **+ Insert(object,object):void** **+ Delete(object):void** **-Find(object):Node** **-FindPre(object):Node** **+IsEmpty():bool** **+Display():void** **+Size():int**

The LinkedList abstract data type (ADT) supports the following:

ADT : LinkedList
{
Data: Head data type class Node

Operations:

A constructor(LinkedList()) :initialize the Head member set to null.

AddAtEnd(object): insert Node at the last position in the Linked List

Input : object Output: None.

AddAtFront(object): insert Node at the first position in the Linked List

Input : object; Output: None.

DelFront(): delete the first node in the Linked List

Input : None ; Output: None

DelEnd(): delete the last Node in the Linked List

Input : None ; Output: None

Insert(object , object): To insert a new node after an existing node, we have to first find the "after" node. To do this, we create a Private method,(**-Find(object))**, that searches through the Element field of each node until a match is found.

Input : two objects Output: None.

Delete(object): (deleting a node with given value) **.** We need to find the node before the node we want to remove, we'll define a Private method, **(- FindPre(object))** . This method walks down the list, stopping at each node and looking ahead to the next node to see if that node's Element field holds the item we want to remove.

Input : object Output: None.

IsEmpty() : Return a Boolean value indicating if the Linkedlist is empty.

Input : None; Output: Boolean.

Size(): Return the number of objects in the LinkedList .

Input : None; Output: Integer.

Display (): display all elements of linked list.

Input: None Output : None

End ADT LinkList

Here's the code for the LinkedList class:

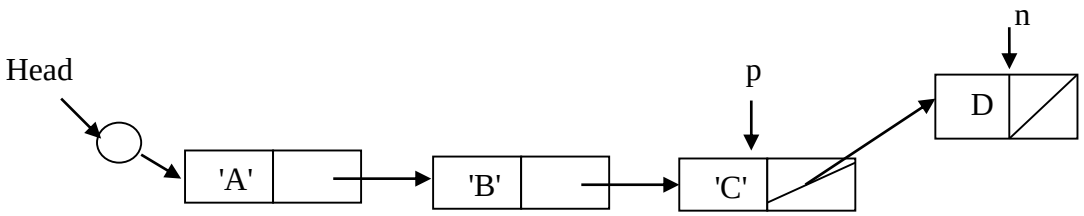
```
class LinkedList
  {
// data member  or data value
  private  Node Head;
// Constructer
      public LinkedList()
       {
         Head=null;
       }
```

Single LinkedList Operations

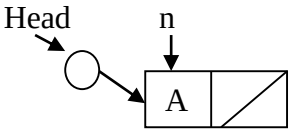
We describe how to use this method to implement a Linked list in code :

Pseudocode AddAtEnd(object item)

Example:

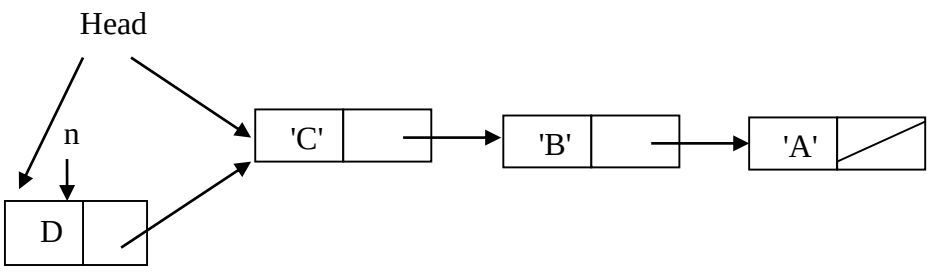


IsEmpty



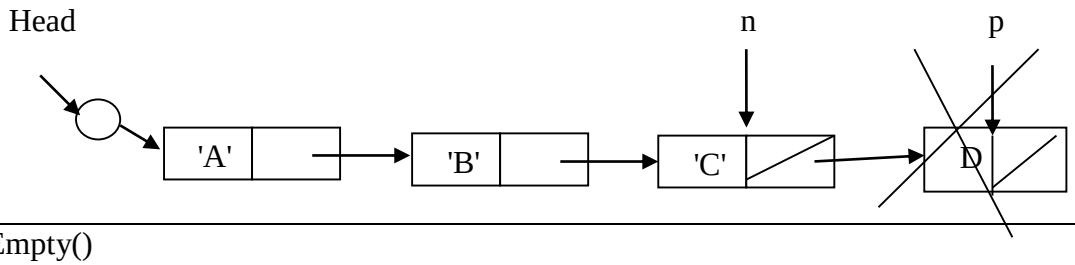

```
Node n = new Node(item)
if   IsEmpty()
     {
          n.next← null
          Head← n
      }
  else
     {
          Node p ← Head
search the last node in the list
          while( p.next notEqual   null )
                    p←p.next
       p.next←n
    }
```

Pseudocode AddAtFront(object item)



```
Node n ← new Node(item)
Link a new node
    n.next← Head
 update  Head pointer
    Head←n
```

Pseudocode_DelEnd()



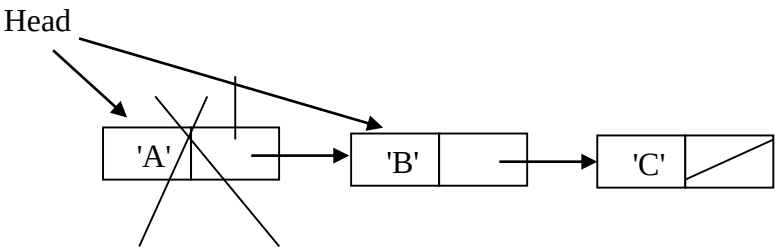

```
if   IsEmpty()
    print linked list empty
else
   {
       Node    p← Head
       Node    n← null


      Move the temporary P to the node that is before the last node
       while(p.next  not equal null)
        {
             n←p
             p←p.next
         }

       Make the node that is before the last node point to Null

       n.next←null
   }
```

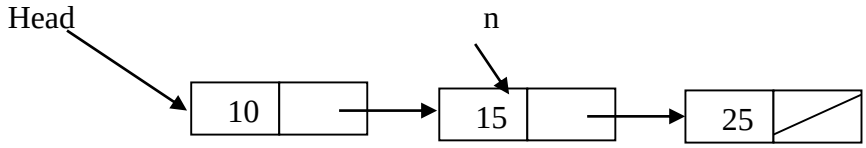
Pseudocode_DelFront()



```
if   IsEmpty()
    print linked list empty
else
   Head← Head.next
```

Pseudocode_Find(object after)

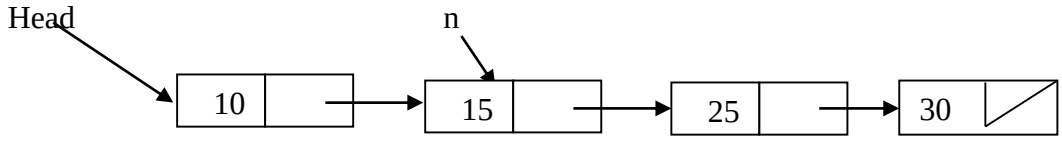
Ex: after= 15




```
Node n← Head;
while  (n not equal null and  n.data not equal after)
    n←n.next ;
return n
```

Pseudocode_FindPre(object value)

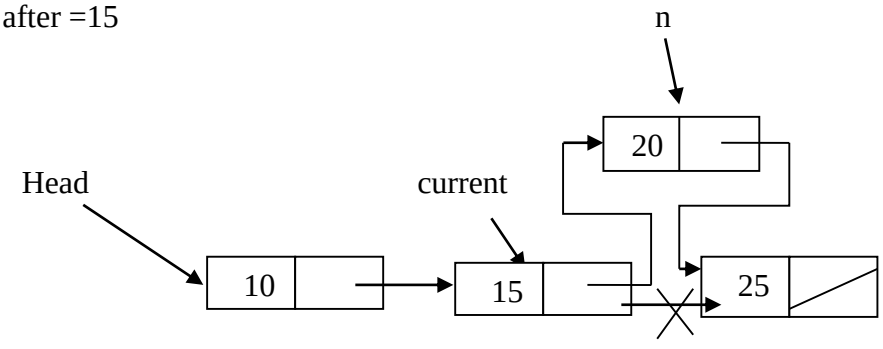
Ex: value =25




```
Node n← Head;
while  (n.next  not equal null and  n.next.data not equal value)
    n←n.next
return n
```

Pseudocode_Insert(object item, object after)

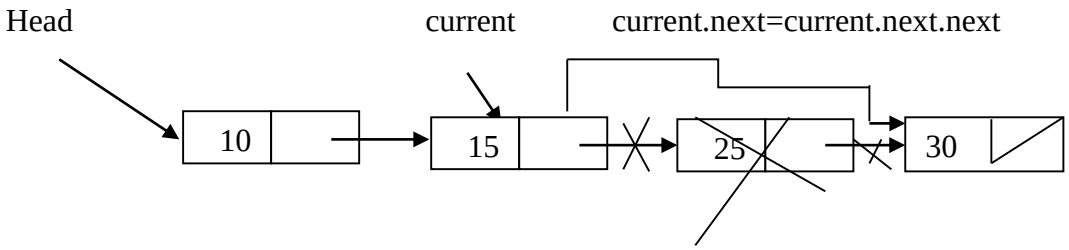
Ex: after =15



```
Node n← new Node(item);
Node current←Find(after)
if (current not equal  null)
 {
    n.next←current.next
    current.next←n
  }
else
 print the after not found in the linked list
```

Pseudocode_delete(object value)

Ex: value =25




```
 if IsEmpty()
      print linked list is empty
else
   {
     Node current ← FindPre(value)
     if    (current.next not equal null)
       current.next ←current.next.next
     else
       print the value not found in the linked list
  }
```

Pseudocode_IsEmpty()

```
if  Head equal null
            return true
else
          return false
```

Pseudocode_Size()

```
if  IsEmpty()
            count←0
 else
   {
        Node p← Head
         while ( p not equal  null )
            {
                 count← coun+1
                 p←p.next
            }
   }
Return count
```

Pseudocode_Display()

```
if  IsEmpty()
            print link list empty
 else
   {
        Node p← Head
         while ( p not equal  null )
            {
                 Print    p.data
                 p←p.next
            }
   }
```

LINKED LIST DESIGN MODIFICATIONS

There are several modifications we can make to our linked list design in order to better solve certain problems. Two of the most common modifications are :

1- **A doubly linked list** makes it easier to move backward through a linked list and to remove a node from the list.

2-**A circularly linked list** is convenient for applications that move more than once through a list.

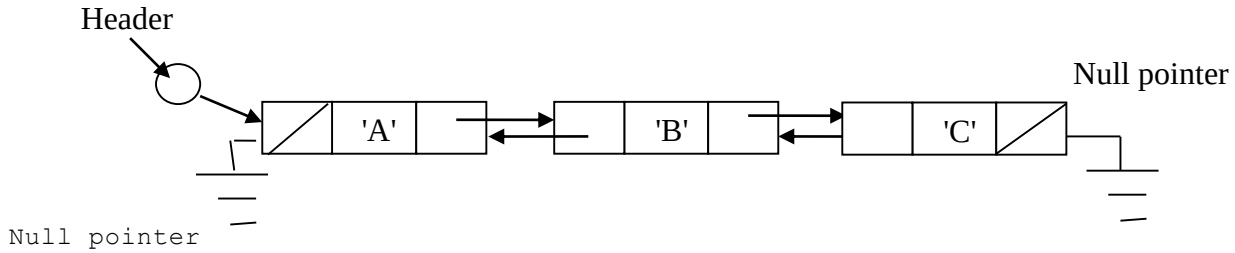
1-The Doubly Linked List

Although traversing a linked list from the first node in the list to the last node is very straightforward, it is not as easy to traverse a linked list backward. We can make this procedure much easier if we add a field to our Node class that stores the link to the previous node. When we insert a node into the list, we'll have to perform more operations in order to assign data to the new

field, but we gain efficiency when we have to remove a node from the list, since we don't have to look for the previous node.

We first need to modify the Node class to add an extra link to the class.

To distinguish between the two links, we'll call the link to the next node the FLink, and the link to the previous node the BLink. These fields are set to Nothing when a Node is instantiated.



Double Linked List

The Node Class

Class **Node** specification :

Node
+data:object **+ Flink:Node** **+ Blink:Node**
+Node() **+ Node (object)**

Here's the code:

```
public class Node
{
    public Object data;
    public Node Flink;
    public Node Blink;

  // Constructors
     public Node()
     {
        Flink = null;
        Blink = null;
      }
     public Node(Object item)
     {
        data = item;
        Flink = null;
        Blink = null;
     }
}
```

The Insertion method is similar to the same method in a singularly linked list, except we have to set the new node's back link to point to the previous node.

```
 Node n← new Node(item)
 Node current← Find(after)
if (current not equal null)
 {
   n.Flink ← current.FLink
   n.Blink ← current
   current.Flink ← n
   n.Flink.Blink ←n
 }
else
 print the after not found in the linked list
```

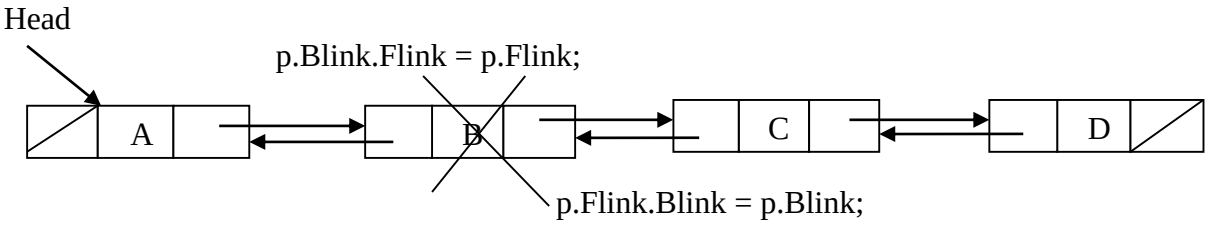
The Remove method for a doubly linked list is much simpler to write than for a singularly linked list.

1-We first need to find the node in the list;

2- then we set the node's back link property to point to the node pointed to in the deleted node's forward link.

3- Then we need to redirect the back link of the link the deleted node points to and point it to the node before the deleted node.

Figure below illustrates a special case of deleting a node (B) from a doubly linked list .



The **Pseudocode** for the Remove method of a doubly linked list is as follows.

```
   if IsEmpty()
       print linked list is empty
else
   {
       Node p ← Find(n)
```

```
        if (!(p.Flink Equal  null))
            {
                p.Blink.Flink ← p.Flink
                p.Flink.Blink ← p.Blink
            }
         else      // delete last node
            p.Blink.Flink ← null
    }
```

We'll end this section on implementing doubly linked lists by writing a method that prints the elements of a linked list in **reverse order**. In a singularly linked list, this could be somewhat difficult, but with a doubly linked list, the method is easy to write. First, we need a method that finds the **last node** in the list. This is just a matter of following each node's forward link until we reach a link that points to null. This method, called **FindLast**, is defined as follows:

The Pseudocode FindLast()

```
  Node current ← Head
     while(!(current.Flink Equal  null))
     current ← current.Flink
   return current
```

Once we find the last node in the list, to print the list in reverse order we just follow the backward link until we get to a link that points to null, which indicates we're at the header node.

The Pseudocode PrintReverse

```
if  IsEmpty()
    print linked list is empty
else
  {
      Node current ← FindLast()
      while (current  not  equal null))
      {
          Print (current.data)
          current ← current.Blink
      }
}
```

2-The Circularly Linked List

A circularly linked list is a list where the last node points back to the first node (which may be a header node). Figure below illustrates how a circularly linked list works.


```
public class Node
{
    public Object data;
    public Node next;
    public Node()
      {
           data = null;
           next = null;
     }
    public Node(Object item)
    {
           data = item;
           next= null;
     }
}
public class LinkedList
{
      private  Node current;
      private Node Head ;
      public LinkedList()
      {
          Head= null;
   }
```

The Pseudocode IsEmpty ()

```
    if (Head equal  null)
        return true
else
      return false
```

The Pseudocode MakeEmpty()

```
Head ← null
```

The Pseudocode PrintList()

```
Node current ←  Head
      while ( (current.next not equal Head))
{
            Print (current. data)
            current ← current.next
      }
   Print (current.data)
```

The Pseudocode MoveHead(int n)

```
Node current ←Head
for(int i ← 0; i < n; i++)
current ← current.next
Head ← current
```

Stack and Queues using Linked Structures

Implementing stacks or Queues using arrays

- Simple implementation.
- The size of the stack or (queues) must be determined when a stack object is declared.
- Space is wasted if we use less elements.
- We cannot "push" or "enqueue" more elements than the array can hold.

Implementing stacks using Linked List

- Allocate memory for each new element dynamically .
- Use one pointer, *Top (Head)* to mark the top (last element) of the stack.
- Each node in the stack should contain two parts:
  - data: the user's data.
  - next: the address of the next element in the stack .

Class **SLinkedList** specification :

SLinkedList	
-Top:Node	**similar -Head: Node**
+SLinkedList()	
+ Push(object):void	**similar +AddAtFront(object):void**
+ PoP() :void	**similar +DelFront():void**
+Peek(): object	
+IsEmpty():bool	**similar +IsEmpty():bool**
+Display():void	**similar +Display():bool**
+Size():int	**similar +Size():int**

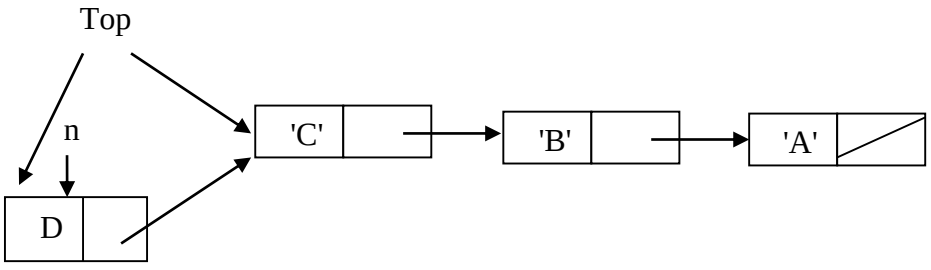
```
class SLinkedList
   {
// data member  or data value
   private  Node Top;

// Constructer
        public SLinkedList()
         {
           Top=null;
        }
```

StackLinkList Operations

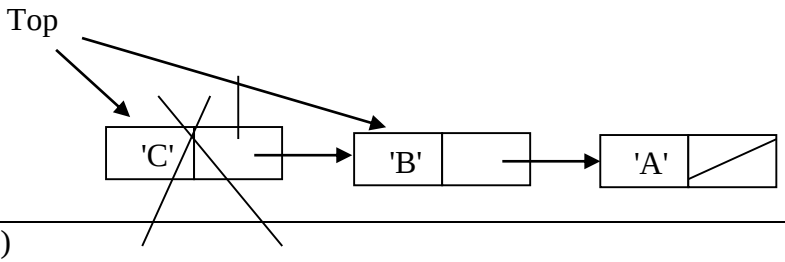
We describe how to use this method to implement a stack Linked list in code :

Pseudocode Push (object item) similar Pseudocode AddAtFront(object item)



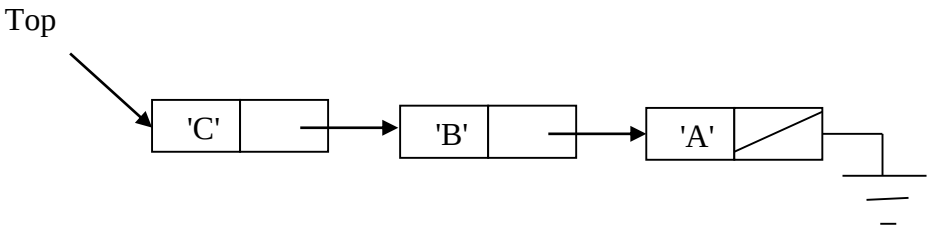
```
Node n = new Node(item)
    n.next← Top
    Top←n
```

Pseudocode_Pop() similar Pseudocode_DelFront()




```
if  IsEmpty()
    print stacklinked list empty
else
   Top← Top.next
```

Pseudocode_Peek()




```
if  IsEmpty()
    print stacklinked list empty
else
   item ← Top.data      // return (Top.data)
```

Pseudocode_IsEmpty()

```
  if  (Top equal null )
              return true
  else
            return false
```

Pseudocode Size()

```
if IsEmpty
        count←0
else
  {
     Node p← Top
      while ( p not equal null )
        {
            count= coun+1
            p=p.next
        }
  }
Return count
```

Pseudocode Display()

```
if IsEmpty()
        print stack empty
else
  {
     Node p← Top
      while ( p not equal null )
        {
            Print   p.data
            p←p.next
        }
  }
```

Implementing queues using linked list

- Allocate memory for each new element dynamically .
- Use two pointers, *qFront* and *qRear*, to mark the front and rear of the queue
- Each node in the Queue should contain two parts:
    - data: the user's data
    - next: the address of the next element in the stack

Class **QLinkedList** specification :

QLinkedList
-qFront:Node **-qRear:Node**
+QLinkedList() **+ Enqueue(object):void** **+ Dequeue():void** **+ Front():object** **+IsEmpty():bool** **+Display():void** **+Size():int**

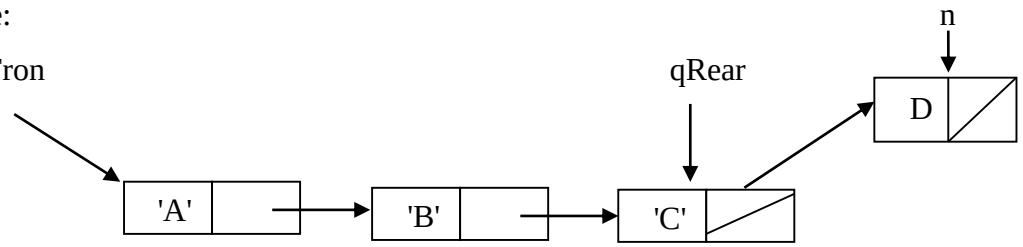
```
class QLinkedList
   {
// data member  or data value
   private  Node qFront;
    private  Node qRear;
// Constructer
        public QLinkList()
         {
           qFront =null;
           qRear =null;
        }
```

QLinkList Operations

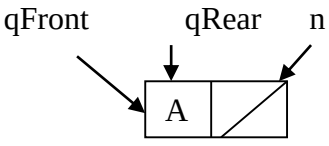
We describe how to use this method to implement a QLinked list in code :

Pseudocode enqueue(object item)

Example:

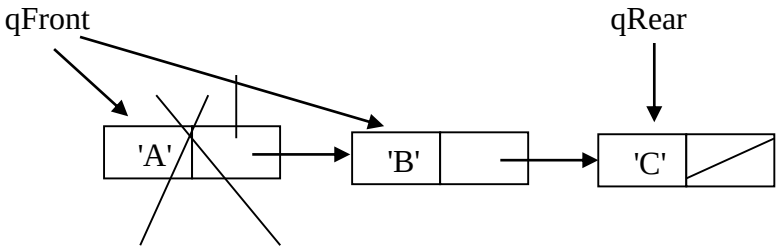


IsEmpty



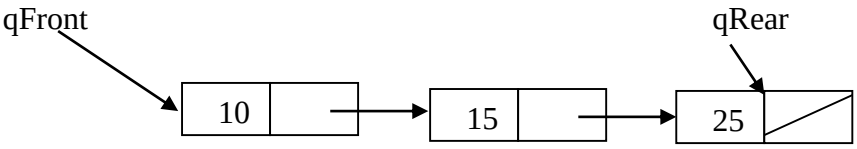
```
Node n = new Node(item)
if   IsEmpty()
     {
          n.next← null
          qFront← n
          qRear← n
      }
  else
     {
        qRear.next← n
        q.Rear←  n
       }
```

Pseudocode_dequeue()




```
if   IsEmpty()
     print Queuelinked list empty
else
        if (q.Rear Equal  q.Front)    // one node
                {
                    q.Rear←null
                     q.Front←null
                 }
         else
             qFront← qFront.next;
```

Pseudocode_Front()



```
  if   IsEmpty()
    print Queuelinked list empty
else
  return qFront.data;
```

Pseudocode_IsEmpty()

```
  if  (qFront  equal null  )
                return true
  else
            return false
```

Pseudocode_Size()

```
  if  IsEmpty()
            count←0
   else
     {
         Node p← qFront
          while ( p not equal  null )
             {
                  count← coun+1
                  p←p.next
             }
    }
 Return count
```

Pseudocode_Display()

```
  if  IsEmpty()
            print Queuelink list empty
   else
     {
         Node p← qFront;
          while ( p not equal  null )
             {
                  Print    p.data
                  p←p.next
             }
    }
```

The table show the running times of methods in realization of a linked list:

Method	Worst case	Best case	Average case
AddAtEnd(object)	O(n)	O(1)	O(n)
AddAtFront(object)	O(1)	O(1)	O(1)
DelFront()	O(1)	O(1)	O(1)
DelEnd()	O(n)	O(1)	O(n)
Insert(object,object)	O(n) if position of element end of the list .	O(1) if position of element in the first of the list.	O(log n) If the position of element in the middle of the list
Delete(object)	O(n) if position of element end of the list .	O(1) if position of element in the first of the list.	O(log n) If the position of element in the middle of the list
IsEmpty	O(1)	O(1)	O(1)
Display()	O(n)	O(n)	O(n)
Size()	O(n)	O(n)	O(n)

Comparisons :

1-Comparision between Single Linked List and Circular Linked List:

Single Linked List	Circular Linked List
1-Searching must begin from the beginning of the list .	1-Searching can begin from the middle of list .
2- Last node points to Null.	2-last node points to first node .

2- Comparison between Single Linked List and Double Linked List:

Single Linked List	Double Linked List
1-Searching is in one direction (forward) .	1-Searching can be done in two directions (forward and backward) .
2- each node has one pointer that point to the successor (Next) .	2- Each node has two pointers : one points to the successor (Flink) and one points to the predecessor (Blink) .
3-Takes less storage.	3- Takes more storage .

Exercises :

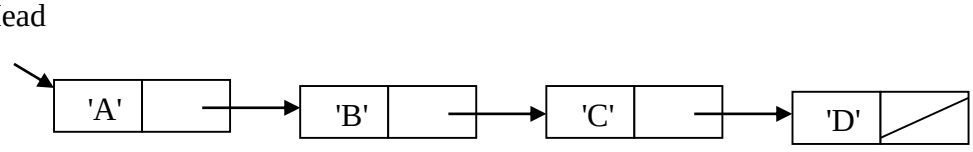
Q1 Draw the LinkedList resulted from executing the following code:

```
Node N1= new Node();
Node N2 = new Node(70);
N1.next =  new Node(10);
N1.next.next =  N2;
N1.data= N1.next.data;
N2.next=N1;
```

Q2:/ Complete the following method where needed?

```
int  sum   List()
 {
    int sum=0;
   Node  p=                ;
   while ( p       null)
     {
        sum=sum+ p.        ;
        p=                 ;
     }
  return    sum
}
```

Q3/ Redraw the following list after executing the piece of code shown underneath?




```
Node p=Head;
while(p.next.next !=null)   p=p.next;
p.next=Head;
Head=p;
```

Q4/ choose the most suitable answer:

1-Consider a "LinkedList " implementation of the Queue class , where from dose the "dequeue" method remove the element on the linked list?

a-from the head of the linked list .

b-from the entry following the head of the linked list

c-from the entry priority the end of the linked list.

d- from the end of the linked list

2- one or more statements is true about linked list:

a-Linked List is better used when the amount of data items to be stored is not Known until running.

b-Linked List is better when the amount of data items to be stored is Known in advance.

c-Storage allocation for linked list is done at running time.

d- Storage allocation for linked list is done at compile time.

3-In linked List implementation of general list , which operations require **linear time** for their worst case behavior?

a- DelEnd() b-IsEmpty() c- AddAtFront(object) d-Non of these operation require Linear Time

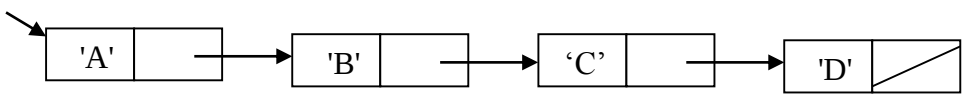
4-which structure is faster if you want to insert a new integer between the first and the second integer in the sequence?

a-linkedlist b- Array

Q5/Define a new method of Linked List class: void ListIncrement() that increment (add one to) List entries?

Q6/ Given the following list:

Head



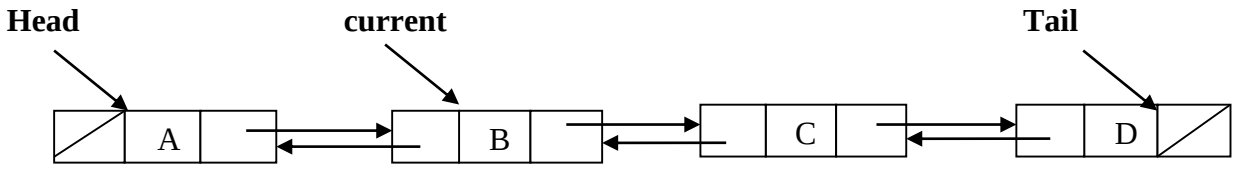
Write c# methods to:

a- Delete first element from it.

b- Add one element after data value C.

c- Display all the elements.

Q7/Which of the following correctly describe(s) the steps to delete element C from the below doubly Linked list



1-Current.Flink.Flink.Blink=current.Blink; current.Flink=current.Flink.Flink;

2-Current.Flink=current.Blink.Flink; current.Flink.Flink.Blink=current.Blink;

3-Current.Flink=current.Flink.Flink; current.Flink.Flink.Blink=current.Flink.Blink;

4-Current.Flink=Tail; Head.Flink=current.Flink.Blink;

5- Current.Flink=Tail ; Tail.Blink=Current;

Q8/Write method to delete a last node from double linked list?

Q9/ Compare between sequential and dynamic allocation of storage.

Q10/ Define a new method of Linked List class: int Count() to count the number of odd value elements in any linked list.