Data Structures Using C#

Presented By : Ahmed Wa'el Abu Fares

Topics

- Introduction
- ArrayList
- Generic Methods and Generic Class
- Dynamic and Implicitly Typed Local Variables
- Stack
- Queue
- Generic List
- Recursion
- Linked List
- Double Linked List
- Hash Table
- Tree

Section One

Introduction

INTRODUCTION

- In computer science, a data structure is a particular way of organizing data in a computer so that it can be used efficiently.

- Different kinds of data structures are suited to different kinds of applications, and some are highly specialized to specific tasks. For example, relational databases commonly use B-tree indexes for data retrieval, while compiler implementations usually use hash tables to look up identifiers.

- Data structures are generally based on the ability of a computer to **fetch** and **store** data at any place in its memory

INTRODUCTION

- The study of data structures and algorithms is critical to the development of the professional programmer.

COLLECTIONS

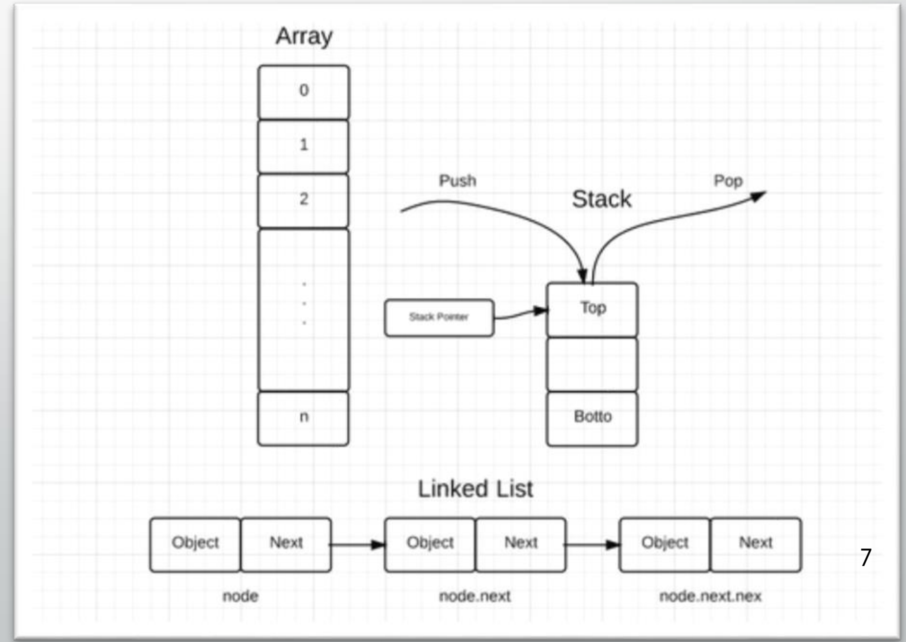
- A collection is a structured data type that stores data and provides operations for adding data to the collection, removing data from the collection, updating data in the collection, as well as operations for setting and returning the values of different attributes of the collection. Collections can be broken down into two types:

  [1] linear

  [2] nonlinear

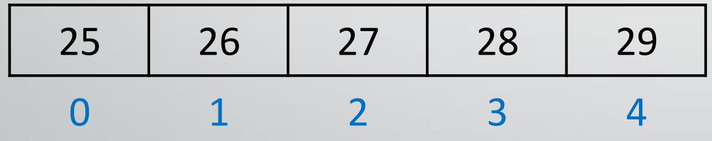
LINEAR COLLECTION (1)

- linear collection is a list of elements where one element follows the previous element. Elements in a linear collection are normally ordered by position (first, second, third, etc.). In the real world, a grocery list is a good example of a linear collection; in the computer world (which is also real), an array is designed as a linear collection



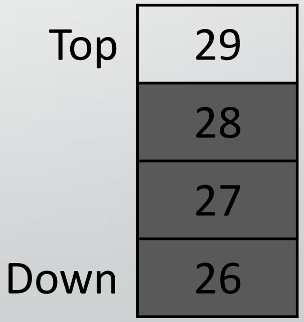
LINEAR COLLECTION (2)

- Linear collections can be either **direct access** collections or **sequential access** collections



25	26	27	28	29
0	1	2	3	4

Array [Direct Access]



Top 29

28

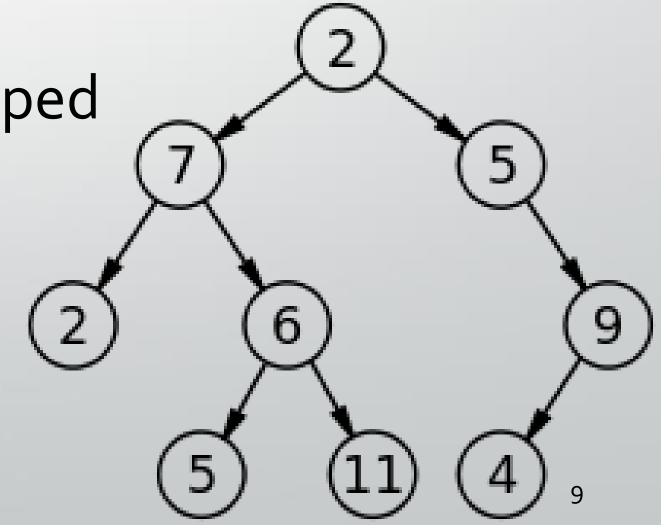
27

Down 26

Stack [Sequential Access]

NONLINEAR COLLECTION

- Nonlinear collections hold elements that do not have positional order within the collection. An organizational chart is an example of a nonlinear collection, as is a rack of billiard balls. In the computer world, trees, heaps, graphs, and sets are nonlinear collections.
- nonlinear collections can be either hierarchical or grouped



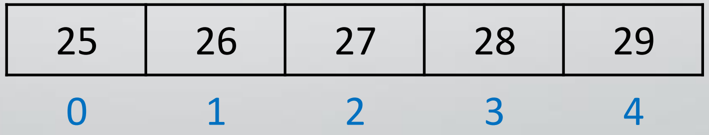
Section Two

Array List

Part One
ArrayList Implementation

Array

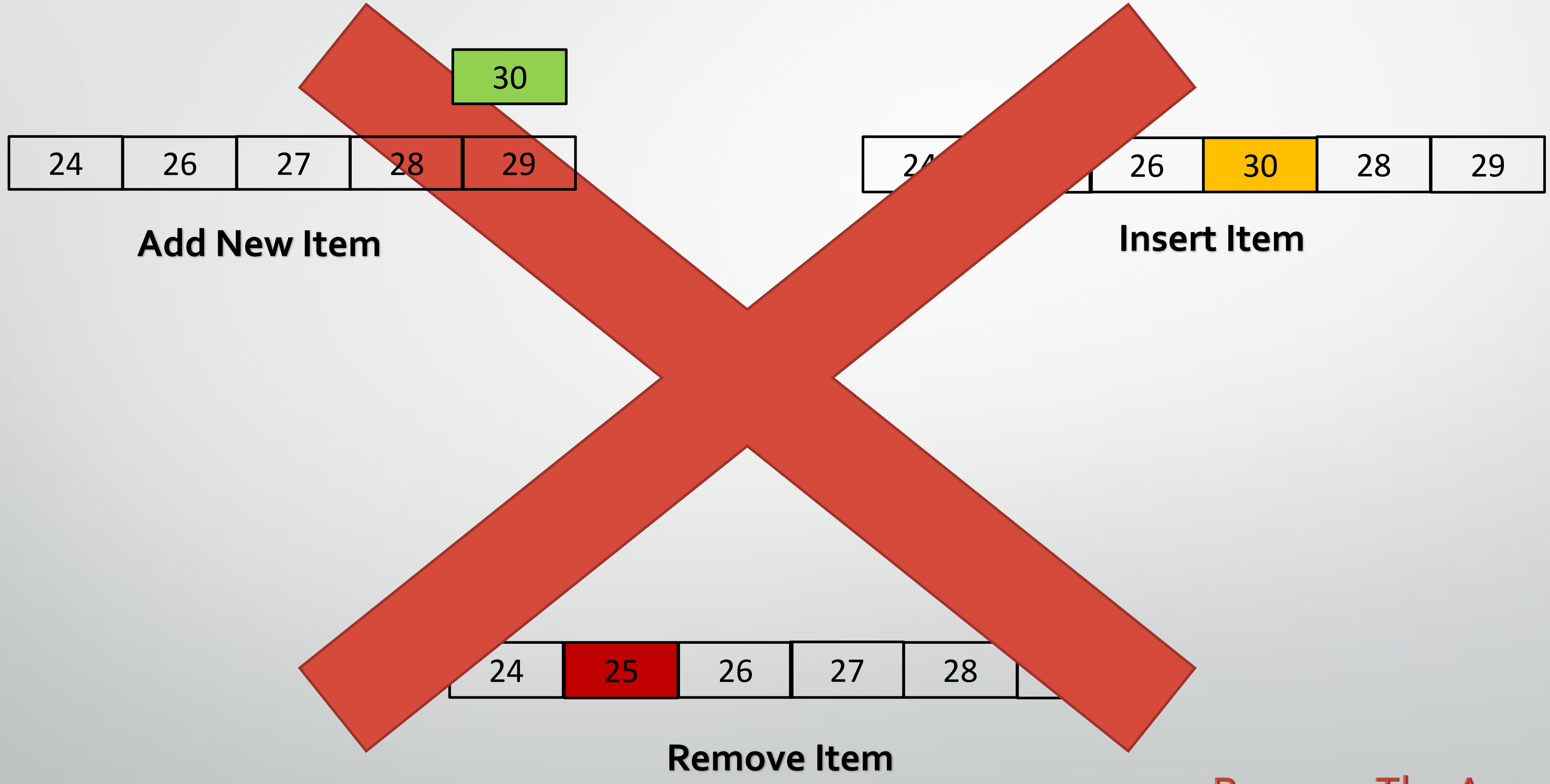
- it is a data structure with a fixed size , contains a defined number of values ,it is built up in any programming language to reserve a sequential data memory places , this data structure can be one or many dimensions .
- You can access any element of array using index , the index based zero
- The problem of use the array is fixed size , and not fixable



25	26	27	28	29
0	1	2	3	4

Array

Array



30

24	26	27	28	29

Add New Item

24		26	30	28	29

Insert Item

24	25	26	27	28	

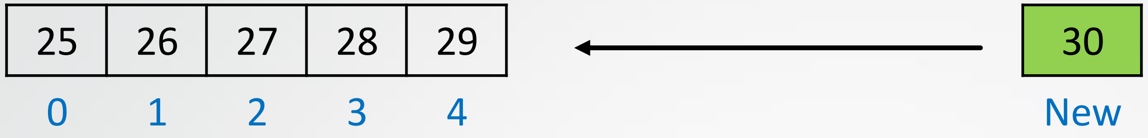
Remove Item

Because The Array Is Fixed

ArrayList

- arrays are not very useful when the size of an array is unknown or change during the lifetime of a program. One solution to this problem is to use a type of array that automatically resizes itself when the array is out of storage space. This array is called an ArrayList and it is part of the System.Collections namespace in the .NET Framework library.

ArrayList implementation [Add]



25	26	27	28	29		30
0	1	2	3	4		New

Step[1] : crate a new array of size the original array size + 1

0	1	2	3	4	5

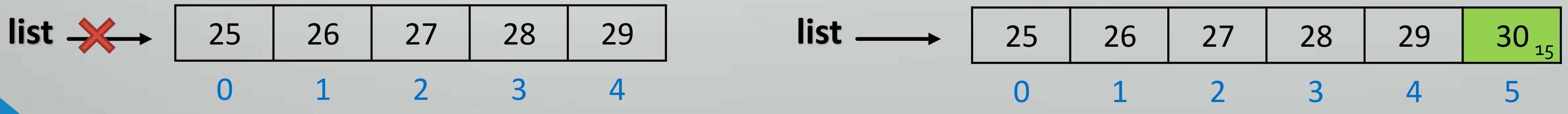
Step[2] : copy the value of original array to a new array

25	26	27	28	29	
0	1	2	3	4	5

Step[3] : add new item in last element of new array

25	26	27	28	29	30
0	1	2	3	4	5

Step[4] : make reference variable pointing to new array



list

25	26	27	28	29
0	1	2	3	4

list

25	26	27	28	29	30
0	1	2	3	4	5

ArrayList implementation [Add]

```
public class CArrayList
{
 object [] list ;
 public CArrayList()
        {
             list = new object[0];
        }
 public int Add(object item)
        {
            object[] newArray = new object[list.Length + 1];
            for (int i = 0; i < list.Length; i++)
                newArray[i] = list[i];
            newArray[list.Length] = item;
            list = newArray;
            return list.Length-1;
        }
```

ArrayList implementation [Insert]

25	26	27	30	28	29
0	1	2	3	3	4

Step[1] : crate a new array of size the original array size + 1

0	1	2	3	4	5

Step[2] : put item in index

			30		
0	1	2	3	4	5

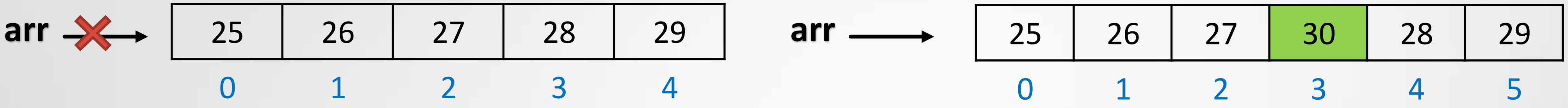
Step[3] : copy item from original array as sorted if the element is null , else put the next element

25	26	27	28	29		Original Array
			30			New Array
0	1	2	3	4	5	

```
if (new[i] == null)
    {
        new[i] = list[i]
    }
else
    {
        new[i + 1] = list[i];
    }
```

ArrayList implementation [Insert]

Step[4] : make reference variable pointing to new array



arr

25	26	27	28	29
0	1	2	3	4

arr

25	26	27	30	28	29
0	1	2	3	4	5

```
public void Insert (int index, object item)
        {
            object[] newArray = new object[list.Length + 1];
            newArray[index] = item;
            for (int i = 0; i < list.Length; i++)
            {
                if (newArray[i] == null)
                    newArray[i] = list[i];
                else
                    newArray[i + 1] = list[i];
            }
            list = newArray;
        }
```

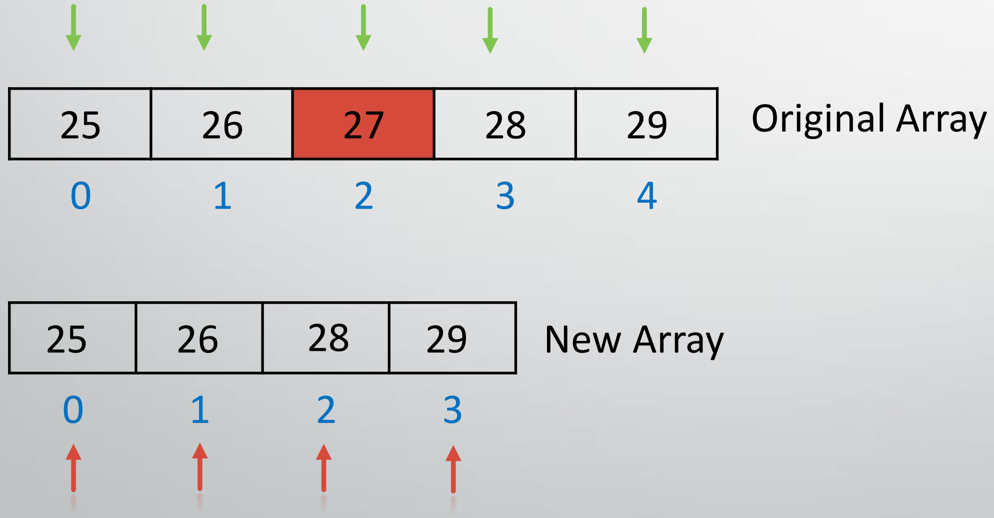
ArrayList implementation [Remove]

25	26	27	28	29
0	1	2	3	4

Step[1] : crate a new array of size the original array size -1

0	1	2	3

Step[2] : copy all element to new array except the item that you want to delete



Original Array

25	26	27	28	29
0	1	2	3	4

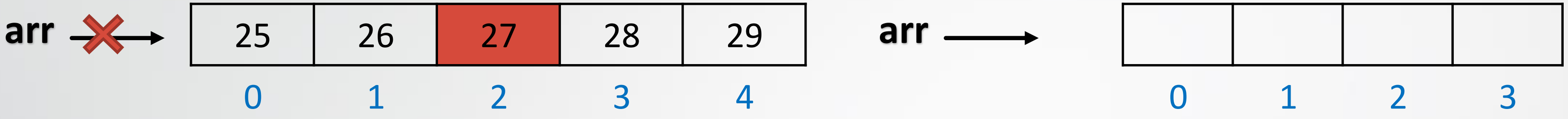
New Array

25	26	28	29
0	1	2	3

```
for (int i = 0; i < list.Length; i++)
        {
           if (i != index)
           {
              temp[n] = list[i];
              ++n;
           }
        }
// index : index for element wants to delete
// n: index of new array
// i: index of original array
```

ArrayList implementation [Remove]

Step[4] : make reference variable pointing to new array



arr

25	26	27	28	29
0	1	2	3	4

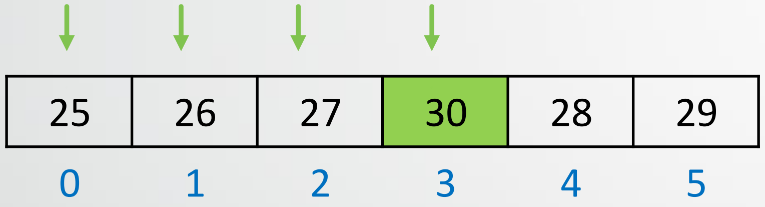
arr

0	1	2	3

```
public void RemoveAt(int index)
       {
           int nIndex= 0;
           object[] newArray = new object[list.Length - 1];
           for (int i = 0; i < list.Length; i++)
           {
               if (i != index)
               {
                   newArray[nIndex] = list[i];
                   ++nIndex;
               }
           }
           list = newArray;
       }
```

ArrayList implementation [Get Item]

```
public object GetItemAt(int index)
       {
           return list[index];
       }
```

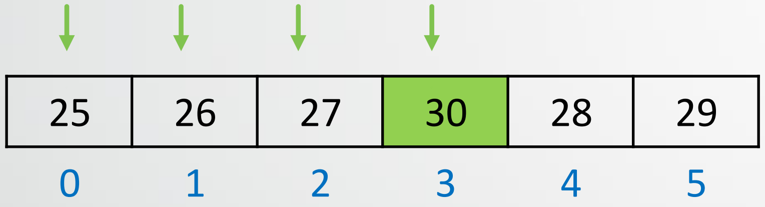


25	26	27	30	28	29
0	1	2	3	4	5

```
public int IndexOf(object item)
       {
           for (int i = 0; i < list.Length; i++)
               if (list[i].ToString().Equals(item.ToString()))
                   return i;
           return -1;
      {
```

ArrayList implementation [Get Item]

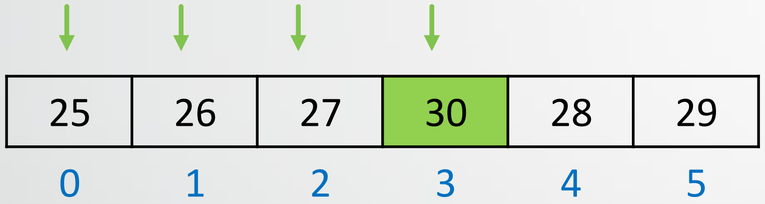
```
public void ItemAt(int index, object value)
       {
           list[index] = value;
       }
```



25	26	27	30	28	29
0	1	2	3	4	5

```
public int Length
       {
          get { return list.Length;}
       {
```

ArrayList implementation [Get Item]



25	26	27	30	28	29
0	1	2	3	4	5

ArrayList implementation [Get Item]

You can re-implement the ItemAt() method and GetItamAt() method using property and indexer to make the ArrayList Like array as access the elements and change value of elements

```
public object this[int index]
        {
            get { return list[index]; }
            set { list[index] = value; }
        }
```

Example :

```
int[] arr = { 1, 2, 3 };
Console.WriteLine(arr[0]);
arr[0] = 9;
Console.WriteLine(arr[0]);
CArrayList list = new CArrayList();
list.Add(5);
Console.WriteLine(list[0]);
list[0] = 9;
Console.WriteLine(list[0]);
```

Part Two
Built in - ArrayList

	Member	Description
+	ArrayList()	Initializes a new instance of the ArrayList class that is empty and has the default initial capacity.
+	ArrayList(ICollection)	Initializes a new instance of ArrayList class that contains elements copied from the specified collection and that has the same ,initial capacity as the number of elements copied.
+	ArrayList(int)	Initializes a new instance of the ArrayList class that is empty and has the specified initial capacity.
+	Capacity : int	Gets or sets the number of elements that the ArrayList can contain.
+	Count : int	Gets the number of elements actually contained in the ArrayList
+	Add(object) : int	Adds an object to the end of ArrayList,return index at which the value has been added.
+	AddRange(ICollection) : void	Adds the elements of any collection to the end of ArrayList.
+	Clear() : void	Removes all elements from.
+	Contains(object) : bool	Determines whether an element is in the ArrayList.
+	CopyTo(Array) : void	Copies the entire ArrayList to a compatible one-dimensional array starting at the beginning of the target array.
+	CopyTo (Array, int) : void	Copies the entire ArrayList to a compatible one-dimensional array, starting at the specified index of the target array.
+	CopyTo(int, Array, int, int) : void	Copies a range of elements from the ArrayList to a compatible one-dimensional array, starting at the specified index of the target array.

	Member	Description
+	GetRange(int, int) : ArrayList	Returns an ArrayList which represents a subset of the elements in the source ArrayList.
+	IndexOf(object) : int	Searches for the specified item and returns the zero-based index of the first occurrence within the entire ArrayList.
+	IndexOf(object, int) : int	Searches for the specified item and returns the zero-based index of the first occurrence within the range of elements in ArrayList that extends from the specified index to the last element.
+	IndexOf(object, int,int) : int	Searches for the specified item and returns the zero-based index of the first occurrence within the range of elements in the ArrayList that starts at the specified index and contains the specified number of elements.
+	Insert(int,object) : void	Inserts an element into the ArrayList at the specified index.
+	InsertRange(int, ICollection) : void	Inserts the elements of a collection into the ArrayList at the specified index.
+	LastIndexOf(object) : int	Searches for the specified item and returns the zero-based index of the last occurrence within the entire ArrayList.
+	LastIndexOf(object, int) : int	Searches for the specified item and returns the zero-based index of the last occurrence within the range of elements in the ArrayList that extends from the first element to the specified index.
+	LastIndexOf(object, int, int) : int	Searches for the specified item and returns the zero-based index of the last occurrence within the range of elements in the ArrayList that contains the specified number of elements and ends at the specified index.

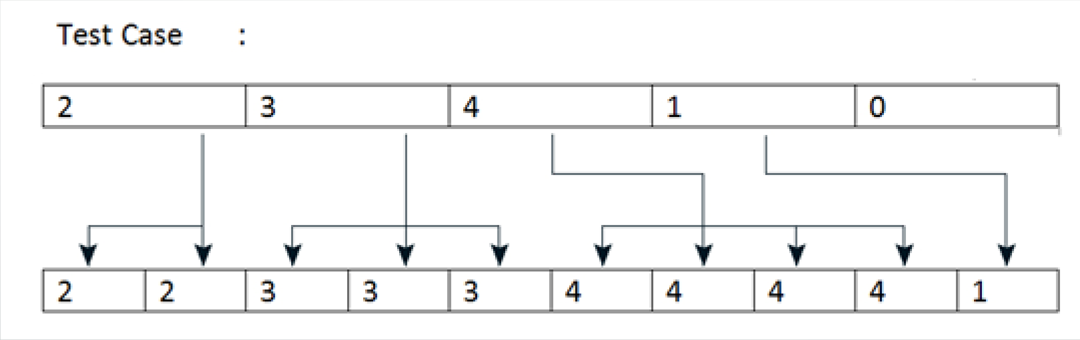
	Member	Description
+	Remove(object) : void	Removes the first occurrence of a specific object from the ArrayList.
+	RemoveAt(int) : void	Removes the element at the specified index of the ArrayList.
+	RemoveRange(int, int) : void	Removes a range of elements from the ArrayList .
+	Reverse() : void	Reverses the order of the elements in the entire ArrayList.
+	Reverse(int, int) : void	Reverses the order of the elements in the specified range.
+	Sort() : void	Sorts the elements in the entire ArrayList.
+	ToArray() : object []	Copies the elements of the ArrayList to a new Object array.
+	TrimToSize() : void	Sets the capacity to the actual number of elements in the ArrayList.

Array list Application

[1] write a program to create array of element {2,3,4,1,0} and duplicate each element in Array to Array list .

[A] : Find sum of all elements in array list

[B] : make the array list as reverse order



Test Case :

2	3	4	1	0

2	2	3	3	3	4	4	4	4	1

Array list Application

[2] write a program to create class called Student , Student class contains ID as integer , name as string ,GPA as double .and Print Method to Print information about student .

[A] : create array list and ask the user how many add student

[B] : add student in array list using keyboard

[C] : create new object of Student (1234,"unkown",0.0) and insert it in

second position of array list

[c] : invoke the print for all student in array list

Array list Application

[3] Create two array lists "George",“Yara", “Ahmed", “Sara", “Khaled", “Ra’ed“ and "George", “Billal", “Khaled", “Eman", "Ryan" and find their union, difference, and intersection

- Union : George , Yara , Ahmed , Sara , Khaled , Ra’ed ,Billal,Eman,Rayan

- Intersection : George , Khaled

- Difference : Yara , Ahmed , Sara, Ra’ed

Section Three

Generic Class , Methods

Why Generic?

- Create an array list of 3 integer numbers and calculate sum of these numbers.
- To solve this problem you must use type casting from (object) to (int) or you can use generic class

Generic

- One of the problems with OOP is a feature called "code bloat." One type of code bloat occurs when you have set of methods that take more than one possible data types of the method's parameters .
  One solution to code bloat is the ability of one value to take on multiple data types, while only providing one definition of that value. This technique is called generic programming.
- A generic program provides a data type "placeholder" that is filled in by a specific data type at compile-time. This placeholder is represented by a pair of angle brackets (< >), with an identifier placed between the brackets
- To solve code bloat problem you can use polymorphism or generic technique

Generic

- Generics introduce to the .NET Framework the concept of type parameters, which make it possible to design classes and methods that defer the specification of one or more types until the class or method is declared and instantiated by client code. For example, by using a generic type parameter T you can write a single class that other client code can use without incurring the cost or risk of runtime casts or boxing operations.

Generic technique usage

- Generic technique can use in :

  [1] : Class : determined the data type when creating the object

  ```
  ArrayList <int> arr = new ArrayList<int>();
  ```

  [2] : Methods : determined the data type when calling the method

  ```
  Swap<int>(x,y);
  ```

- Re-write the array list implementation using generic technique

Generic Methods

- Write a static method to swap two variables , the variables can integer, double , char , bool , string without using polymorphism .
- To write this method you must re write the implantation of method for all data type (overloading)
- But when using generic method you can write one method for all data type.

Swap Method

```
static void Swap(ref int x,ref int y)
  {
      int temp = y;
      y = x;
      x = temp;
  }
```

```
static void Swap(ref double x,ref double y)
  {
      double temp = y;
      y = x;
      x = temp;
  }
```

```
static void Swap(ref char x,ref char y)
  {
      char temp = y;
      y = x;
      x = temp;
  }
```

```
static void Swap(ref string x,ref string y)
  {
      string temp = y;
      y = x;
      x = temp;
  }
```

......

Generic Swap Method

```
static void Swap<t>(ref t x,ref t y)
  {
     t temp = y;
     y = x;
     x = temp;
  }
 static void Main(string[] args)

  {

     int var1 = 12,var2=13;

     double var3 = 9.5,var4 = 7.3;

     Swap<int>(ref var1, ref var2);

     Swap<double>(ref var3, ref var4);

  }
```

Applications

[1] write a generic method *IsEqual()* that take two parameter of any data type (bool, double , float , int , string , char) if the value of first parameter equal second parameter return true otherwise return false

Hint : use the Equals() method for Comparison , don't use (==) for Comparison

[2] write a generic method Sum() that calculate any tow numbers and return the summation

Note : You can't use the operator with any variable of type generic

Section Four

Dynamic , Var Type

Part One
Dynamic Type

DYNAMIC TYPE

- The dynamic keyword is new to C# 4.0, and is used to tell the compiler that a variable's type can change or that it is not known until runtime. Think of it as being able to interact with an Object without having to cast it.
- To declare dynamic variable use the *dynamic* keyword

```
dynamic dyn = 1;
object obj = 1;
Console.WriteLine(obj*6); // compile error
Console.WriteLine(dyn * 6);
```

- When using dynamic variable you don’t need use casting type
- You can use dynamic variable as parameter , instance or local

DYNAMIC TYPE

```
static void Main(string[] args)
{
     dynamic dyn ;
  // dyn is int
      dyn = 5;
 // dyn is string
      dyn = "Hello";
 // dyn is [] int
      dyn = new[] { 0, 1, 2 };
 // dyn is DateTime
      dyn = System.DateTime.Now;
}
```

Part Two
Implicitly Typed Local Variables

Implicitly Typed Local Variables

- Local variables can be given an inferred "type" of var instead of an explicit type. The var keyword instructs the compiler to infer the type of the variable from the expression on the right side of the initialization statement. The inferred type may be a built-in type, an anonymous type, a user-defined type, or a type defined in the .NET Framework class library.
- You can use the var type only within method (local) , can’t can parameter or instance variable
- Example :

```
int x = 5;
var y = 5;
```

Implicitly Typed Local Variables

```
static void Main(string[] args)
{
  // i is compiled as an int
      var i = 5;
  // s is compiled as a string
      var s = "Hello";
  // a is compiled as int[]
      var a = new[] { 0, 1, 2 };
  // d is compiled as DateTime
      var d = System.DateTime.Now;
}
```

Section Five

Generic list

Generic List

- The generic List<T> class is the simplest of the collection classes. You can use it much like an array you can reference an existing element in a List<T> collection by using ordinary array notation, with square brackets and the index of the element, although you cannot use array notation to add new elements. However, in general, the List<T> class provides more flexibility than arrays and is designed to overcome the following restrictions exhibited by arrays :

  [1] If you want to resize an array

  [2] If you want to remove an element from an array
  [3] If you want to insert an element into an array

  [4] if you want to use generic technique

	Member	Description
+	List<T>()	Initializes a new instance of the List<T> class that is empty and has the default initial capacity.
+	List<T>(ICollection)	Initializes a new instance of List<T> class that contains elements copied from the specified collection and that has the same ,initial capacity as the number of elements copied.
+	List<T>(int)	Initializes a new instance of the List<T> class that is empty and has the specified initial capacity.
+	Capacity : int	Gets or sets the number of elements that the List<T> can contain.
+	Count : int	Gets the number of elements actually contained in the List<T>
+	Add(T) : void	Adds an object to the end of List<T>,return index at which the value has been added.
+	AddRange(ICollection) : void	Adds the elements of any collection to the end of List<T>.
+	Clear() : void	Removes all elements from.
+	Contains(T) : bool	Determines whether an element is in the List<T>.
+	CopyTo(Array) : void	Copies the entire List<T> to a compatible one-dimensional array starting at the beginning of the target array.
+	CopyTo (Array, int) : void	Copies the entire List<T> to a compatible one-dimensional array, starting at the specified index of the target array.
+	CopyTo(int, Array, int, int) : void	Copies a range of elements from the List<T> to a compatible one-dimensional array, starting at the specified index of the target array.

	Member	Description
+	GetRange(int, int) : List<T>	Returns an List<T> which represents a subset of the elements in the source List<T> .
+	IndexOf(T) : int	Searches for the specified item and returns the zero-based index of the first occurrence within the entire List<T>.
+	IndexOf(T, int) : int	Searches for the specified item and returns the zero-based index of the first occurrence within the range of elements in List<T> that extends from the specified index to the last element.
+	IndexOf(T, int,int) : int	Searches for the specified item and returns the zero-based index of the first occurrence within the range of elements in the List<T> that starts at the specified index and contains the specified number of elements.
+	Insert(int,T) : void	Inserts an element into the List<T> at the specified index.
+	InsertRange(int, ICollection) : void	Inserts the elements of a collection into the List<T> at the specified index.
+	LastIndexOf(T) : int	Searches for the specified item and returns the zero-based index of the last occurrence within the entire List<T>.
+	LastIndexOf(T, int) : int	Searches for the specified item and returns the zero-based index of the last occurrence within the range of elements in the List<T> that extends from the first element to the specified index.
+	LastIndexOf(T, int, int) : int	Searches for the specified item and returns the zero-based index of the last occurrence within the range of elements in the List<T> that contains the specified number of elements and ends at the specified index.

	Member	Description
+	Remove(object) : void	Removes the first occurrence of a specific object from the List<T>.
+	RemoveAt(int) : void	Removes the element at the specified index of the List<T>.
+	RemoveRange(int, int) : void	Removes a range of elements from the List<T>.
+	Reverse() : void	Reverses the order of the elements in the entire List<T>.
+	Reverse(int, int) : void	Reverses the order of the elements in the specified range.
+	Sort() : void	Sorts the elements in the entire List<T>.
+	ToArray() : T[]	Copies the elements of the List<T> to a new Object array.
+	TrimExcess: void	Sets the capacity to the actual number of elements in the List<T>.
+	BinarySearch(T) : int	Searches the entire sorted List<T> for an element using the default comparer and returns the zero-based index of the element.

Section Six

Stack

Stack

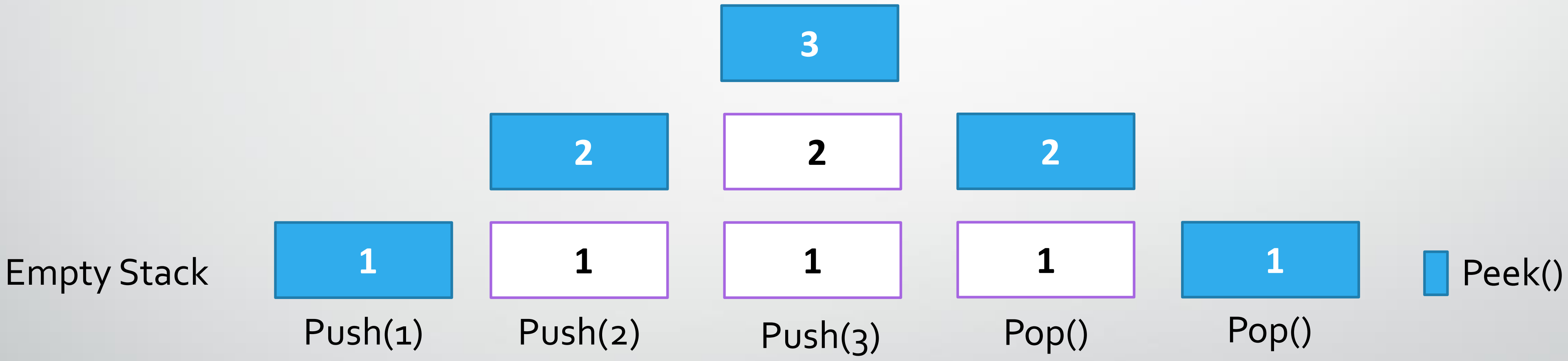
- The stack is one of the most frequently used data structures
- We define a stack as a list of items that are accessible only from the end of the list, which is called the top of the stack .
- always removed from the top of stack
- A stack is known as a Last-in, First-out (LIFO) data structure.
- Operators in Stack are :

  [1] Push : Inserts item at the top of the stack

  [2] Pop : Removes and returns the item at the top of the stack

  [3] Peek : Returns the item at the top of the stack without removing it.

Stack



Stack Operators

Stack Implantation

Stack Implantation

- You can build the stack using array list or array . Here we build the stack using array list
- You can use integer variable to know the index of last item in stack
- Operations :

[1] Push : Inserts item at the top of the stack

[2] Pop : Removes and returns the item at the top of the stack

[3] Peek : Returns the item at the top of the stack without removing it.

[4] Clear : Removes all objects from the stack.

[5] Count : Gets the number of elements contained in the stack

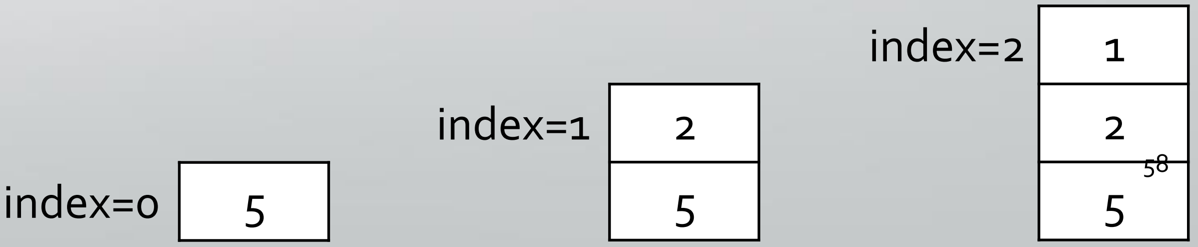
[6] ToArray : Copies the stack to a new array

Stack Implantation [Push]

Inserts item at the top of the stack

```
class BStack
    {
        private ArrayList list;
        private int index;
        public BStack()
        {
            list = new ArrayList();
            index = -1;
        }
        public void Push(object item)
        {
            list.Add(item);
            ++index;
        }
```

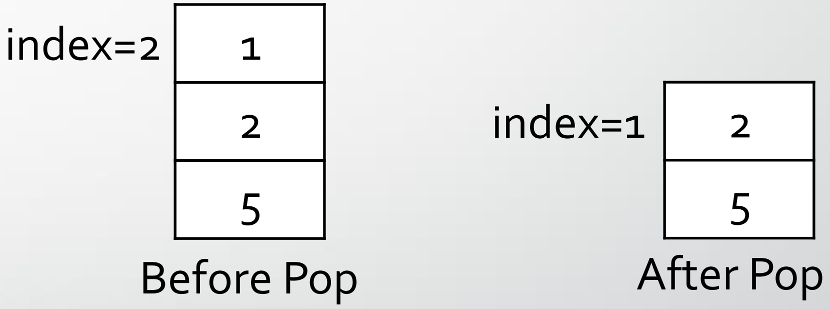
Empty Stack index=-1



Stack Implantation [Pop]

Removes and returns the item at the top of the stack

```
public object Pop()
 {
     object item = list[index];
     list.RemoveAt(index);
     --index;
     return item;
 }
```



index=2

1
2
5

Before Pop

index=1

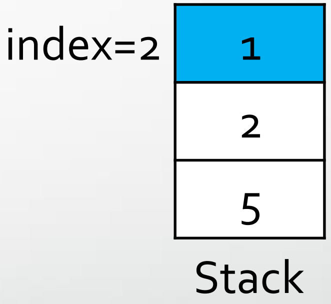
2
5

After Pop

Stack Implantation [Peek]

Returns the item at the top of the stack without removing it.

```
public object Peek()
 {
     return list[index];
 }
```



index=2

1

2

5

Stack

Stack Implantation

```
public object[] ToArray()
{
     ArrayList list2 = new ArrayList(list);
     list2.Reverse();
     return list2.ToArray();
}
public int Count
{
    get
    {
        return list.Count;
    }
}
public void Clear()
{
    list.Clear();
}
```

Stack Applications

- Write a program to input a string , and find if the string is palindrome or not

  Test case : “madam” , “dad” and “soos” are palindrome but “ahmed” is not palindrome

- Write a program to input string that contains only brakets “{}[]()” and check the string is a balanced or not

  Test case : “()” , “{{()[()]}}” and “{[()]}” are balanced but “{[)}”,”{{[]” is not balanced

- Write a program to make simple internet browser with three functions

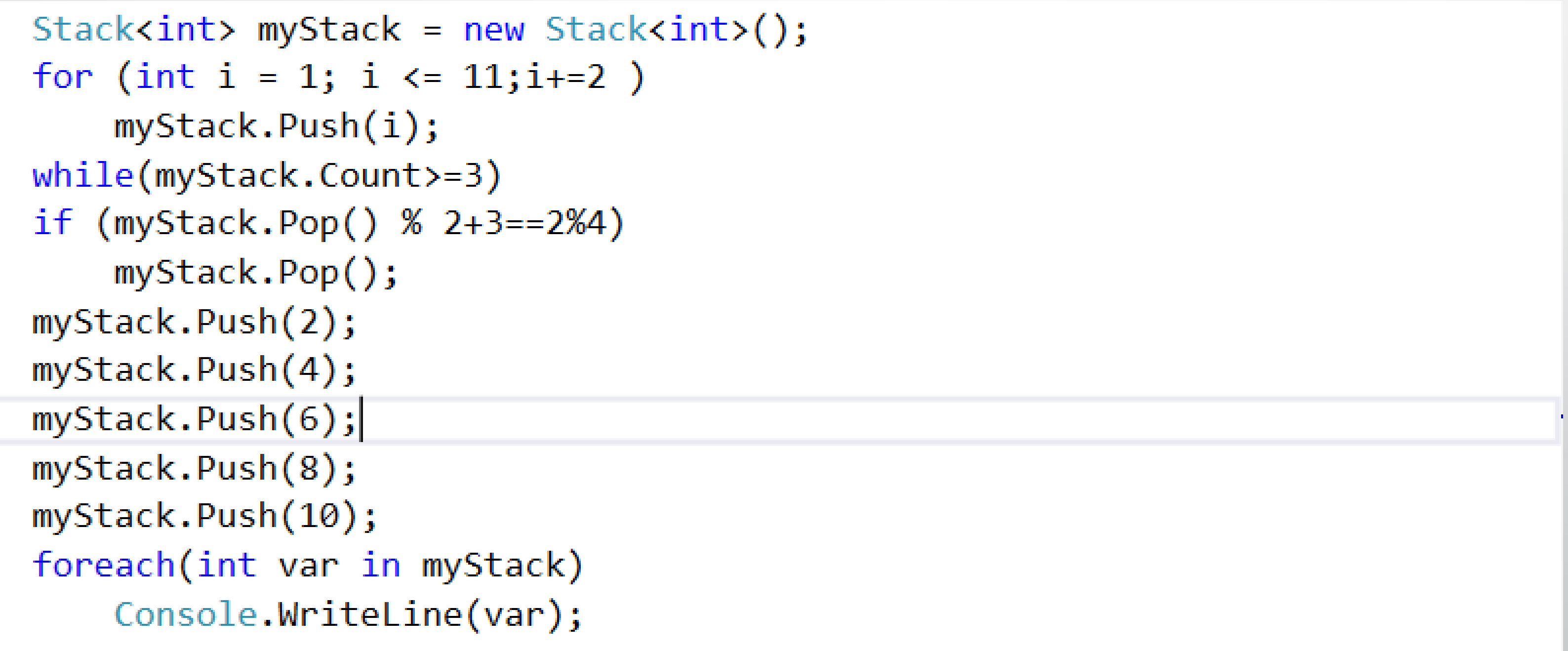
  [1] Go to website : allow user to enter website link and go to it

  [2] Back : to return the last website

  [3] forward : to go the next website

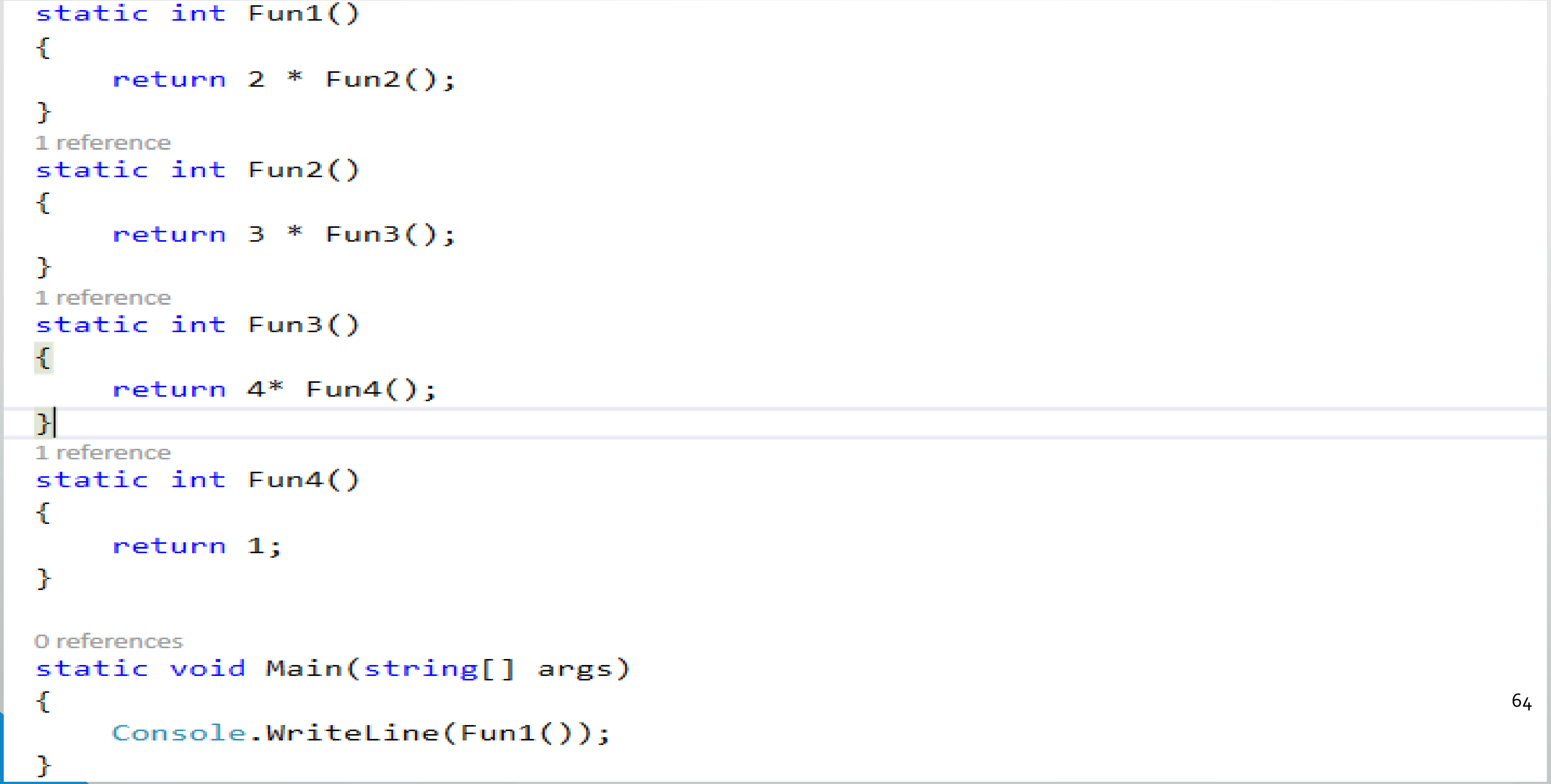
Stack Applications

- What is the output after execution the following code :



```
Stack<int> myStack = new Stack<int>();
for (int i = 1; i <= 11;i+=2 )
    myStack.Push(i);
while(myStack.Count>=3)
if (myStack.Pop() % 2+3==2%4)
    myStack.Pop();
myStack.Push(2);
myStack.Push(4);
myStack.Push(6);
myStack.Push(8);
myStack.Push(10);
foreach(int var in myStack)
    Console.WriteLine(var);
```

Stack Applications[Calling Methods]



```
static int Fun1()
{
    return 2 * Fun2();
}
1 reference
static int Fun2()
{
    return 3 * Fun3();
}
1 reference
static int Fun3()
{
    return 4* Fun4();
}
1 reference
static int Fun4()
{
    return 1;
}

0 references
static void Main(string[] args)
{
    Console.WriteLine(Fun1());
}
```

Stack Applications[infix to postfix]

- Infix expression : 5*2/(5+3)
- Postfix expression : 5 2 * 5 3 + /
- To Convert between infix to post fix use stack to push operators and show the operands into output
- When pushing the operators :
- [1] Push : if the operator has more priority
- [2] Pop : if the operator is low priority more than top of the stack

Stack Applications[infix to postfix]

- Example : 5*2/5+3
- Output :
- Postfix :

Stack Applications[infix to postfix]

- Example : 6/2-3+4*2
- Output :
- Postfix :

Stack Applications[infix to postfix]

- Example : 4/(2-1+6)*2+14
- Output :

- Postfix :

Stack Applications[infix to postfix]

- Example : 24+18*(5+2*4)^2
- Output :
- Postfix :

Section Seven

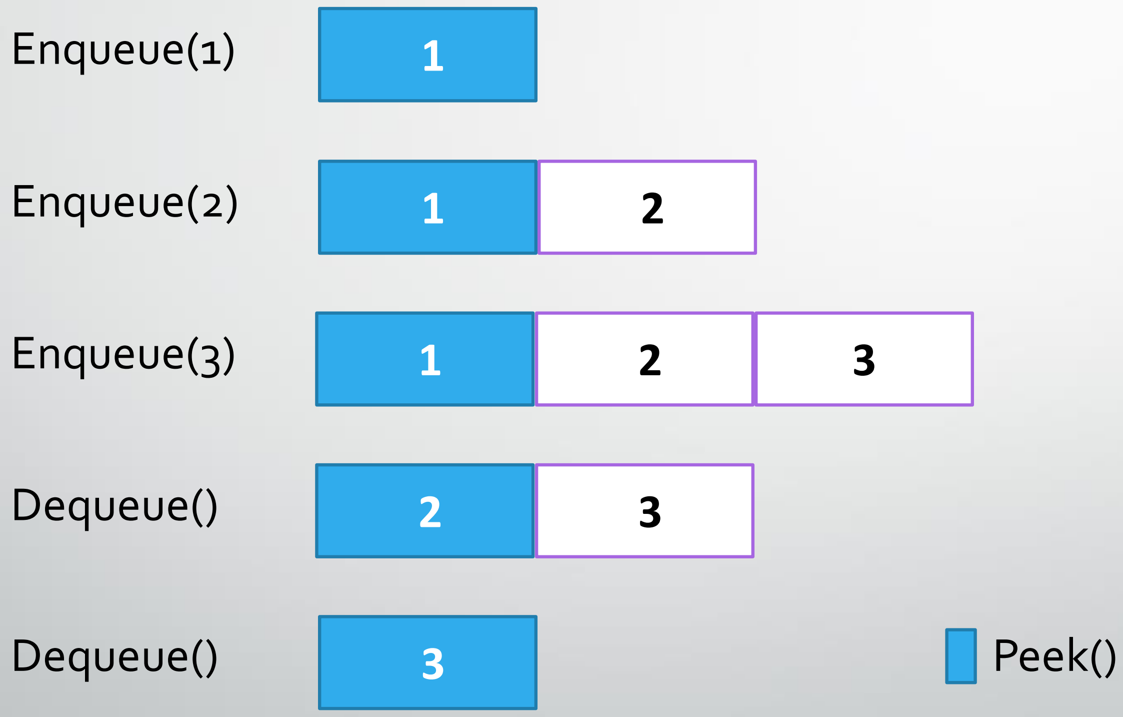
Queue

Queue

- A queue is a data structure where data enters at the rear of a list and is removed from the front of the list. Queues are used to store items in the order in which they occur. Queues are an example of a first-in, first-out (FIFO) data structure.
- The two primary operations involving queues are adding a new item to the queue and removing an item from the queue. The operation for adding a new item is called Enqueue, and the operation for removing an item from a queue is called Dequeue. The Enqueue operation adds an item at the end of the queue and the Dequeue operation removes an item from the front (or beginning) of the queue .
- The other primary operation to perform on a queue is viewing the beginning item is Peek method
- The queue is linear data structure and sequential access data structure

Queue

Empty Queue



Enqueue(1) 1

Enqueue(2) 1 2

Enqueue(3) 1 2 3

Dequeue() 2 3

Dequeue() 3

🟦 Peek()

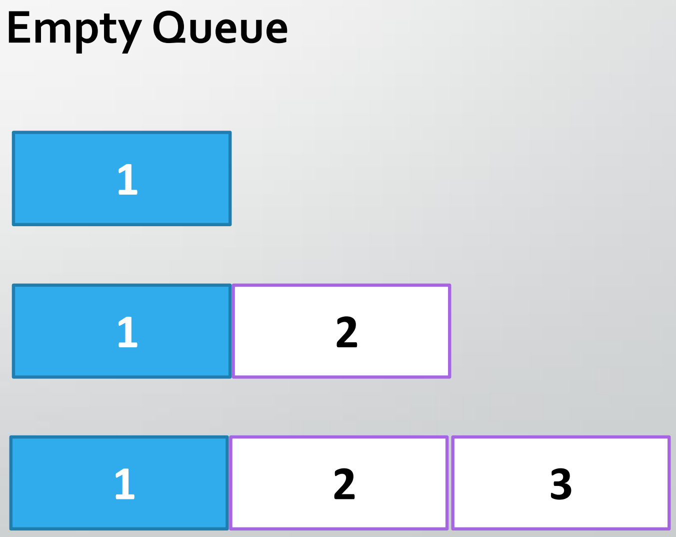
Queue Operators

Queue Implantation

Queue Implantation [Enqueue]

Adds an object to the end of the queue

```
class BQueue
    {
        private ArrayList list;
        public BQueue()
        {
            list = new ArrayList();
        }
        public void Enqueue(object item)
        {
            list.Add(item);
        }
```

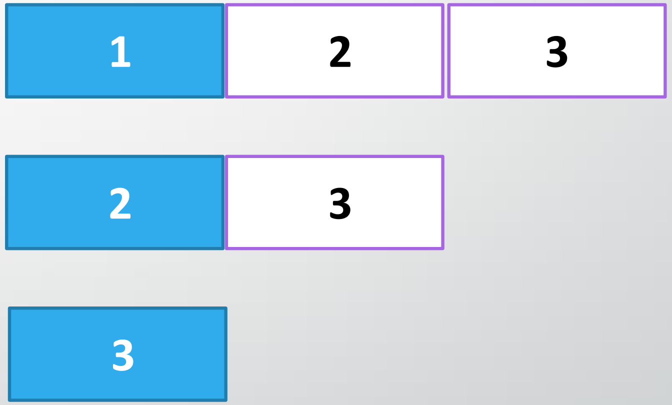


Empty Queue

Queue Implantation [Dequeue]

Removes and returns the object at the beginning of the queue

```
public object Dequeue()
{
    object item = list[0];
    list.RemoveAt(0);
    return item;
}
```



1 2 3

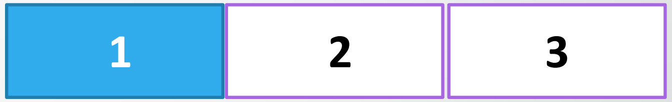
2 3

3

Queue Implantation [Peek]

Returns the object at the beginning of the queue without removing it

```
public object Peek()
{
      return list[0];
}
```



1 2 3

Queue Implantation

```
public object[] ToArray()
{
    return list.ToArray();
}
public int Count
{
    get
    {
        return list.Count;
    }
}
public void Clear()
{
    list.Clear();
}
```

Queue[Applications]

- Write a program to define a queue and put these element inside it {‘A’, ’%’ , ’5’ , ’B’ ,’&’ ,’C’ ,’7’, ’9’} , create three queues and put the numbers in first queue , symbol in second queue , and letter in third queue .
- Write a program to fill number in a queue from user , find sum for odd numbers and even numbers .
- write a program to create a queue and fill the queue capital letter and small letter from A-Z , and print all element in queue as Capital letter and small latter in each line

  Aa

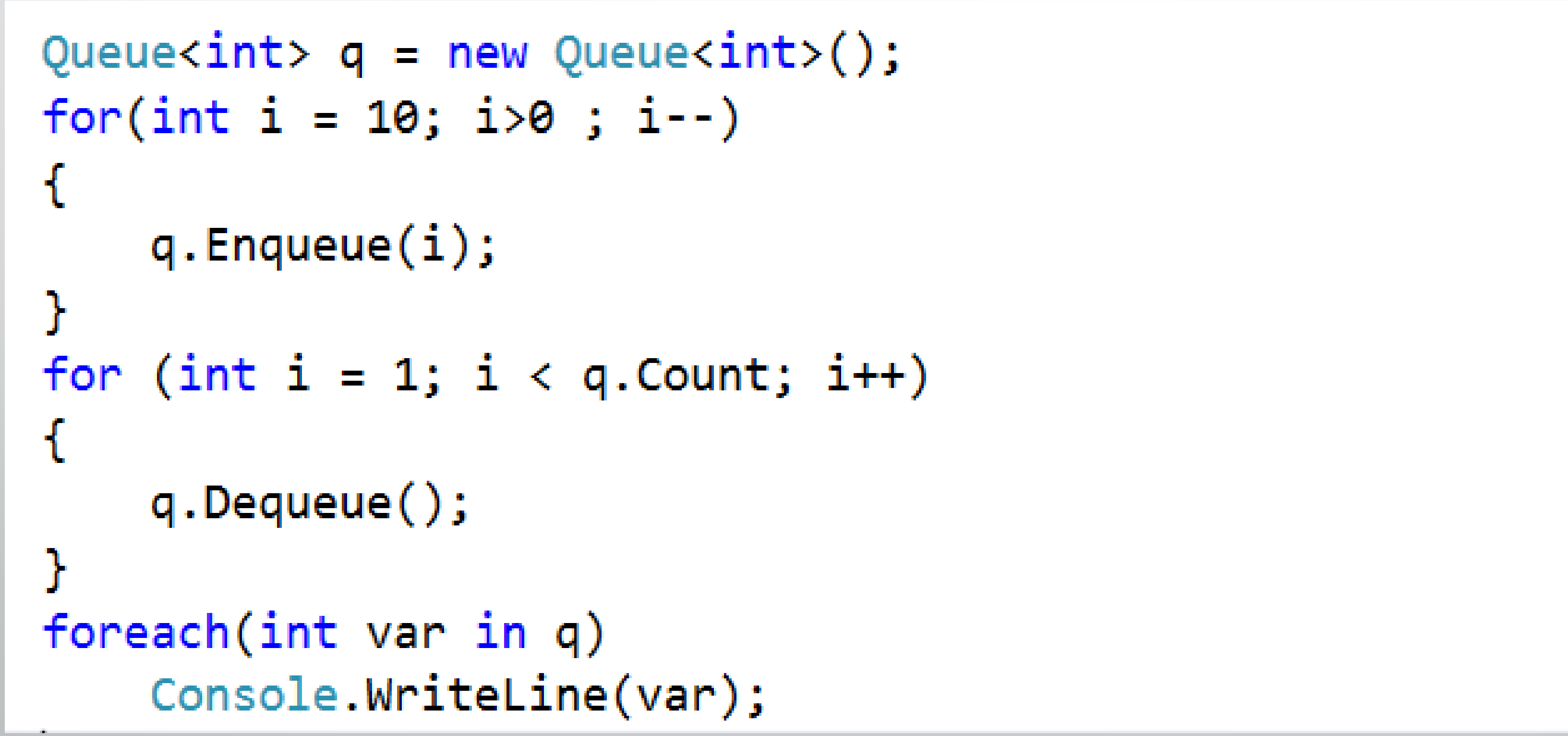
  Bb

  ..

  Zz

Queue[Applications]

- What is the output after execution the following code :



```
Queue<int> q = new Queue<int>();
for(int i = 10; i>0 ; i--)
{
    q.Enqueue(i);
}
for (int i = 1; i < q.Count; i++)
{
    q.Dequeue();
}
foreach(int var in q)
    Console.WriteLine(var);
```

Section Seven

Recursion

Recursion

- A recursive method is a method that call itself .
- A recursive algorithm typically follows a divide and-conquer approach.
- Divide and Conquer is a method of algorithm design. This method has three distinct steps:

  [1] Divide : divide the problem to sub problem

  [2] Conquer : solve the sub problem recursively .

  [3] Combine : merge the solution of sub problem to the original problem.
- Divide and conquer strategy often leads to efficient algorithms.
- Divide and conquer often produces efficient sort and search algorithms
- A recursive method must have at least one base, or stopping, case

Recursion

- **Recursive** Code

  - Easier to understand than their non-recursive

  - Code typically has fewer lines (shorter).

  - Often easier to maintain!

- **Non-Recursive** Code (iteration)

  - Code is longer.

  - Code executes faster

Example

- **Problem :** Write a program to find the factorial of 5 ?
- **Solution :**

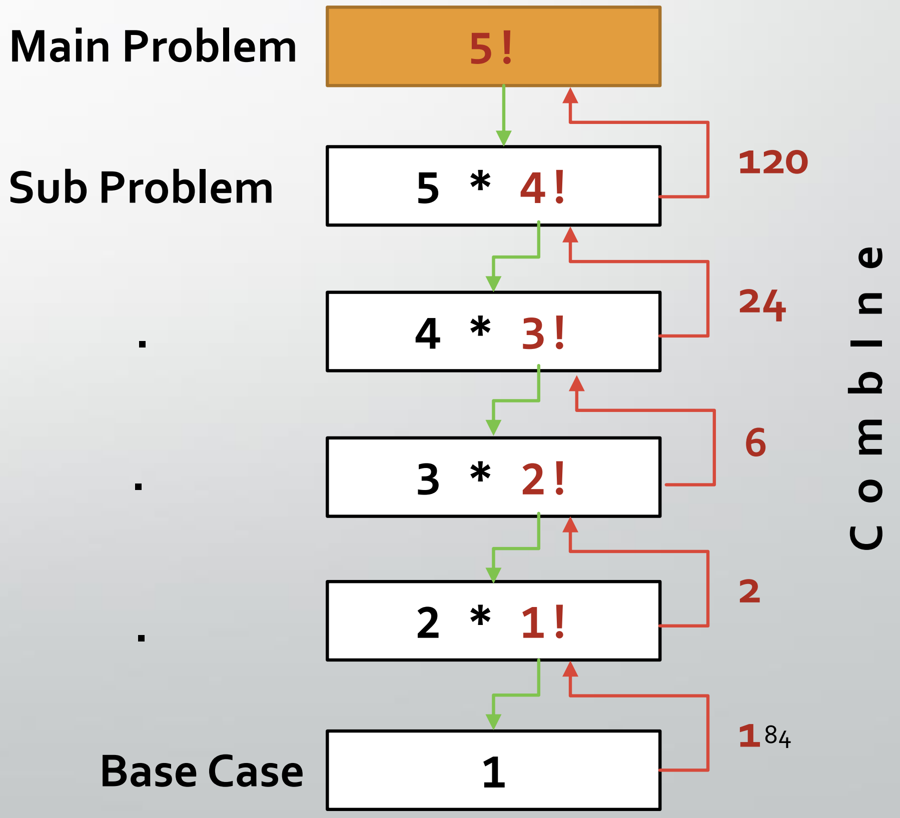
```
Console.Write("Enter number : ");
int num = Convert.ToInt32(Console.ReadLine());
int fact = 1;
for (int i = 2; i <=num; i++)
{
    fact *= i;
}
Console.WriteLine("Factorial = " + fact);
```

This solution is using alteration

Example

- **Problem :** Write a program to find the factorial of 5 ?
- **Solution :**

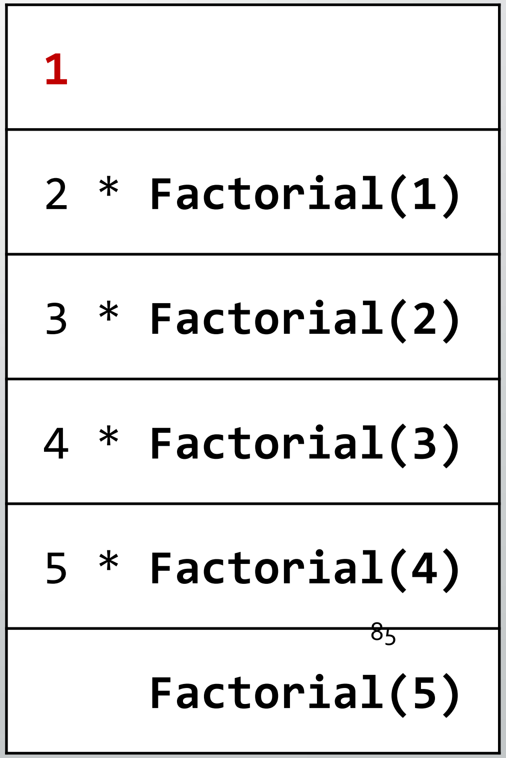
```
static int Factorial(int n)
{
    if (n == 1)
        return 1;
    else
        return n * Factorial(n - 1);
}
```



Tracing

- **To trace recursive method use stack**

```
static int Factorial(int n)
{
    if (n == 1)
        return 1;
    else
        return n * Factorial(n - 1);
}

static void Main(string[] args)
{
    Console.WriteLine("Factorial = " + Factorial(5));
}
```



1

2 * Factorial(1)

3 * Factorial(2)

4 * Factorial(3)

5 * Factorial(4)

Factorial(5)

Write Recursive Method

- **To write recursive method :**

  [1] Determine the base case

  [2] Determine the recursion case

  [3] Testing

- **Example** : write a recursive method to find sum of 3,2,4,5,.....n

  - base case : 3

  - recursion case : n-1

- **Example** : write a recursive method to find sum odd number from 1 to n ; if n is odd number

  - base case : 1

  - recursion case : n-2

Fibonacci Series

1	1	2	3	5	8	13	21
1	2	3	4	5	6	7	8

- $F_1 = 1$ and $F_2 = 1$ Or $F_1 = 0$ and $F_2 = 1$
- To Find Fibonacci number use the formula $F_n = F_{n-1} + F_{n-2}$

$F_5 = F_4 + F_3 = 3 + 2 = 5$

Fibonacci Series : 1 , 1 , 2 , 3 , 5 , 8 , 13 , 21 … **or** 0 , 1 , 1 , 2 , 3 , 5 , 8 , 13 , 21 …

Fibonacci Series

```
static void Main(string[] args)
 {
     Console.WriteLine("Fibonacci = " + Fibonacci(5));
 }
static int Fibonacci(int n)
 {
     if (n == 1 || n == 2)
       return 1;
     else
       return Fibonacci(n - 1) + Fibonacci(n - 2);
 }
```

Applications

[1] Write a recursive method to find if the element inside the array or not

[2] Write a recursive method to print “Hello World!” as reverse order.

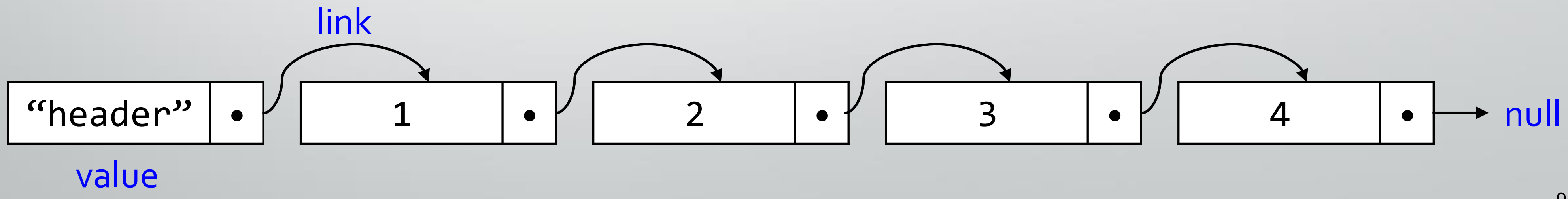
[3] Write a recursive method to find the max number in stack

Section Eight

Linked List

Linked list

- The linked list is series of nodes, each node has at least a single pointer to the next node , and in the last node's case a null pointer representing that there are no more nodes in the linked list.
- The first node in linked list called header
- Linked list is sequential data structure

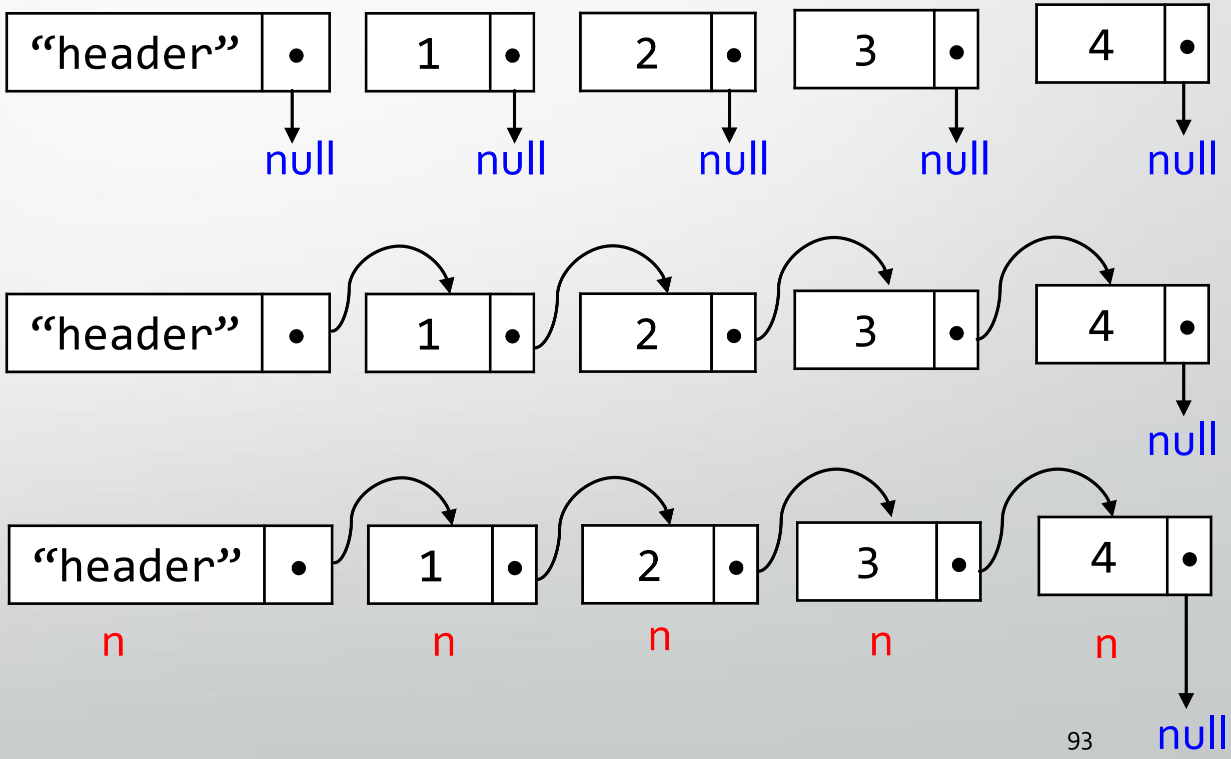


Linked list[Node]

```
public class Node
{

    public object value;
    public Node link;
    public Node(object value)
    {
        this.value = value;
    }
    public Node()
    {
    }

}
```

Linked list[Node]

```
Node header = new Node("header");
Node n1 = new Node(1);
Node n2 = new Node(2);
Node n3 = new Node(3);
Node n4 = new Node(4);
header.link = n1;
n1.link = n2;
n2.link = n3;
n3.link = n4;
Node n = header;
while(n.link!=null)
{

   n = n.link;
   Console.WriteLine(n.value);
}
```



Linked list [implementation]

- The class contains :

[1] Find : Finds the first node that contains the specified value (for insertion).

[2] FindPrevious : Finds the previous node of the node that contains the specified value.

[3] Add : Add new node at the end of linked list.

[4] Contains : Determines whether a value is in the linked list.

[5] Count : Gets the number of nodes actually contained in the linked list.

[6] Header : Get and set the value of header.

[7] Replace : To change the value of specified node .

[8] Insert : Inserts new node into the linked list after the specified node.

[9] Remove : Remove specified node from linked list.

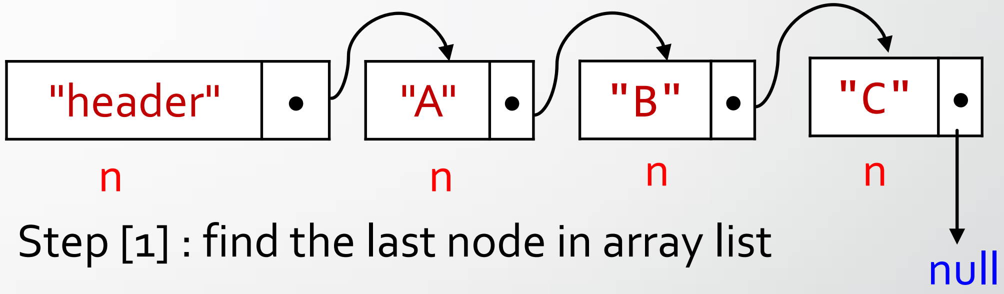
[10] ElementAt : Get the value of node at specified index.

[11] Print : To print value of all nodes in linked list

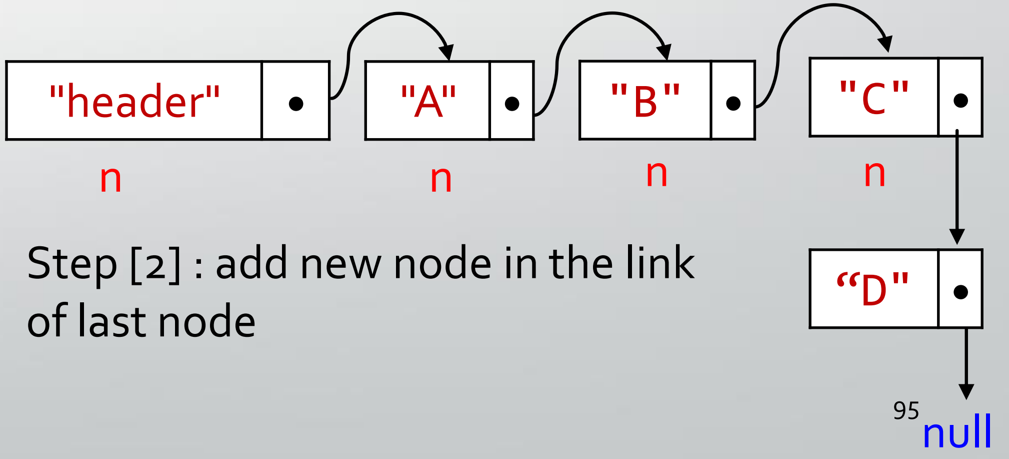
Linked list [Add]

Add new node at the end of linked list.

```
public class ALinkedList
{

   private Node header;
   public ALinkedList()
   {
      header = new Node("header");
   }
   public void Add(object value)
   {
      Node p = header;
      while (p.link!=null)
      {
         p = p.link;
      }
      p.link = new Node(value);
   }
```



Step [1] : find the last node in array list

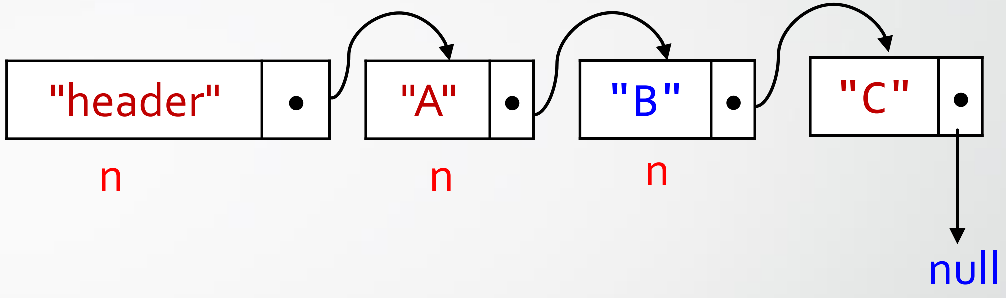


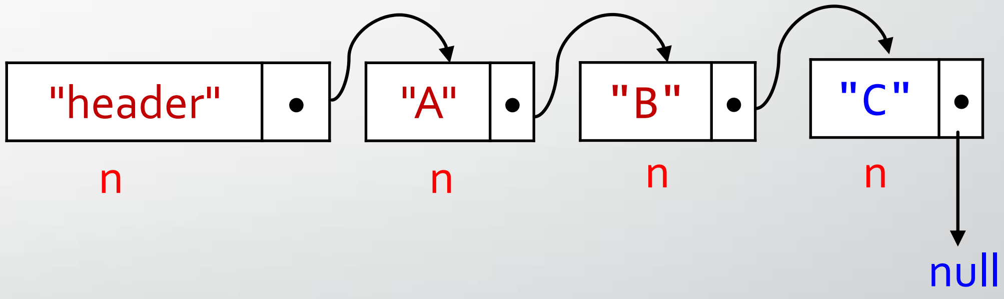
Step [2] : add new node in the link of last node

Linked list [Find]

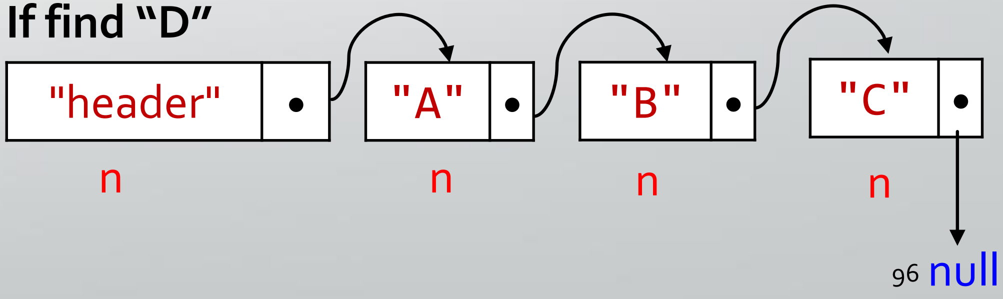
Finds the first node that contains the specified value

```
private Node Find(object value)
{
   Node p = header;
   while (p.link!=null)
   {
       if(p.value.Equals(value))
       {
         break;
       }
       p = p.link;
   }
   if(p.value.Equals(value))
       return p;
   else
       return null;
}
```





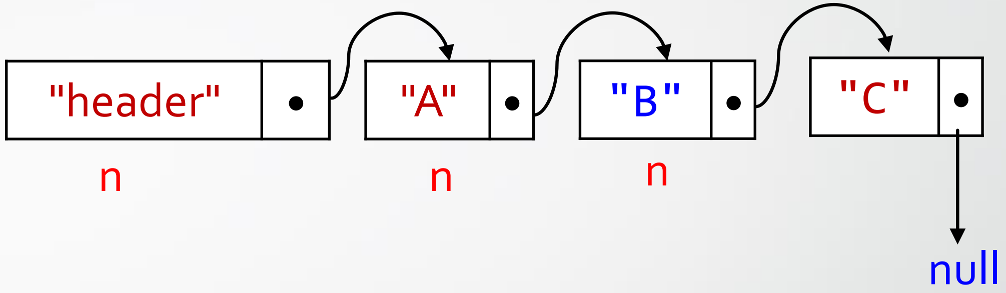
If find “D”

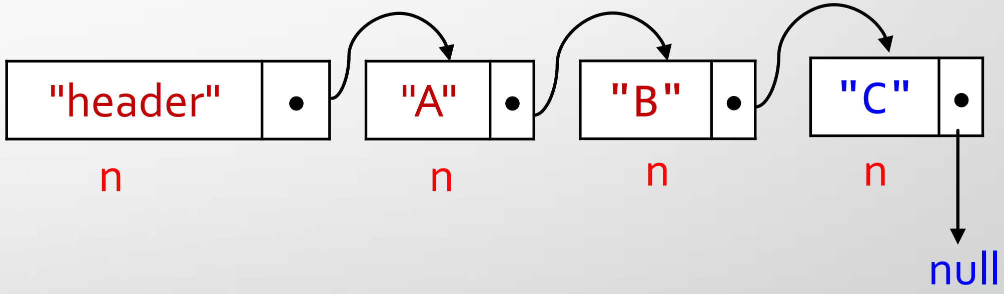


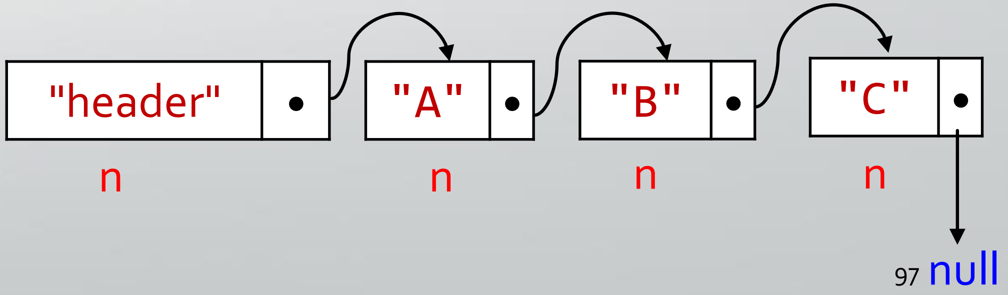
Linked list [FindPrevious]

Finds the previous node of node that contains the specified value

```
private Node FindPrevious(object value)
{
   Node p = header;
   while (p.link!=null)
   {
       if(p.link.value.Equals(value))
       {
         break;
       }
       p = p.link;
   }
   if(p.link!=null)
       return p;
   else
       return null;
}
```



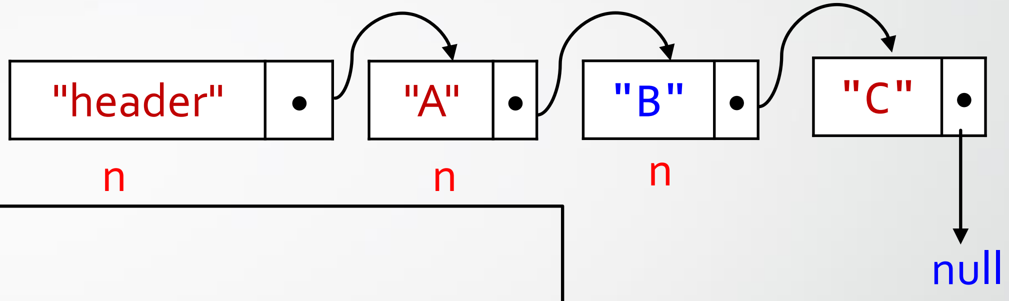




Linked list [Contains]

Determines whether a value is in the linked list

```
public bool Contains(object value)
{

   if(Find(value)!=null)


      return true;
   else
      return false;
}
```

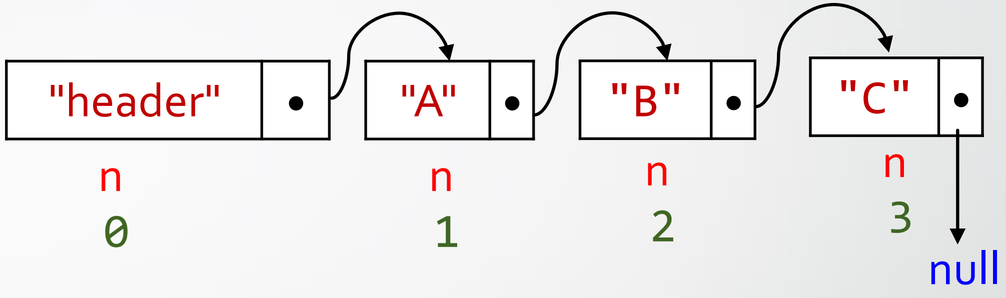


```
private Node Find(object value)
{
   Node p = header;
   while (p=null)
   {
       if(p.value.Equals(value)
       {
         break;
       }
       p = p.link;
   }
   if(p.value.Equals(value)
       return p;
   else
       return null;
}
```

Linked list [Count]

Gets the number of nodes actually contained in the linked list

```
public int Count()
{
  int count = 0;
  Node p = header;
  while(p.link!=null)
  {
     p = p.link;
     ++count;
  }
  return count;
}
```



"header" "A" "B" "C"

n n n n

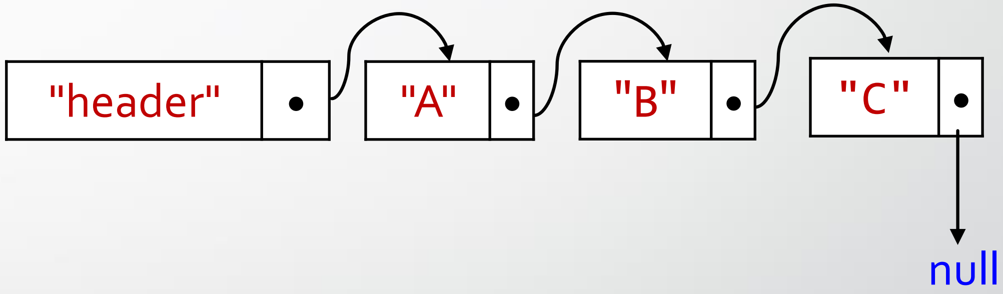
0 1 2 3

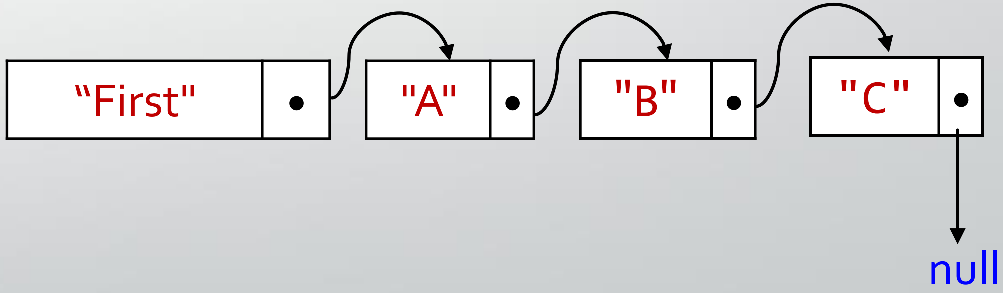
null

Linked list [Header]

Get and set the value of header

```
public object Header
{
  set {
        header.value = value ;
      }

  get {
        return header.value;
      }

 }
```

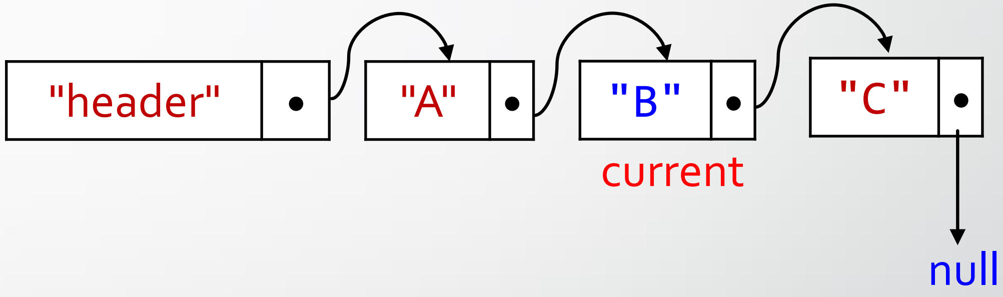


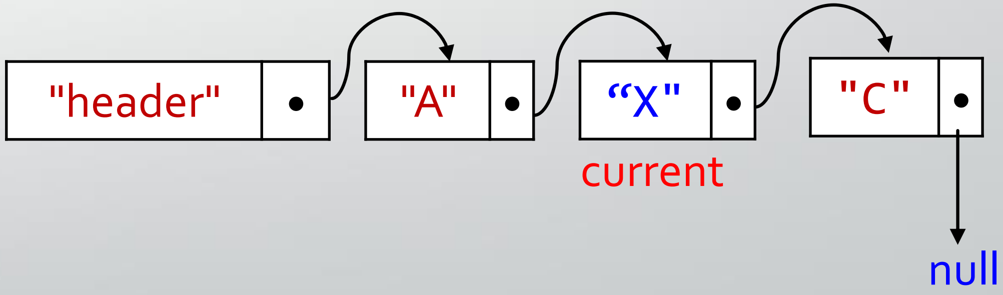


Linked list [Replace]

To change the value of specified node

```
public void Replace(object oldValue , object newValue)
{
   Node current = Find(oldValue);
   if(current!=null)
      current.value = newValue ;
}
```

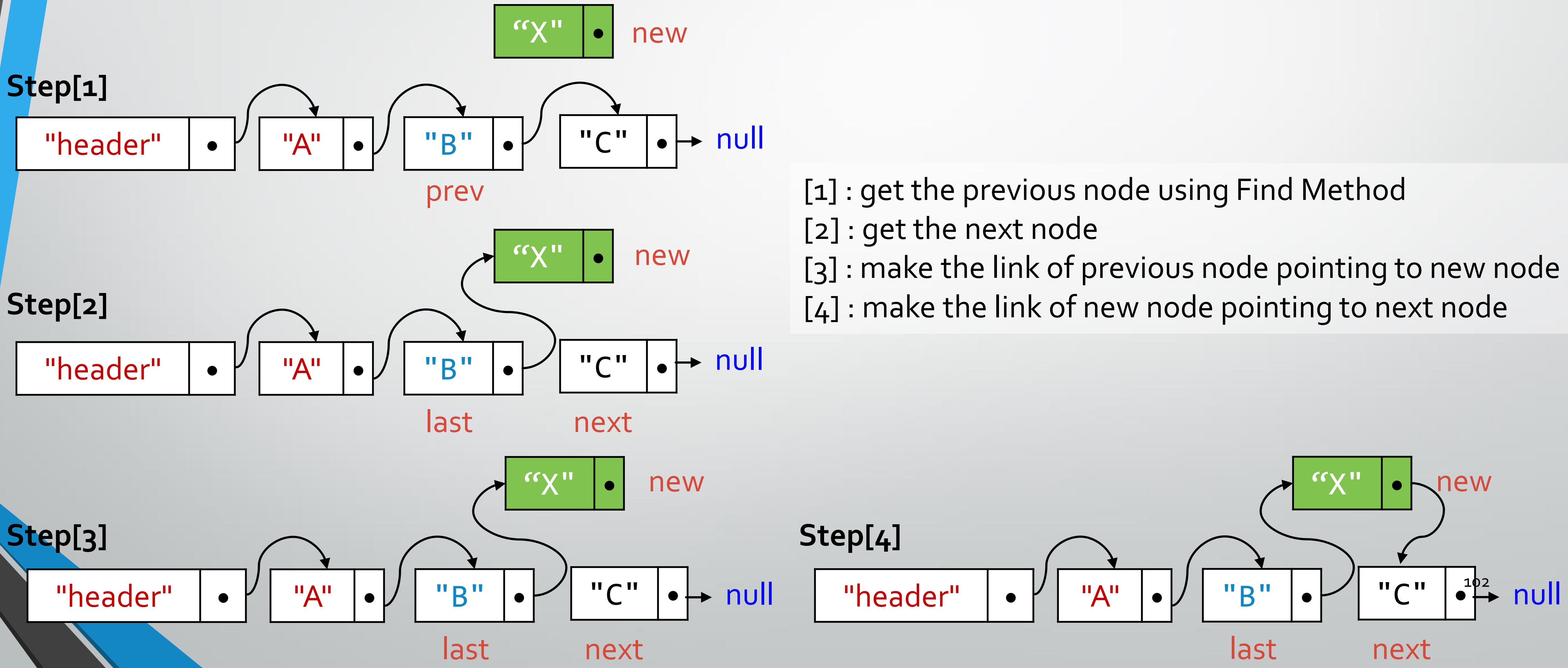




```
Note : the object of any class is reference type
```

Linked list [Insert]

Inserts new node into the linked list after the specified node



[1] : get the previous node using Find Method
[2] : get the next node
[3] : make the link of previous node pointing to new node
[4] : make the link of new node pointing to next node

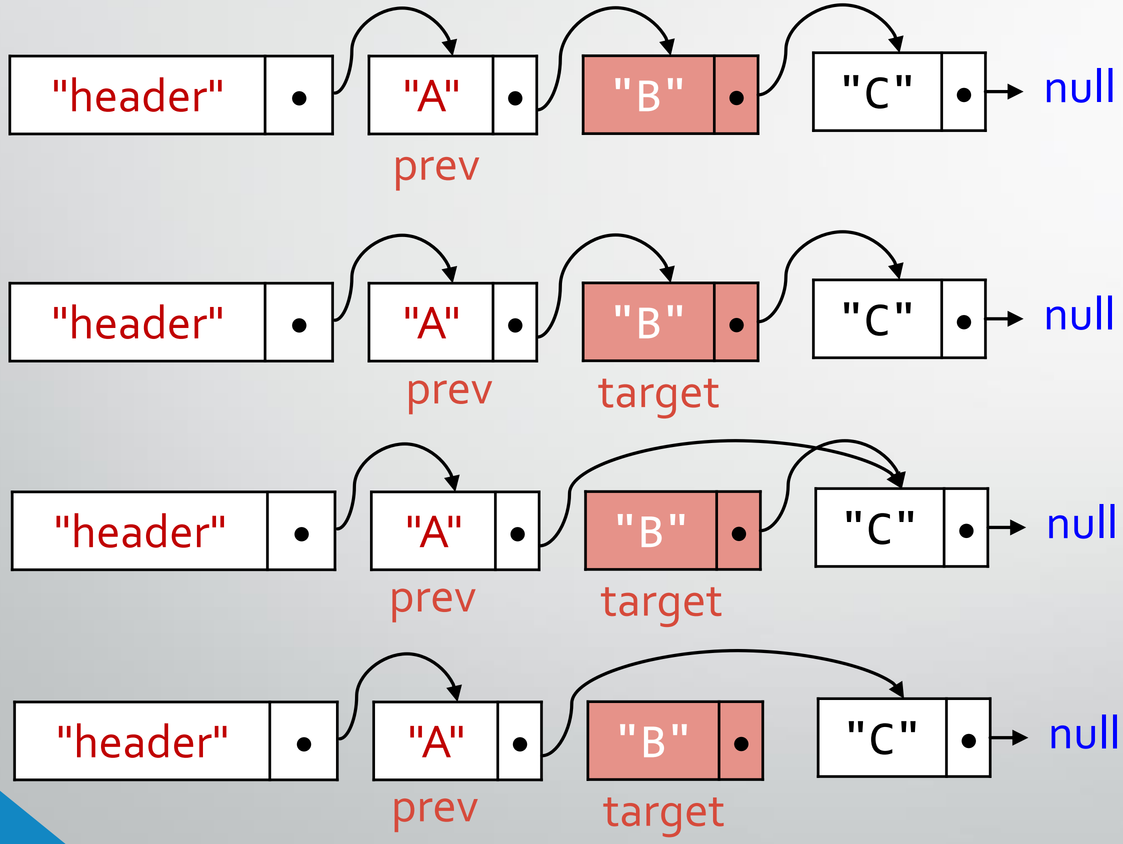
Linked list [Insert]

Inserts new node into the linked list after the specified node

```
public void Insert(object tar, object value)
{
   Node newNode = new Node(value);
   Node prev = Find(tar);
   if(tar==null)
      Add(newNode);
   else
   {
     Node next = prev.link;
     prev.link = newNode;
     newNode.link = next;
   }
}
```

Linked list [Remove]

Remove specified node from linked list



[1] : get the previous node using FindPrevious Method

[2] : get the target node

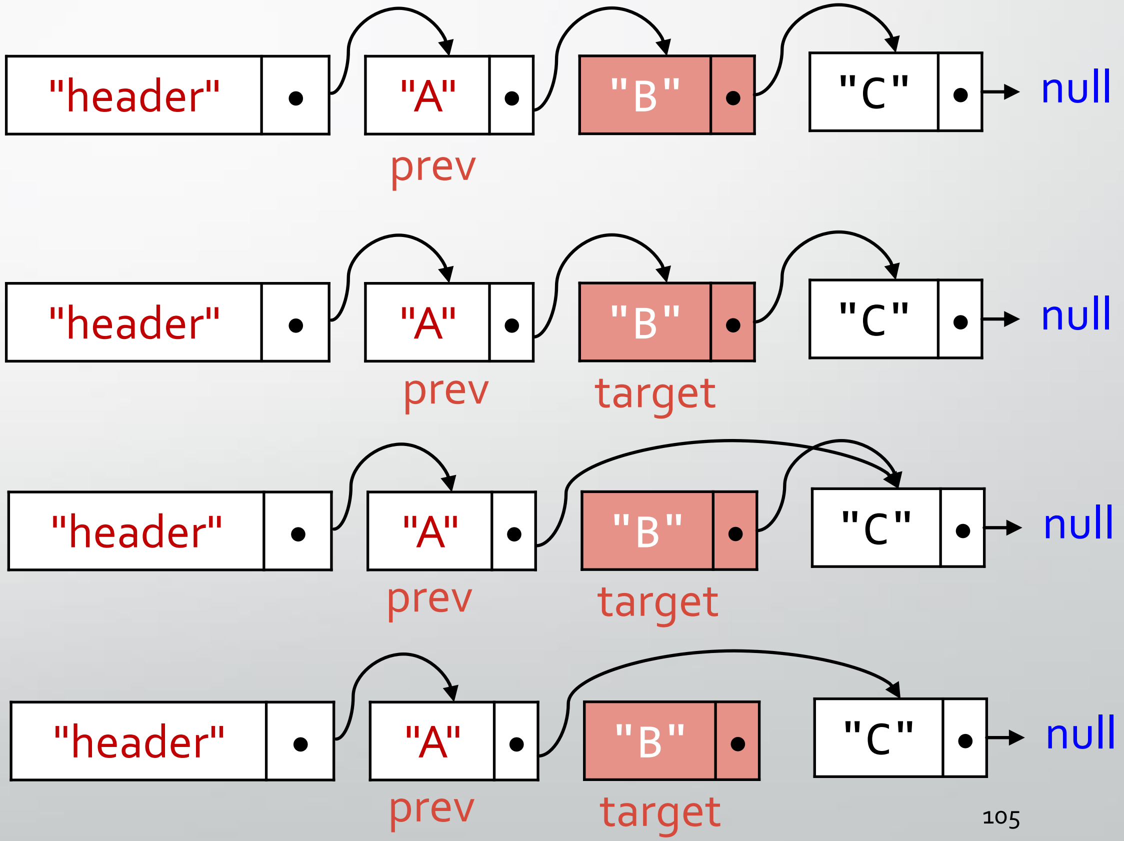
[3] : make the link of previous node pointing to next link of target node

[4] : make the link of target node pointing to null

Linked list [Remove]

Inserts new node into the linked list after the specified node

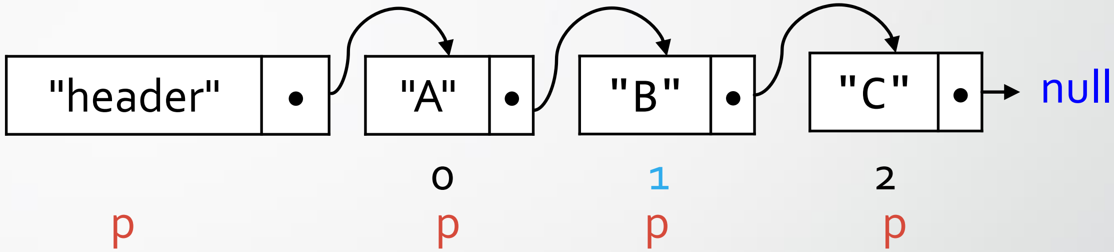
```
public void Remove(object tar)
{
    Node prev = FindPrevious(tar);
    if(prev==null)
        return;
    else
    {
      Node target = prev.link;
      prev.link = target.link;
      target.link = null;
    }
}
```



Linked list [ElementAt]

Get the value of node at specified index

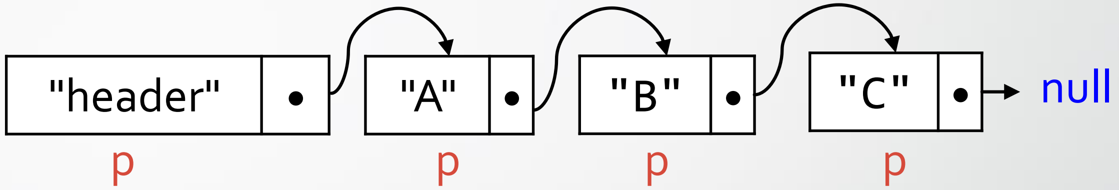
```
public object ElementAt (int index)
{
   Node p = header;
   int count = 0;
   object value = null;
   while(p.link != null)
   {
      p = p.link;
      if(index == count)
      {
        value = p.value;
        break;
      }
      ++count;
   }
   return value;
}
```



Linked list [Print]

To print value of all nodes in linked list

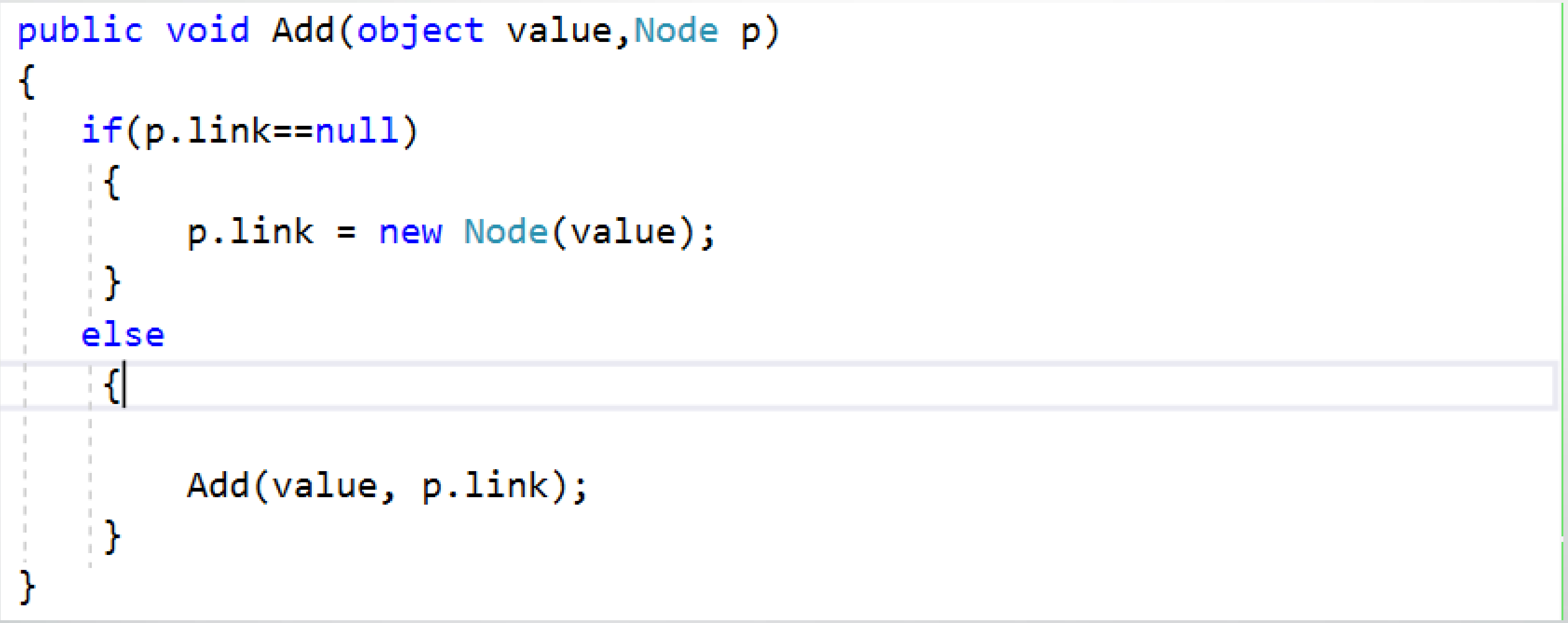
```
public void Print ()
{
   Node p = header;
   while (p.link != null)
   {
       p = p.link;
       Console.WriteLine(p.value);
   }
}
```



Recursion

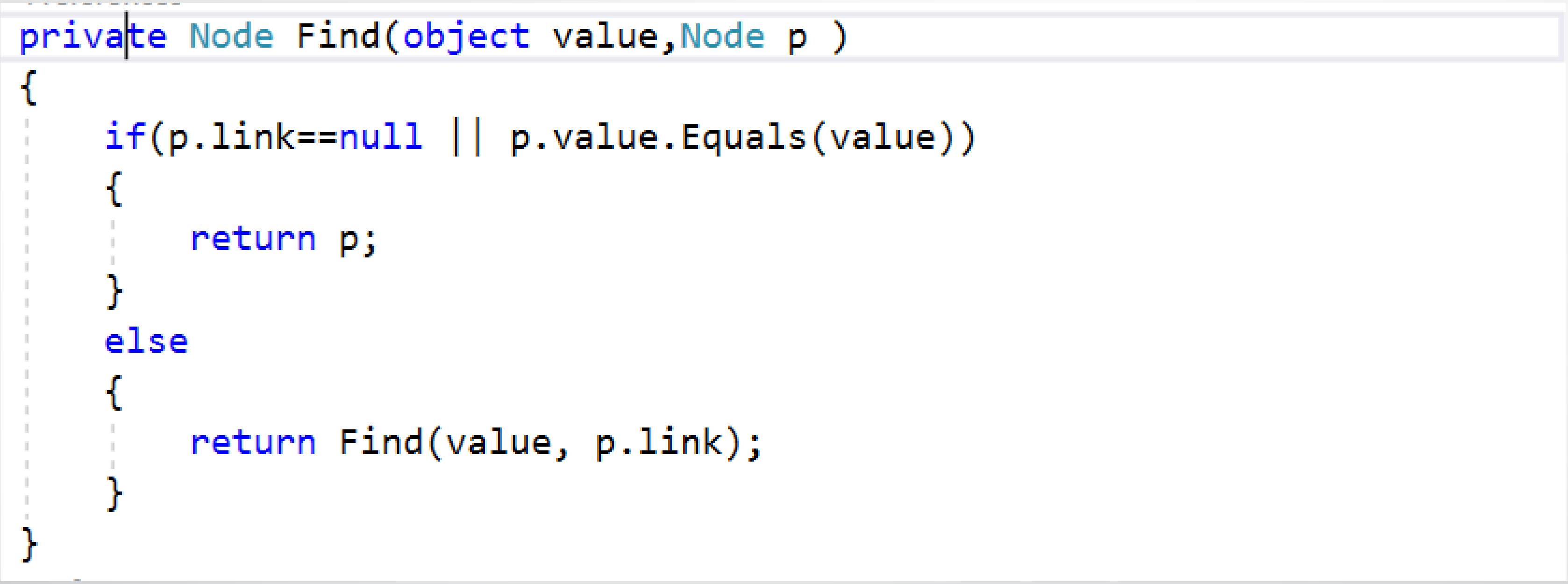
Method	Base Case	Recursion Case
void Add(object item)	Last node of linked list	Go to the next node
int Count()	Last node of linked list	Go to the next node and increment count by one
void Print()	Last node of linked list	Print the value of node and go to the next node
Node Find(object tar)	Find the target node or Last node of linked list	Go to the next node
Node FindPrev(object tar)	Find the target node or Last node of linked list	Go to the next node
Node ElementAt(int index)	Last node of linked list or index is equal of count	Go to the next node and increment count by one

Recursion



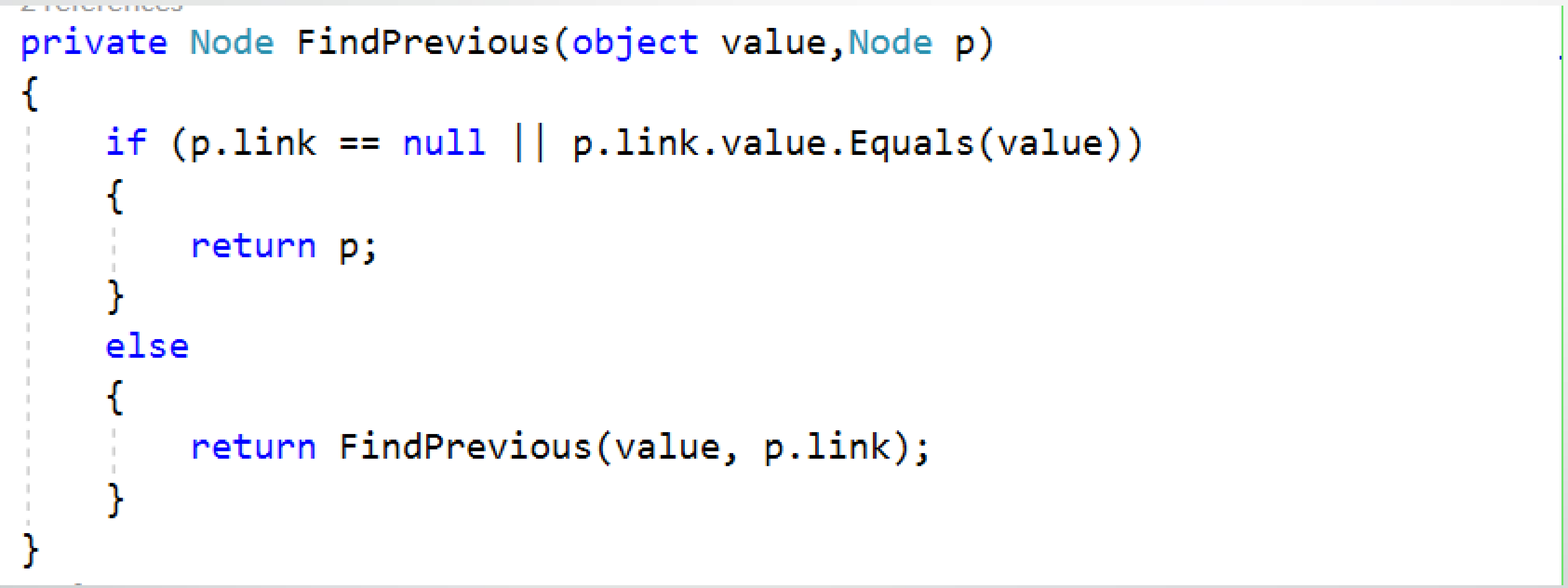
```
public void Add(object value,Node p)
{
   if(p.link==null)
    {
        p.link = new Node(value);
    }
   else
    {

        Add(value, p.link);
    }
}
```

Recursion



```
private Node Find(object value,Node p )
{
    if(p.link==null || p.value.Equals(value))
    {
        return p;
    }
    else
    {
        return Find(value, p.link);
    }
}
```

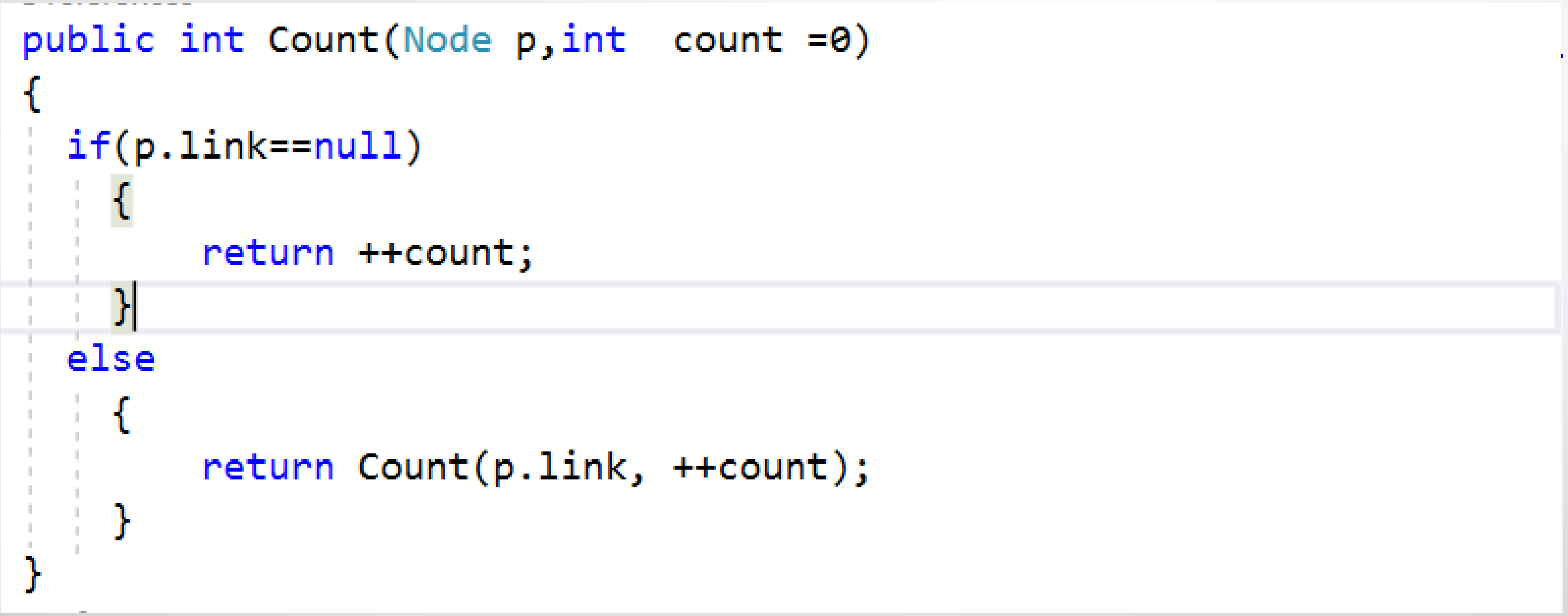
Recursion



2 references

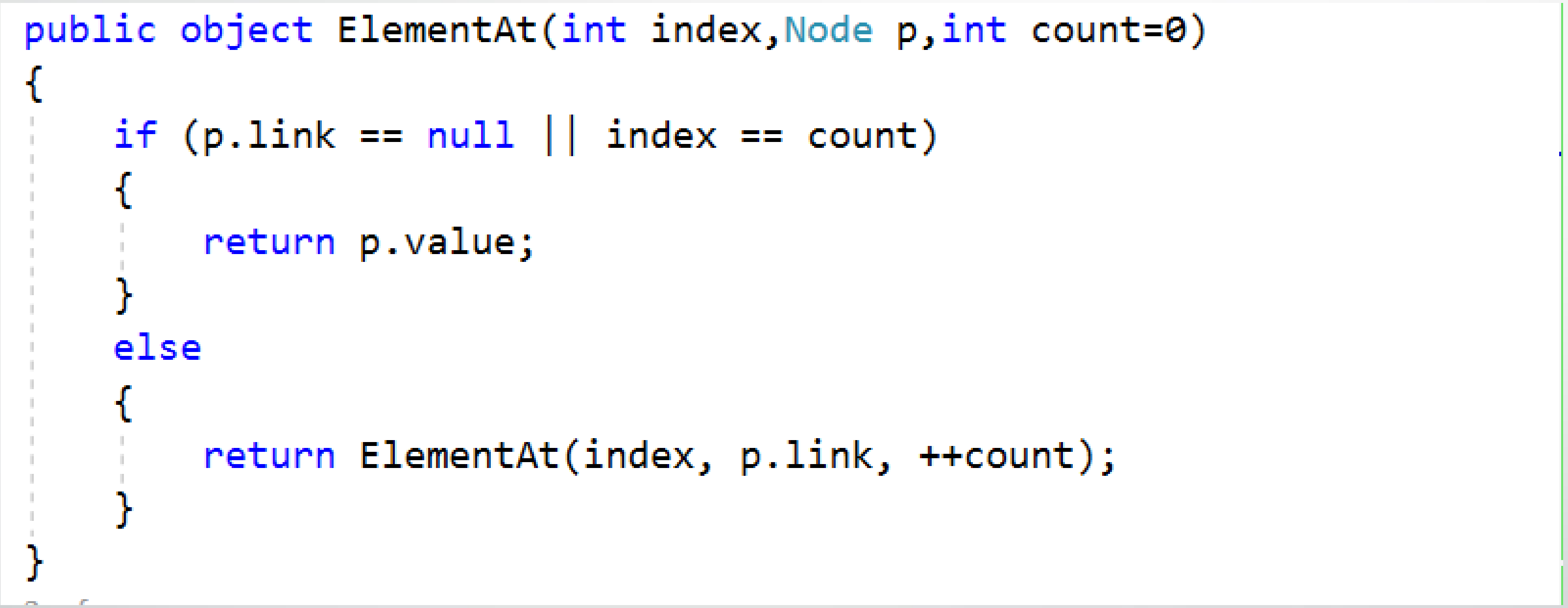
```
private Node FindPrevious(object value,Node p)
{
    if (p.link == null || p.link.value.Equals(value))
    {
        return p;
    }
    else
    {
        return FindPrevious(value, p.link);
    }
}
```

Recursion



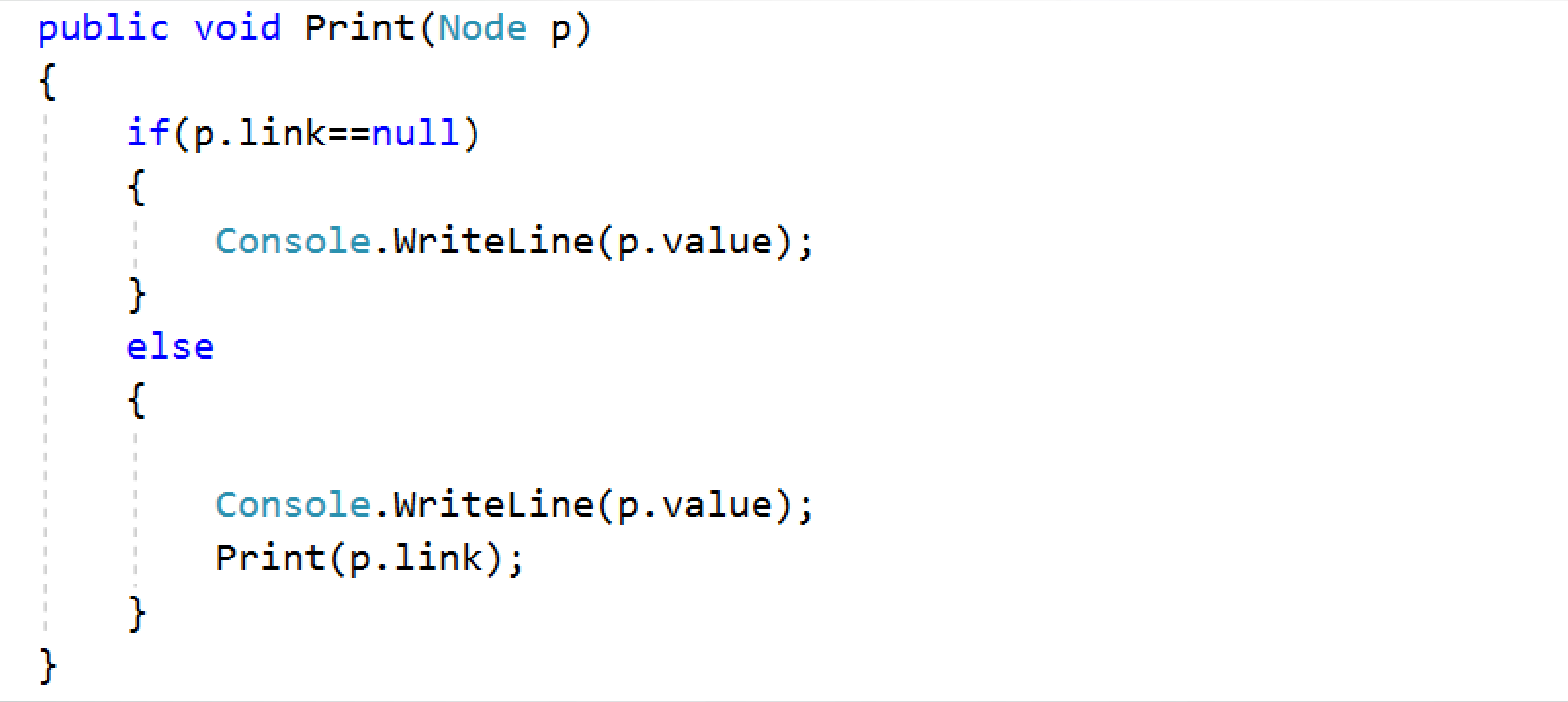
```
public int Count(Node p,int  count =0)
{
  if(p.link==null)
    {
        return ++count;
    }
  else
    {
        return Count(p.link, ++count);
    }
}
```

Recursion



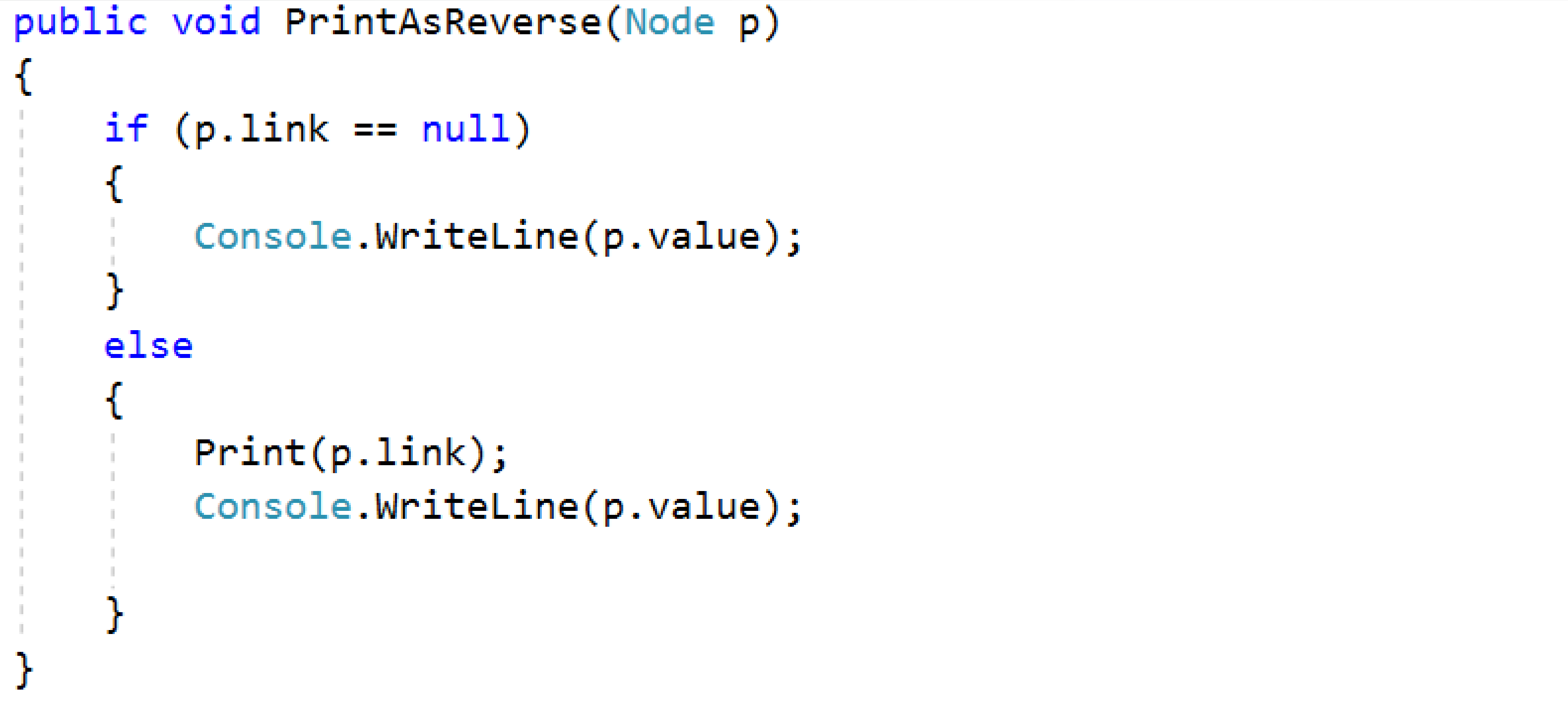
```
public object ElementAt(int index,Node p,int count=0)
{
    if (p.link == null || index == count)
    {
        return p.value;
    }
    else
    {
        return ElementAt(index, p.link, ++count);
    }
}
```

Recursion



```
public void Print(Node p)
{
    if(p.link==null)
    {
        Console.WriteLine(p.value);
    }
    else
    {

        Console.WriteLine(p.value);
        Print(p.link);
    }
}
```

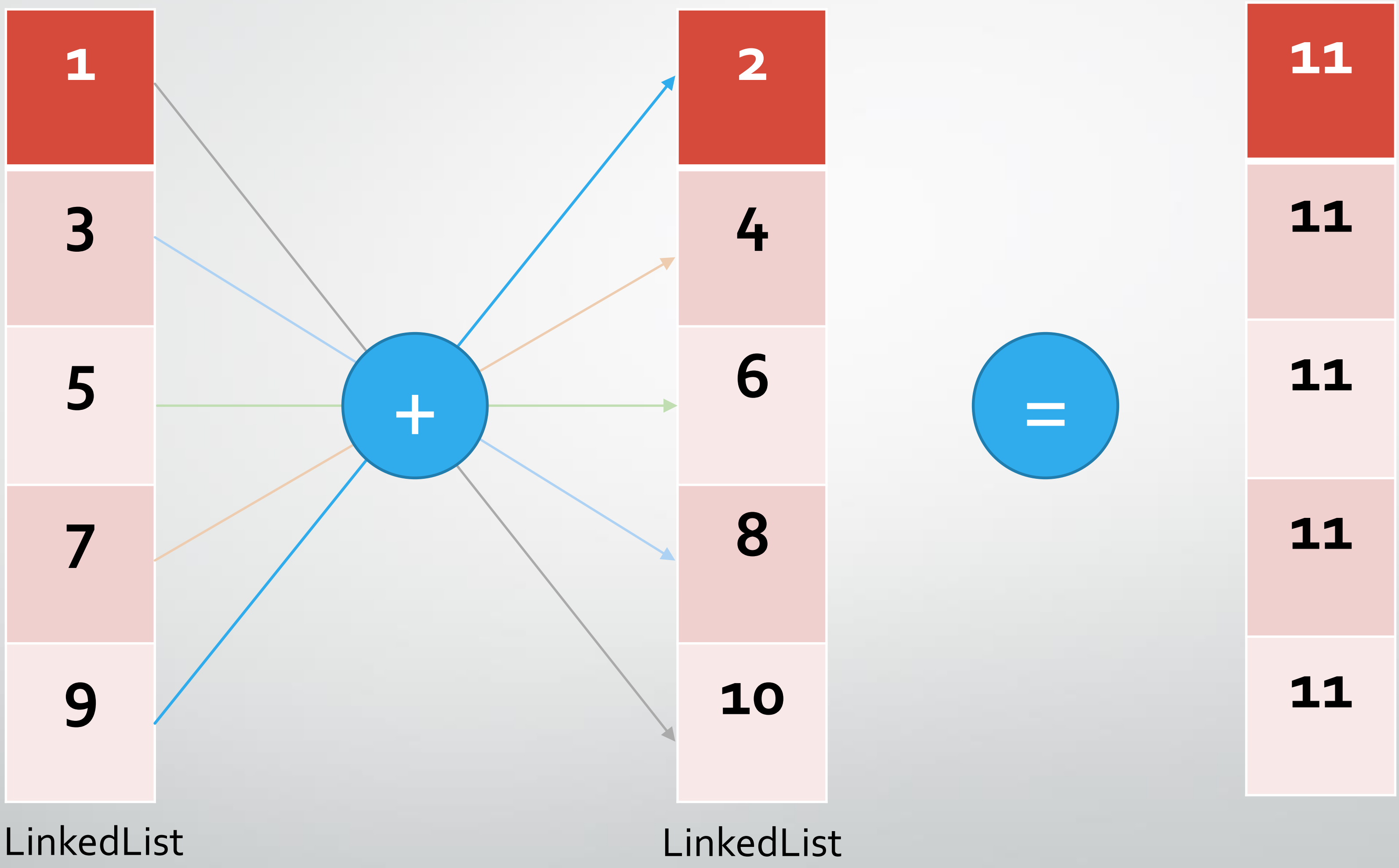
Recursion



```
public void PrintAsReverse(Node p)
{
    if (p.link == null)
    {
        Console.WriteLine(p.value);
    }
    else
    {
        Print(p.link);
        Console.WriteLine(p.value);

    }
}
```

Section Nine

Doubly Linked List



1
3
5
7
9

LinkedList

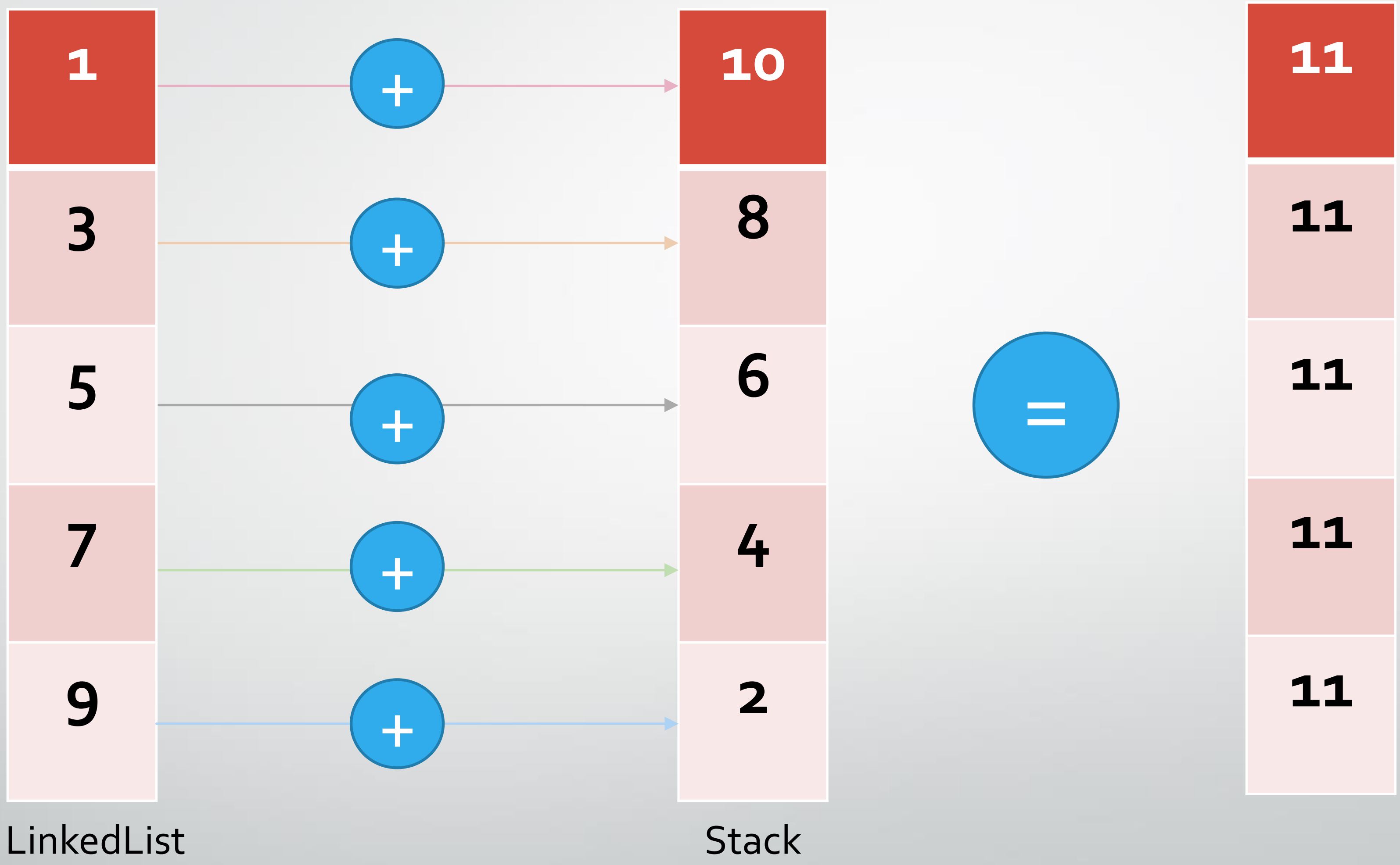
+

2
4
6
8
10

LinkedList

=

11
11
11
11
11



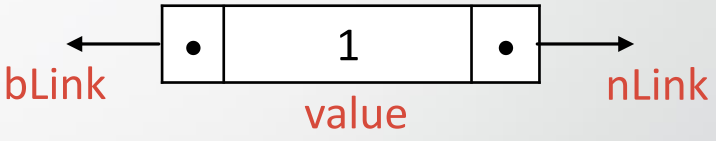
1	+	10		11
3	+	8		11
5	+	6	=	11
7	+	4		11
9	+	2		11
LinkedList		Stack		

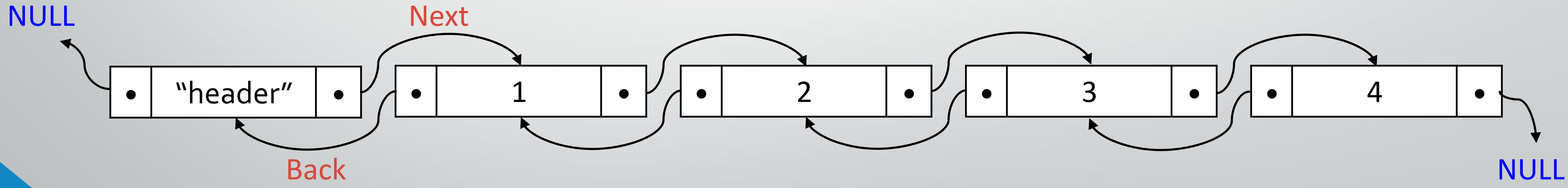
Doubly Linked list

- Each node in doubly linked list has two links :

  - First (Next) : pointing to next node

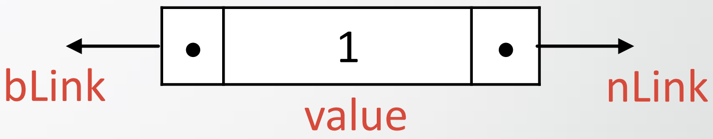
  - Second (Back) : pointing to previous node





Doubly Linked list[Node]

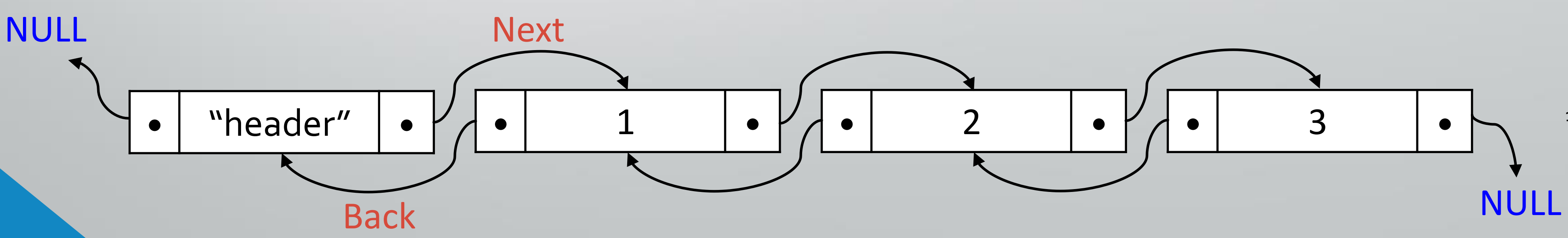
```
public class Node
{

    public object value;
    public Node bLink;
    public Node nLink;
    public Node(object value)
    {
        this.value = value;
    }
    public Node()
    {
    }

}
```



Doubly Linked list[Node]

```
Node header = new Node("header");
Node n1 = new Node(1);
Node n2 = new Node(2);
Node n3 = new Node(3);
header.nLink = n1;
n1.bLink=header;
n1.nLink=n2;
n2.bLink = n1;
N2.nLink=n3;
n3.bLink=n2;
```

```
Node p = header;
while(p.nLink!=null)
{
   Console.WriteLine(p.nLink.value);
   p=p.nLink;
}

---------------------------------
while(p.bLink!=null)
{
   Console.WriteLine(p.value);
   p=p.bLink;
}
```



Doubly linked list [implementation]

- The class contains :

[1] Find : Finds the first node that contains the specified value (for insertion).

[2] Add : Add new node at the end of doubly linked list.

[3] Insert : Inserts new node into the doubly linked list after the specified node.

[4] Remove : Remove specified node from doubly linked list.

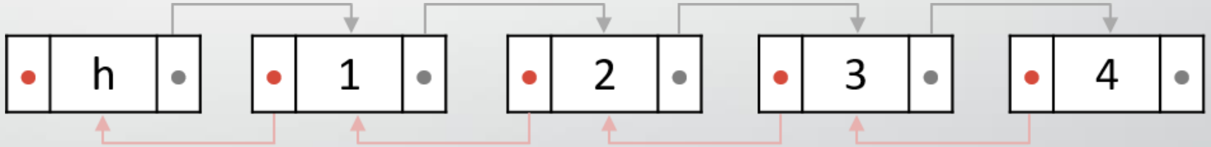
[5] Print : To print value of all nodes in doubly linked list

[6] PrintReversly : To print value of all nodes in doubly linked list as revers order.

Doubly Linked list [Add]

Add new node at the end of doubly linked list.

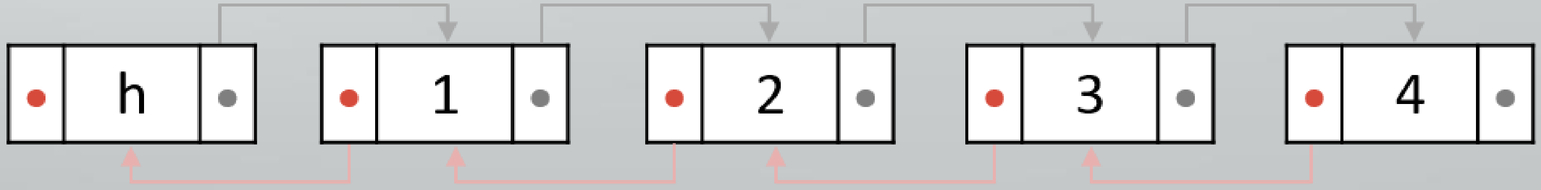
```
public class DoublyLinkedList
{
    private Node header;
    public DoublyLinkedList()
    {
        header = new Node("header");
    }
    public void Add(object item)
    {
        Node p = header;
        while(p.nLink!=null)
        {
            p = p.nLink;
        }
        Node newNode = new Node(item);
        p.nLink = newNode;
        newNode.bLink = p;
    }
```



Doubly Linked list [Find]

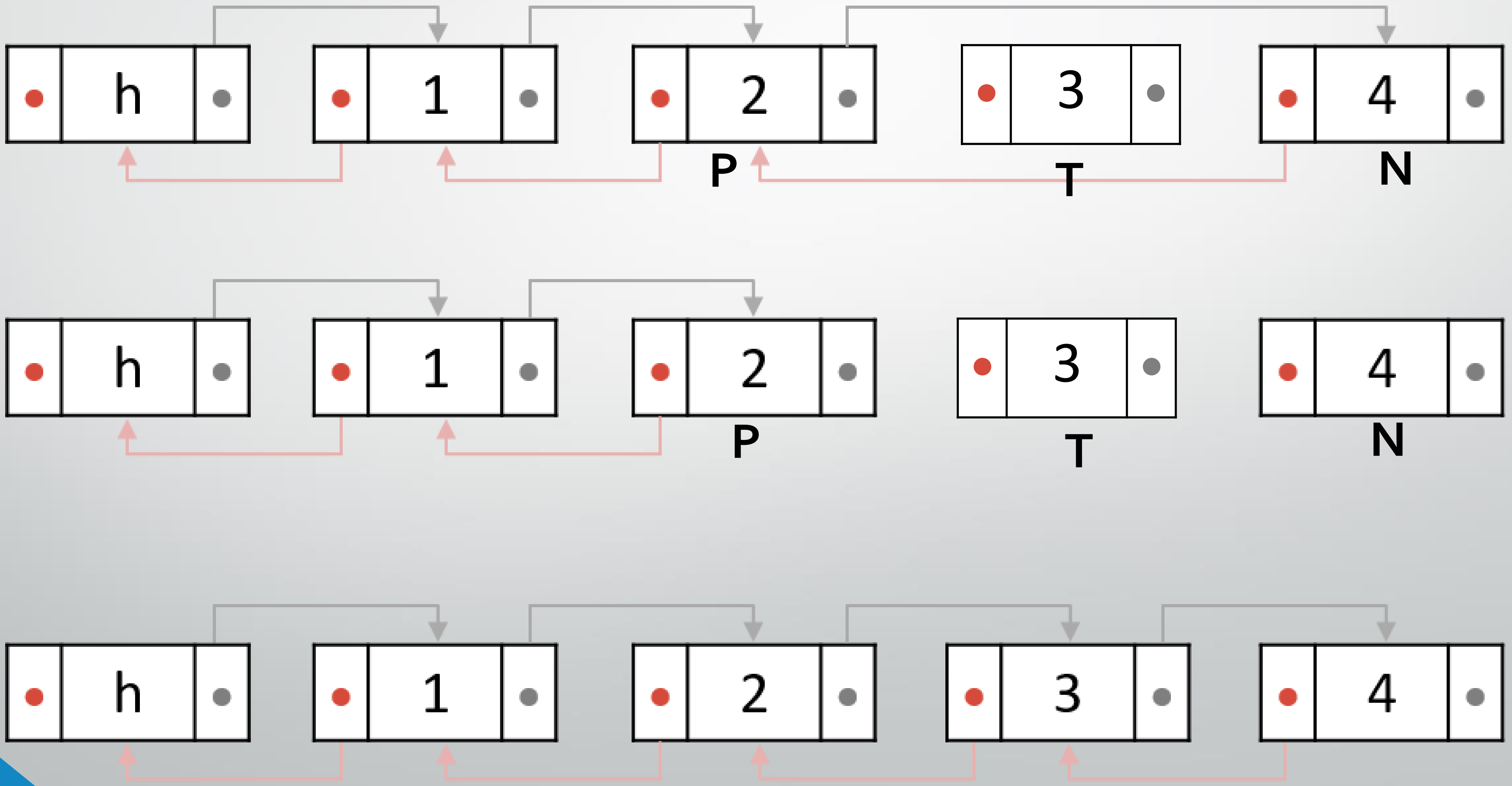
Finds the first node that contains the specified value

```
private Node Find(object value)
{
    Node p = header;
    while(p.nLink!=null)
    {
        if(p.value.ToString().Equals(value.ToString()))
        {
            break;
        }
        p = p.nLink;
    }
    if(p.value.ToString().Equals(value.ToString()))
    {
        return p;
    }
    else
    {
        return null;
    }
}
```



Linked list [Insert]

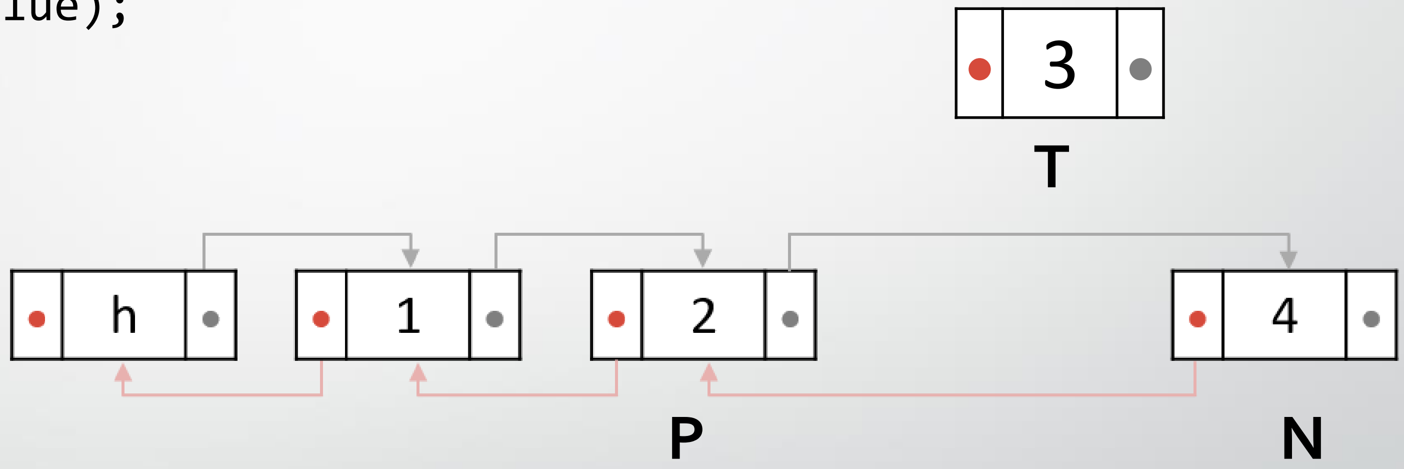
Inserts new node into the doubly linked list after the specified node



Linked list [Insert]

Inserts new node into the doubly linked list after the specified node

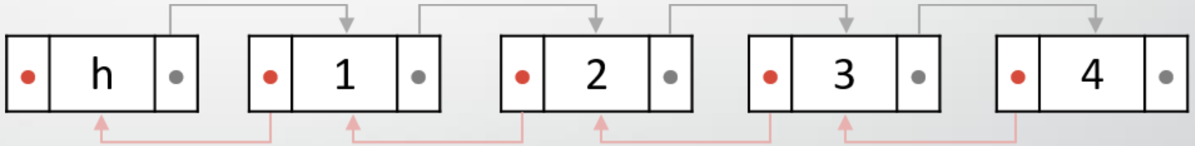
```
public void Insert(object after,object value)
{
    Node target = new Node(value);
    Node prev = Find(after);
    Node next = prev.nLink;
    prev.nLink = target;
    target.bLink = prev;
    next.bLink = target;
    target.nLink = next;

}
```



Linked list [Insert]

Inserts new node into the doubly linked list after the specified node

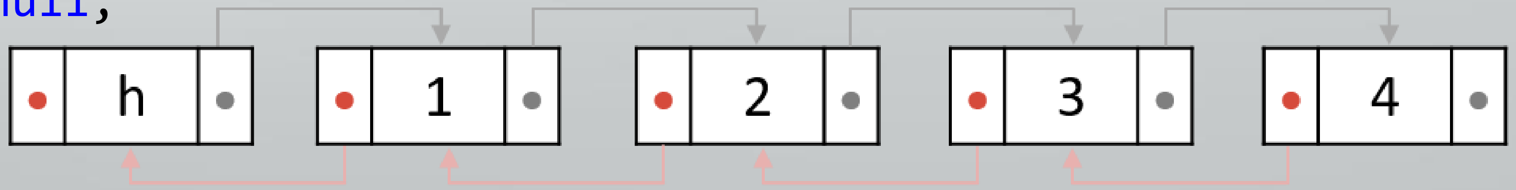
```
public void Insert(object after,object value)
{
    Node target = new Node(value);
    Node prev = Find(after);
    if(prev == null || prev.nLink==null )
    {
        Add(value);
    }
    else
    {
        Node next = prev.nLink;
        prev.nLink = target;
        target.bLink = prev;
        next.bLink = target;
        target.nLink = next;
    }
}
```



Linked list [Remove]

Inserts new node into the linked list after the specified node

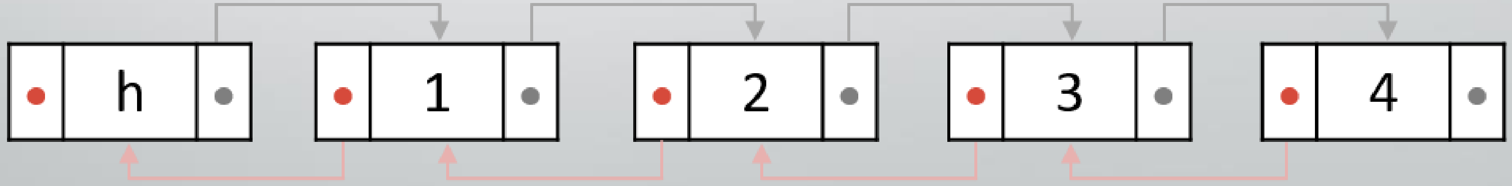
```
public void Remove(object tar)
{
    Node target = Find(tar);
    if (target != null)
    {
        Node prev = Find(tar).bLink;

        Node next = target.nLink;
        prev.nLink = next;
        if (next != null)
        {
            next.bLink = prev;
        }
        target.bLink = null;
        target.nLink = null;
    }
}
```



Linked list [Print]

To print value of all nodes in linked list

```
public void Print()
{
    Node p = header;
    while(p.nLink!=null)
    {
        Console.WriteLine(p.nLink.value);
        p = p.nLink;
    }
}
```



Linked list [PrintReversly]

To print value of all nodes in linked list

```
public void PrintReversly()
{
  Node p = header;
  while (p.nLink != null)
  {
     p = p.nLink;
  }
  while(p.bLink!=null)
  {
     Console.WriteLine(p.value);
     p = p.bLink;
  }
}
```