USN | | | | | | | | | | |

Scheme of Evaluation for Internal Assessment Test III– NOVEMBER 2018

Sub:	Dot Net Application and Framework Development					Sub Code:	15CS564	Branch :	CSE/ISE
Date:	22/11/2018	Duration:	90 min's	Max Marks:	50	Sem / Sec:	5/A,B & C		

Answer any FIVE FULL Questions MARKS

1 What are generics? Explain the importance of generics. [10]

Solution: Definition with example:6M+Importance:4M=10M

Generics are used to create generalized classes and methods. A generic class can be defined using angle brackets <> as shown below

```
Class classname<T>
{
……
……
……

}
```

Where T is a type parameter, which is a placeholder for a real type at compile time.

Example:

```
using System;
class Queue <T>
{
        private T [ ] data;
        private int head = 0, tail = 0;
        private int count = 0;
        public Queue(int s)
        {
                if (s > 0)
                this.data = new T[s];
                else
                {
                        Console.WriteLine("invalid");
                }
        }

public void Enqueue (T ele)
{
        if (this.count == this.data.Length)
        {
                throw new Exception("Queuefull");
        }
        this.data [this.head] = ele;
        this.head++;
        this.head %= this.data.Length;
        this. count ++;
```

```
}
public T Dequeue ()
{
        if (this.count == 0)
        {
                Console.WriteLine("Queueempty");
        }
        T ele = this.data [this.tail];
        this.tail++;
        this.tail %= this.data.Length;
        this.count --;
        return ele;
}

}
public class mainDemo
{
public static void Main()
        {
                Queue<int> n1 = new Queue<int>(5);
                Queue<string> n = new Queue<string>(5);
                n1.Enqueue(1);
                n1.Enqueue(2);
                n1.Dequeue();
                n.Enqueue("one");
                n.Enqueue("two");
                n.Enqueue("three");
                n.Enqueue("four");
                n.Enqueue("five");
                n.Dequeue();


        }
}
```

Importance of generics:

- It is important to know that we have to cast the values returned by a method.
  Ex:
  Console.WriteLine (q.Dequeue ()); **//Error**
- This compiler throws an error saying that implicit type casting is not possible.
- Suppose if we perform invalid casting, say
  circle mycircle = (circle) q.Dequeue ();
  But the deleted element is Horse Object.So syntactically it is correct but at the runtime throws an exception i.e. System.InvalidCastException.
- Generalized class and methods consumes additional memory and processor time if the runtime needs to convert an object to a value type.
  For this purpose, we use generic class.
- Generic classes use type parameters. Generalized class accepts any type of parameter and at different type.
- In generic classes, boxing type casting is not required but generalized class needs it.

- In generic class, based on type, the compiler generates a separate code. This is called as constructed type.

2 Write a C# program to implement Binary Search Tree operations insert and traversal using IEnumerable interface. [10]

Solution: Program:10M

```
using System;
using System.Collections;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace BinaryTree
{

    public class Tree<TItem> : IEnumerable<TItem> where TItem : IComparable<TItem>
    {
        public TItem NodeData { get; set; } public
        Tree<TItem> LeftTree { get; set; } public
        Tree<TItem> RightTree { get; set; }

        public Tree(TItem nodeValue)
        {
            this.NodeData = nodeValue;
            this.LeftTree = null;
            this.RightTree = null;
        }

        public void Insert(TItem newItem)
        {
            TItem currentNodeValue = this.NodeData;


            if (currentNodeValue.CompareTo(newItem) > 0)
            {
                // Insert the item into the left subtree if
                (this.LeftTree == null)
                {
                    this.LeftTree = new Tree<TItem>(newItem);
                }


                else
                {
                    this.LeftTree.Insert(newItem);
                }
```

```
            }

            else
            {
                // Insert the new item into the right subtree
                if (this.RightTree == null)
                {
                    this.RightTree = new Tree<TItem>(newItem);
                }
                else
                {
                    this.RightTree.Insert(newItem);
                }
            }
        }

        public string WalkTree()
        {
            string result = "";

            if (this.LeftTree != null)
            {
                result = this.LeftTree.WalkTree();
            }

            result += String.Format($" {this.NodeData.ToString()} ");

            if (this.RightTree != null)
            {
                result += this.RightTree.WalkTree();
            }

            return result;
        }

        IEnumerator<TItem> IEnumerable<TItem>.GetEnumerator()
        {
            return new TreeEnumerator<TItem>(this);
        }

        IEnumerator IEnumerable.GetEnumerator()
        {
            throw new NotImplementedException();
        }
    }
}
```

3 Write a C# program to implement QUEUE operations Insert and Delete using Generics concepts. [10]

Solution: Program:10M

```
using System;
class Queue <T>
{
        private T [ ] data;
        private int head = 0, tail = 0;
        private int count = 0;
        public Queue(int s)
        {
                if (s > 0)
                this.data = new T[s];
                else
                {
                        Console.WriteLine("invalid");
                }
        }
public void Enqueue (T ele)
{
        if (this.count == this.data.Length)
        {
                throw new Exception("Queuefull");
        }
        this.data [this.head] = ele;
        this.head++;
        this.head %= this.data.Length;
        this. count ++;
}
public T Dequeue ()
{
        if (this.count == 0)
        {
                Console.WriteLine("Queueempty");
        }
        T ele = this.data [this.tail];
        this.tail++;
        this.tail %= this.data.Length;
        this.count --;
        return ele;
}

}
public class mainDemo
{
public static void Main()
        {
                Queue<int> n1 = new Queue<int>(5);
                Queue<string> n = new Queue<string>(5);
```

```
            n1.Enqueue(1);
            n1.Enqueue(2);
            n1.Dequeue();
            n.Enqueue("one");
            n.Enqueue("two");
            n.Enqueue("three");
            n.Enqueue("four");
            n.Enqueue("five");
            n.Dequeue();
        }
}
```

4 What are collection classes? Explain in detail about collection classes. [10]

Solution: Defition:2M, Explanation with examples:8M =10M

The Microsoft .NET framework provides several classes that collect elements together that an application can access the elements in specialized ways. These collection classes live in System.Collections.Generic namespace.

- We have to create instances for the generic classes to use them.
- System.Collections.Generic namespace contains all these generic classes.
- Different types of collection:
    - List
    - Linked List
    - Stack
    - Queue
    - Hash

Brief overview of these collections

collection	Description
List<T>	A list of objects that can be accessed by index as with an array, but additional methods with which to search the list and sort the contents of the list
Queue<T>	A first in first out data structure with methods to add an element to one end of the queue remove an item from another end, and examine the item without removing it
Stack<T>	A first in last out data structure with methods to push an item on to the top of the stack pop an item from the top of the stack, and examine the item at the top of the stack without removing it
LinkedList <T>	A double ended ordered list optimized to support insertion and removal at either end, this collection can act like queue or stack but it also supports random access like a list does
HashSet <T>	An unordered set of values that is optimized for the fast retrieval of data, it provides set oriented methods for determining whether the items it holds are the subset of those in another HashSet <T> object as Well as computing the intersection and union of HashSet <T>objects.

Dictionary<TKey,TValue>	A collection of values that can be identified and retrieved by using keys rather than indexers.
SortedList<Tkey,TValue>	A sorted of key/value pairs. The keys must implement the IComparable <T>interface

List<T>

- It is the simplest form of collection classes.
- We can use square brackets and the index for displaying the elements of the list as like array, but adding or deleting an element is not possible by using array syntax.
- No need to specify the capacity when we create it because it will grow or shrink as we add or delete elements.
- Methods:

    - Add (val) – Inserts a value at the end of the list.
    - Insert (index, val) – Inserts a value at a specific index position.
    - Remove (val) – Removes the values. If there are multiple same values, it removes the first occurrence of the value. It returns the value that is removed.
    - RemoveAt (index) – Removes value from the specified index position.
    - Sort – Sorts the values in ascending order.

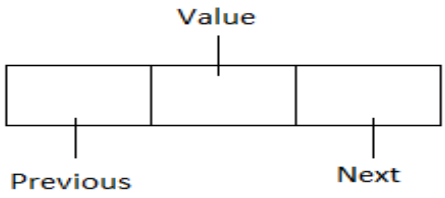
- Uses property count:
  It uses count to keep track of number of values in the list.

Ex:

```
static void Main()
{
        List <int> l = new List <int>
        (); l.add (2);
        foreach (int x in new int [5] {10, 4, 15, -4, 3})
                l.add (x);
        l.Insert (3,100);
        i l.Remove (100);              //100 is removed
        l.RemoveAt (3);                //15 is removed (index 3)
        //Displaying the list elements
        Console.WriteLine ("Elements in list:");
        foreach (int x in l)
                Console.WriteLine (x);
        l.sort();                      //Sorts the list values in ascending order
        Console.WriteLine ("Elements in list after sorting:");
        foreach (int x in l)
                Console.WriteLine (x);
}
```

LinkedList<T>

- It is an implementation of Doubly Linked List.
- It uses 5 properties:
  - Previous – Holds the reference of previous node.
  - Value – Hold the value.
  - Next – Holds the reference of next node.
  - First – Holds the reference of first record.
  - Last – Holds the reference of last records



- Methods:
  - AddFirst (value) – Inserts the value at beginning movind the previous first item up and setting its previous property to refer new item.
  - AddLast (value) – Inserts the value at the end by setting new node previous property will hold the reference of previous last node.
  - AddBefore (LinkedListNode<T>node, value) – Inserts value before the node.
  - AddAfter (LinkedListNode<T>node, value) – Inserts value after the node.
  - (both requires to traverse to retrive the node)
  - RemoveFirst () – Removes the first node.
  - RemoveLast () – Removes the last node.
  - Remove (val) – Removes the first occurance node whose value matches with the argument passed.

Ex:

```
Static void Main ()
{
        LinkedList <int> l = new LinkedList <int>
        (); l.AddFirst (5);
        foreach (int x in new int [3] {10, 20, 30})
                l.AddLast (x);
        l.AddBefore (l.Find (20), 15);
        l.AddAfter (l.Find (20),
        25); l.RemoveFirst ();
        l.RemoceLast ();
        l.Remove (25);
        //Displaying
        Console.WriteLine ("The list is:");
        for (LinkedListNode <int> i = l.First ; i != null ; i = i.Next)
                Console.WriteLine (i.Value);
```

Display can also be done using foreach loop.

```
foreach(int x in l)
{
                Console.WriteLine (x);
}
```

Queue <T>

It implements first in first out mechanism an element is inserted at the back and removed from the front.

- Methods:
  - Enqueue (val) – Inserts the value to the back of the queue.
  - Dequeue-removes the element from the front
- Property – Count
  - o Keeps a track of the number of elements in the queue.

Ex:

```
Main ()
{
        Queue <int> q = new Queue <int> ();
        q.Enqueue (10);
        foreach (int x in new int [3] {20,30,40})
                q.Enqueue (x);


        //Display
        foreach (int x in q)
        {
                Console.WriteLine (x);
        }
        Console.WriteLine("first element deleted is {0}",q.Deque());
}
```

Stack <T>

It implements last in first out mechanism an element joins the stack at the top(push(ele)) and leaves the stack at the top(pop()).

- Methods:
  - Push (ele)
  - Pop ()
- Property – Count
  - o Keeps track of number of elements in the stack.

Ex:

```
Main()
{
        Stack <int> s = new Stack <int>
        (); s.Push (10);
        foreach (int x in new int [3] {20,30,40})
                s.Push (x);
        //Display
        foreach (int x in s)
        {
                Console.WriteLine (x);
        }
    Console.WriteLine("popped element deleted is {0}",s.Pop());
    }
```

Dictionary <Tkey, Tvalue>

- Usually List<T> has integer index.But sometimes for mapping we require other types like string double, etc.This type of arrays is known as associated arrays.

  Syntax : Dictionary <Tkey, Tvalue>
- Dictionary maintains 2 arrays, one for keys and one for values.
- Dictionary cannot contain duplicate keys. If we try to call the ADD() method for the key which is already present we get the Exception
- Methods:
  - Add (key, value) – Adds the elements to the dictionary.
    - Syntax : object.Add (key, value)
      If we try to add a key which is already present, we get an exception.
- Internally the dictionary uses sparse data structure when it has plenty of memory.
- The dictionary can grow faster as we insert more elements
- To retrieve we use foreach loop.

  Displaying:

```
foreach (KeyValuePair <String, int> x in d)
{
        Console.WriteLine ("Name = {0} , Age = {1}", x.key, x.value);
}
```

- Properties: It uses 2 properties
  - Key – To retrieve keys.
  - Value – To retrieve values.

Program

```
Class Program
{
    static void Main(string [ ] args)
    {
        Dictionary<string,int>ages=new Dictionary<string,int>();
        ages.Add("john",51);
        ages.Add("ram",41);
        ages.Add("sam",34);
        ages.Add("john",21);//Exception
        ages["john"]=21//overwrites 51 to 21
        Console.WriteLine ("the contents are");
        foreach (KeyValuePair <String, int> x in ages)
        {
            Console.WriteLine ("Name = {0} , Age = {1}", x.key, x.value);
        }
    }
}
```

SortedList <TKey, Tvalue>

- Similar to dictionary but the difference is that keys arrays is always sorted.
- It takes longer time to insert data into a sorted list but retrieval is faster and uses less memory.
- Even SortedList cannot contain duplicate keys
- First the key is sorted in the key array. After sorting, it obtains the index of all the keys. Based on the index, respective values are stored in the value array of same index to ensure that key value pairs remain synchronized.

Program

```
class Program
{
        static void Main(string [ ] args)
        {
                SortedList<string,int>ages=new SortedList <string,int>();
                ages.Add("john",51);
                ages.Add("ram",51);
                ages.Add("john",21);//Exception
                ages["john"]=21//overwrites 51 to 21
                Console.WriteLine ("the contents are");
                foreach (KeyValuePair <String, int> x in ages)
                {
                        Console.WriteLine ("Name = {0} , Age = {1}", x.key, x.value);
                }
        }
}
```

HashSet<T>

Is optimized for performing set operations such as determing set membership and generating the union and interaction of sets.

- Methods:
  - Add (val) – Inserts val to the hash set.
  - UnionWith (Hashset) – Finds union of 2 hash sets and stores to first hash set.
    - **Syntax:** hs1.UnionWith (hs2)
      - Hs1 <- hs1 U hs2
  - IntersectWith (HashSet) – Finds the intersection of 2 hash sets and stores to first hash set.
    - **Syntax:** hs1.IntersectWith (hs2)
      - Hs1 <- hs1 ∩ hs2
  - ExceptWith (hashset) – Finds the difference between 2 hash sets and stores to first hash set.
    - Syntax: hs1 <- hs1 – hs2
    -

We can also determine whether the HashSet<T> collection is superset or subset by using IsSubsetOf(),IsSuperSet(),IsProperSupersetOf methods this returns Boolean values

Program:

```
static void Main()
{
        HashSet <string> a= new List <string> (){“ram”,”sam”,”Kumar”};
        HashSet <string> b= new List <string> (){“ramesh”,”sam”,”Kumar”,”ajay”};
        foreach(string x in a)
        {
                Console.WriteLine(x);
        }
        foreach(string x in b)
        {
                Console.WriteLine(x);
        }
        a.UnionWith(b);
        a.IntersectWith(b
        );
        a.ExceptWith(a);
}
```

5 Write a C# program to perform operations (+,-, ==, ++ and type conversions) on complex numbers using operator overloading. [10]

Solution: Program: 10M

```
Class complex
{
Public int real{get;set;}
Public int img{get;set;}
Public complex(int real,int img)
{
This.real=r;
This.img=I;
}
Public override string ToString()
{
Return $”({this.real}+{this.img}i)”;
}
Public static complex operator +(complex lh, complex rh)
{
Return new complex(lh.r+rh.r,lh.i+rh.i);
}
Public static complex operator -(complex lh, complex rh)
{
Return new complex(lh.r+rh.r,lh.i+rh.i);
}
Public static complex operator ++(complex lh)
```

```
{
arg.value++;
Return arg;
}

Public static bool operator ==(complex lh, complex rh)
{
Return lh.equals(rh);
}
Public static bool operator !=(complex lh, complex rh)
{
Return !(lh.equals(rh));
}
Public static implicit operator complex(int from)
{
Return new complex(from);
}
Public static explicit operator int(complex from)
{
Return from.real;
}
Class example
{
Public static void main()
{
Complex f=new complex(10,4);
Complex s=new complex(5,2);
Complex temp=f+s;
If(temp==2)
{
Console.WriteLine("Comparision after conversion :temp==2");

}
Else
{
Console.WriteLine("Comparision after conversion :temp!=2");
}
}
```

6 Write a program in C# Sharp to display the list of items in the array according to the length of the string then by name in ascending order. [10]

Solution: Program: 10M

```
using System;
using System.Linq;
using System.Collections.Generic;

public class  Linq
{
        public static void Main()
        {
```

```
            string[] cities =
            {
             "ROME","LONDON","NAIROBI","CALIFORNIA","ZURICH","NEW
DELHI","AMSTERDAM","ABU DHABI", "PARIS"
        };

        Console.Write("\nLINQ : Display the list according to the length then by name in
ascending order : ");
        Console.Write("\n----------------------------------------------------------------------\n");

        Console.Write("\nThe cities are :
ROME','LONDON','NAIROBI','CALIFORNIA','ZURICH','NEW
DELHI','AMSTERDAM','ABU DHABI','PARIS' \n");
Console.Write("\nHere is the arranged list :\n");
IEnumerable<string> cityOrder =cities.OrderBy(str => str.Length)ThenBy(str => str);
foreach (string city in cityOrder)
Console.WriteLine(city);
Console.ReadLine();
}
}
```

7 Create container Employees for employee class with the following data fname, lname, empid, salary, company name and perform the following operation on it.

i) Add employee
ii) Select employee in list by fname and lname
iii) group employee by company name
iv) Count the No. of employees in the container
v) Display all the employee name

Solution: Program: 10M

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
public class Programs
{
   public class Employee
   {
      public string First { get; set; }
      public string Last { get; set; }
      public int ID { get; set; }
      public int Marks;
      public ContactInfo GetContactInfo(Programs pg, int id)
```

```
        {
            ContactInfo allinfo =
                (from ci in pg.contactList
                 where ci.ID == id
                 select ci)
                .FirstOrDefault();

            return allinfo;
        }

        public override string ToString()
        {
            return First + "" + Last + " :  "    + ID;
        }
    }

    public class ContactInfo
    {
        public int ID { get; set; }
        public string companyname { get; set; }
        public string salary { get; set; }
        public override string ToString() { return companyname
+ "," + salary; }
    }

    public class ScoreInfo
    {
        public double Average { get; set; }
        public int ID { get; set; }
    }
    List<Employee> students = new List<Employee>()
        {
            new Employee {First="Tom", Last=".S", ID=1 },
            new Employee {First="Jerry", Last=".M", ID=2},
            new Employee {First="Bob", Last=".P", ID=3},
            new Employee {First="Mark", Last=".G", ID=4},
        };

        List<ContactInfo> contactList = new
List<ContactInfo>()
        {
            new ContactInfo {ID=111, companyname="Infosys",
salary="93000"},
            new ContactInfo {ID=112,
```

```
companyname="Microsoft", salary="90000"},
        new ContactInfo {ID=113,
companyname="VMWARE", salary="153000"},
        new ContactInfo {ID=114,
companyname="Microsoft", salary="300000"}
    };

  public static void Main(string[] args)
  {
    Programs pg = new Programs();
            List<Employee> s = new List<Employee>();
    s.Add(new Employee() {First="John",
Last="sharp",ID=1234});

            List<ContactInfo> c = new List<ContactInfo>();
            c.Add(new ContactInfo() { ID=115,
companyname="Microsoft", salary="350000"});

    IEnumerable<Employee> Query2
=Employee.Select(cust=>cust.First);
             foreach (Employee s1 in Query2)
    {
      Console.WriteLine(s.ToString());
    }

            IEnumerable<ContactInfo> Query3
=ContactInfo.GroupBy(cu=>cu.companyname);
            foreach (ContactInfo s2 in Query3)
    {
      Console.WriteLine(s.ToString());
    }

            int n=Employee.Select(ee=>ee.ID).Count();
            Console.WriteLine($"Number of
Employees:{n}");

            foreach(var emp in Employee)
            {
                Console.WriteLine(emp);
            }

    Console.ReadLine();
  }
}
```