LaSalle College
Montréal

Final Project: 30%

Course Identification

Name of programs – Codes:	COMPUTER SCIENCE TECHNOLOGY
Course title:	**DATA STRUCTURE**
Course number:	420-SD4-AS
Group:	07264 - 07266
Teachers' names:	Glauber Torres
Duration:	2 weeks
Semester:	Winter 2022

Student Identification

Name: ________________________________ Student number: ______________

Date: March 22, 2022 Result: ______________

☒ I declare that this is an original work, and that I credited all content sources of which I am not the author (online and printed, images, graphics, films, etc.), in the required quotation and citation style for this work.

Standard of the Evaluated Competencies

Statement of the evaluated competency – Code
Use an object-oriented development approach – 016T

Evaluated elements of the competencies
Program a class.
Correct programs – 0171

Evaluated elements of the competencies
Determine the nature of the problem.
Correct the problem.

Part I Stack Dynamic Array (30 points)

Given the following interface and class:

```
public interface IStack<T>
{
    void Push(T item);
    T Peek();
    T Pop();
    void Clear();
    bool Contains(T item);
    int Count();
}

public class StackArray<T> : IStack<T>
{
    protected T[] array;
    protected int count;

    //*** TODO Time complexity:
    public StackArray(int size)
    {
        array = new T[size];
        count = 0;
    }

    //*** TODO Time complexity:
    public void Clear()
    {
        for (int i = 0; i < count; i++)
        {
            array[i] = default(T);
        }
        count = 0;
    }

    //*** TODO Time complexity:
    public bool Contains(T item)
    {
        for (int i = 0; i < count; i++)
        {
            if (array[i].Equals(item))
            {
                return true;
            }
        }
        return false;
    }

    //*** TODO Time complexity:
    public int Count()
    {
        return count;
    }
    //*** TODO Time complexity:
    public virtual void Push(T item)
    {
        if (count == array.Length)
            throw new IndexOutOfRangeException();
        array[count] = item;
        count++;
    }

    //*** TODO Time complexity:
    public T Peek()
    {
        if (count == 0)
            throw new
            InvalidOperationException("Stack empty");

        int last_index = count - 1;
        return array[last_index];
    }

    //*** TODO Time complexity:
    public T Pop()
    {
        if (count == 0)
            throw new
            InvalidOperationException("Stack empty");

        int last_index = count - 1;
        T item = array[last_index];
        array[last_index] = default(T);
        count--;
        return item;
    }
}
```

Question 1 (3 points)
Create a StackDynamicArray<T> class that inherits from Stack<T>

Question 2 (5 points)
In the StackDynamicArray<T> class, call the base constructor StackArray(int) and pass the value 2.

Question 3 (15 points)
In the StackDynamicArray<T> class, override the Push method so that whenever you try to push an element and the internal array is full, double it. Provide the time complexity for this override Push method.

Question 4 (7 points)
Provide the time complexity for all asked members in the Stack<T> class.

Part II Linked List (70 points)

Given the following interface and classes:

```
public class GenericNode<T>
{
    private T data;
    private GenericNode<T> next;

    public GenericNode(T initialData)
    {
        data = initialData;
        next = null;
    }

    public T Data { get => data; set => data = value; }
    public GenericNode<T> Next { get => next; set => next = value; }

    public static int Compare(string left, string right)
    {
        int leftNumeric;
        int rightNumeric;
        if (int.TryParse(left, out leftNumeric) &&
            int.TryParse(right, out rightNumeric))
        {
            if (leftNumeric > rightNumeric)
                return 1;
            else if (leftNumeric == rightNumeric)
                return 0;
            else
                return -1;
        }
        return string.Compare(left, right);
    }

    public static bool operator <(GenericNode<T> left, GenericNode<T> right)
    {
        return Compare(left.Data.ToString(), right.Data.ToString()) < 0;
    }

    public static bool operator >(GenericNode<T> left, GenericNode<T> right)
    {
        return Compare(left.Data.ToString(), right.Data.ToString()) > 0;
    }

    public static bool operator <=(GenericNode<T> left, GenericNode<T> right)
    {
        return Compare(left.Data.ToString(), right.Data.ToString()) < 0;
    }

    public static bool operator >=(GenericNode<T> left, GenericNode<T> right)
    {
        return Compare(left.Data.ToString(), right.Data.ToString()) > 0;
    }
}
```

```csharp
public interface ILinkedList<T>
{
    int Count { get; set; }
    GenericNode<T> Head { get; set; }
    void AddFirst(GenericNode<T> node);
    GenericNode<T> AddFirst(T data);
    void AddLast(GenericNode<T> node);
    GenericNode<T> AddLast(T data);
    void AddBefore(GenericNode<T> node, GenericNode<T> newNode);
    GenericNode<T> AddBefore(GenericNode<T> node, T data);
    void AddSorted(GenericNode<T> newNode);
    GenericNode<T> AddSorted(T data);
    GenericNode<T> RemoveByValue(T data);
    GenericNode<T> RemoveByIndex(int index);
    string ToString();
}

public class LinkedList<T> : ILinkedList<T>
{
    private GenericNode<T> head;
    private int count;

    public int Count { get => count; set => count = value; }
    public GenericNode<T> Head { get => head; set => head = value; }

    //*** TODO Time complexity:
    public LinkedList()
    {
        head = null;
        count = 0;
    }

    //*** TODO Time complexity:
    public void AddFirst(GenericNode<T> node)
    {
        node.Next = head;
        head = node;
        count++;
    }

    //*** TODO Time complexity:
    public GenericNode<T> AddFirst(T data)
    {
        GenericNode<T> node = new GenericNode<T>(data);
        AddFirst(node);
        return node;
    }

    //*** TODO Time complexity:
    public GenericNode<T> AddLast(T data)
    {
        GenericNode<T> node = new GenericNode<T>(data);
        AddLast(node);
        return node;
    }
```

```
//*** TODO Time complexity:
public void AddLast(GenericNode<T> node)
{
    if (head == null)
    {
        AddFirst(node);
        return;
    }
    GenericNode<T> current = head;
    while (current.Next != null)
    {
        current = current.Next;
    }
    current.Next = node;
    count++;
}

//*** TODO Time complexity:
public void AddBefore(GenericNode<T> node, GenericNode<T> newNode)
{
    //TODO PART 2 - QUESTION 1 - Implement here the AddBefore method:
}

//*** TODO Time complexity:
public GenericNode<T> AddBefore(GenericNode<T> node, T data)
{
    GenericNode<T> newNode = new GenericNode<T>(data);
    AddBefore(node, newNode);
    return newNode;
}

//*** TODO Time complexity:
public void AddSorted(GenericNode<T> newNode)
{
    //TODO PART 2 - QUESTION 2 - Implement here the AddSorted method:
}

//*** TODO Time complexity:
public GenericNode<T> AddSorted(T data)
{
    GenericNode<T> newNode = new GenericNode<T>(data);
    AddSorted(newNode);
    return newNode;
}

//*** TODO Time complexity:
public GenericNode<T> RemoveByIndex(int index)
{
    //TODO PART 2 - QUESTION 3: Remove the following line and
    //implement the RemoveByIndex method here:
    throw new NotImplementedException();
}
```

```
    //*** TODO Time complexity:
    public GenericNode<T> RemoveByValue(T data)
    {
        //TODO PART 2 - QUESTION 4 - Fix the RemoveByValue method.
        GenericNode<T> current = head;
        GenericNode<T> previous = null;
        while (current != null) {
            if (current.Data.Equals(data))
            {
                previous.Next = current.Next;
                count--;
                return current;
            }
            previous = current;
            current = current.Next;
        }
        return null;
    }

    //*** TODO Time complexity:
    public override string ToString()
    {
        var sb = new System.Text.StringBuilder();
        var current = head;
        while (current != null) {
            if (sb.Length > 0)
                sb.Append(", ");
            sb.Append(current.Data);
            current = current.Next;
        }
        return sb.ToString();
    }
}
```

Question 1 (15 points)
Implement the method **AddBefore**(GenericNode<T> node, GenericNode<T> newNode). The newNode parameter should be added before the node parameter in the LinkedList.

Question 2 (20 points)
Implement the method **AddSorted**(GenericNode<T> newNode). This method should add newNode right before the first node that is greater than newNode. If no node is greater than newNode or the LinkedList is empty then newNode should be the head node.

Question 3 (15 points)
Implement the **RemoveByIndex**(int index). Consider the index to be zero based. For example: if you execute RemoveByIndex(0), it means you will remove the head node. Do not forget to return the removed node.

Question 4 (10 points)
Fix the **RemoveByValue**(T data) method to be able to remove the head node. Currently this method will not work if you pass the value that corresponds to the head node. Do not forget to return the removed node.

Question 5 (10 points)
Provide the time complexity for all asked members in the LinkedList<T> class.