# COIS 2020H-Data Structures & Algorithms

Winter 2024

**Assignment 2 (15 %)**

# Submission template.

**YOUR NAME: Dikshith Reddy Macherla STUDENT ID: 0789055**

**Fill out the following tables**

1. ArrayList. One box for each method

|  |
| --- |
| **Paste your code for the constructor**  using System;  // Generic ArrayList class that simulates the behavior of a dynamic array.  public class ArrayList<T>  {  private T[] items; // Internal array to store the elements.  private int count; // Current number of elements in the ArrayList.  // Constructor to initialize the ArrayList with a default capacity.  // This can create an empty list.  public ArrayList(int initialCapacity = 4)  {  if (initialCapacity < 1)  {  throw new ArgumentException("Initial capacity must be at least 1.", nameof(initialCapacity));  }  items = new T[initialCapacity]; // Set the initial capacity of the array.  count = 0; // Initially, the list has no elements.  }  // A property to expose the count of elements in the ArrayList.  // This property is read-only from outside the class.  public int Count  {  get { return count; }  }  // Ensures the capacity of the internal array.  // This method is called by methods that add elements to the ArrayList when the internal array is full.  private void Grow()  {  int newCapacity = items.Length \* 2; // Double the current capacity.  T[] newItems = new T[newCapacity]; // Create a new array with the new capacity.  Array.Copy(items, newItems, count); // Copy the elements from the old array to the new one.  items = newItems; // Set the internal array to the new array.  }  // Other methods such as AddFront, AddLast, etc., will be implemented here.  } |
| **Paste your code for the Private Grow:**  private void Grow()  {  // Calculate the new capacity, which is double the current capacity.  // This approach helps in keeping the amortized time complexity of adding an element in check.  int newCapacity = items.Length \* 2;    // Create a new array with the new capacity.  T[] newItems = new T[newCapacity];    // Copy the elements from the old array to the new one.  // This is necessary because the underlying storage is being replaced.  Array.Copy(items, newItems, count);    // Update the reference of the internal array to point to the new array.  items = newItems;  } |
| **Paste your code for the AddFront:**  // Adds an item at the front of the ArrayList, shifting all other elements one position towards the end.  public void AddFront(T item)  {  // If the internal array is full, increase its capacity first.  if (count == items.Length)  {  Grow();  }  // Shift all existing elements one position to the right to make space at the front.  // This loop starts from the end to avoid overwriting any elements.  for (int i = count; i > 0; i--)  {  items[i] = items[i - 1];  }  // Insert the new item at the front (which is now the 0th index).  items[0] = item;  // Increase the count of elements in the list.  count++;  } |
| **Paste your code for the AddLast:**  // Adds an item at the end of the ArrayList.  public void AddLast(T item)  {  // Check if the internal array is full and needs to grow to accommodate more elements.  if (count == items.Length)  {  Grow(); // Call the private Grow method to double the size of the array.  }  // Add the new item at the position indicated by the current count of elements.  // The count also represents the index where the new element should be inserted  // since array indexes start at 0 and count starts at 1 for the first element.  items[count] = item;  // Increment the count to reflect the addition of a new element.  count++;  } |
| **Paste your code for the GetCount:**  // Returns the number of items currently in the ArrayList.  public int GetCount()  {  return count; // Return the current count of elements in the list.  }  // A property to expose the count of elements in the ArrayList.  // This property is read-only from outside the class.  public int Count  {  get { return count; }  } |
| **Paste your code for the InsertBefore**:  // Inserts an item before the specified target item in the ArrayList.  public void InsertBefore(T newItem, T targetItem)  {  // Check if the array needs to grow to accommodate the new item.  if (count == items.Length) Grow();  // Find the index of the target item.  int targetIndex = Array.IndexOf(items, targetItem, 0, count);    // If the target item is not found, do nothing.  if (targetIndex == -1) return;  // Shift elements to the right starting from the targetIndex to make space for the new item.  for (int i = count; i > targetIndex; i--)  {  items[i] = items[i - 1];  }  // Insert the new item at the targetIndex.  items[targetIndex] = newItem;  // Increment the count of elements in the list.  count++;  } |
| **Paste your code for the InPlaceSort:**  // Sorts the ArrayList in place.  public void InPlaceSort()  {  // Sort the portion of the array that contains elements.  Array.Sort(items, 0, count);  } |
| **Paste your code for the Swap(index1, index2):**  // Swaps two elements at the specified indices.  public void Swap(int index1, int index2)  {  // Validate indices.  if (index1 < 0 || index1 >= count || index2 < 0 || index2 >= count)  {  throw new ArgumentOutOfRangeException("Indices must be within the bounds of the list.");  }  // Swap the elements.  T temp = items[index1];  items[index1] = items[index2];  items[index2] = temp;  } |
| **Paste your code for the DeleteFirst:**  // Removes the first element from the ArrayList, shifting all other elements to the left.  public void DeleteFirst()  {  if (count == 0) return; // If the list is empty, do nothing.  // Shift all elements one position to the left.  for (int i = 0; i < count - 1; i++)  {  items[i] = items[i + 1];  }  // Nullify the last element to avoid holding onto an object reference unnecessarily.  items[count - 1] = default(T);  // Decrement the count to reflect the removal.  count--;  } |
| **Paste your code for the DeleteLast:**  // Removes the last element from the ArrayList.  public void DeleteLast()  {  if (count == 0) return; // If the list is empty, do nothing.  // Nullify the last element to avoid holding onto an object reference unnecessarily.  items[count - 1] = default(T);  // Decrement the count to reflect the removal.  count--;  } |
| **Paste your code for the RotateLeft:**  // Rotates all elements in the ArrayList one position to the left.  public void RotateLeft()  {  if (count <= 1) return; // No need to rotate if list is empty or contains only one element.  // Store the first element.  T first = items[0];  // Shift all elements one position to the left.  for (int i = 0; i < count - 1; i++)  {  items[i] = items[i + 1];  }  // Move the first element to the end of the ArrayList.  items[count - 1] = first;  } |
| **Paste your code for the RotateRight:**  // Rotates all elements in the ArrayList one position to the right.  public void RotateRight()  {  if (count <= 1) return; // No need to rotate if list is empty or contains only one element.  // Store the last element.  T last = items[count - 1];  // Shift all elements one position to the right.  for (int i = count - 1; i > 0; i--)  {  items[i] = items[i - 1];  }  // Move the last element to the beginning of the ArrayList.  items[0] = last;  } |
| **Paste your code for the Merge:**  // Merges two ArrayLists into a new one without sorting.  public static ArrayList<T> Merge(ArrayList<T> list1, ArrayList<T> list2)  {  var mergedList = new ArrayList<T>(list1.count + list2.count); // Initialize with enough capacity.  foreach (var item in list1.items.Take(list1.count))  {  mergedList.AddLast(item); // Add items from the first list.  }  foreach (var item in list2.items.Take(list2.count))  {  mergedList.AddLast(item); // Add items from the second list.  }  return mergedList;  } |
| **Paste your code for the StringPrintAllForward:**  // Returns a string representation of the ArrayList from beginning to end.  public string StringPrintAllForward()  {  if (count == 0) return "The list is empty.";  var builder = new StringBuilder();  for (int i = 0; i < count; i++)  {  builder.Append(items[i].ToString() + (i < count - 1 ? ", " : ""));  }  return builder.ToString();  } |
| **Paste your code for the StringPrintAllReverse:**  // Returns a string representation of the ArrayList from end to beginning.  public string StringPrintAllReverse()  {  if (count == 0) return "The list is empty.";  var builder = new StringBuilder();  for (int i = count - 1; i >= 0; i--)  {  builder.Append(items[i].ToString() + (i > 0 ? ", " : ""));  }  return builder.ToString();  } |
| **Paste your code for the Deleteall**  // Clears the ArrayList, effectively removing all elements.  public void DeleteAll()  {  // Loop is not strictly necessary; directly setting count to 0 and relying on  // garbage collection for cleanup is usually sufficient. However, explicitly nullifying  // references can help with memory management in certain scenarios.  for (int i = 0; i < count; i++)  {  items[i] = default(T); // Help with garbage collection by releasing references.  }  count = 0; // Reset the count, effectively clearing the list.  } |

1. LinkedList One box for each method

|  |
| --- |
| **Paste your code for the constructor**  public class DoublyLinkedListNode<T>  {  public T Value { get; set; }  public DoublyLinkedListNode<T> Next { get; set; }  public DoublyLinkedListNode<T> Previous { get; set; }  public DoublyLinkedListNode(T value)  {  Value = value;  Next = null;  Previous = null;  }  }  public class DoublyLinkedList<T>  {  private DoublyLinkedListNode<T> head;  private DoublyLinkedListNode<T> tail;  private int count;  // Constructor initializes an empty DoublyLinkedList.  public DoublyLinkedList()  {  head = null;  tail = null;  count = 0;  }  public int Count => count;  // Methods for AddFront, AddLast, etc., will follow here.  } |
| **Paste your code for the AddFront:**  public void AddFront(T item)  {  var newNode = new DoublyLinkedListNode<T>(item);  if (head == null)  {  // The list is empty, so the new node becomes both head and tail.  head = newNode;  tail = newNode;  }  else  {  // Link the new node with the current head and update the head to be the new node.  newNode.Next = head;  head.Previous = newNode;  head = newNode;  }  count++;  } |
| **Paste your code for the AddLast:**  public void AddLast(T item)  {  var newNode = new DoublyLinkedListNode<T>(item);  if (tail == null)  {  // The list is empty, so the new node becomes both head and tail.  head = newNode;  tail = newNode;  }  else  {  // Link the new node with the current tail and update the tail to be the new node.  tail.Next = newNode;  newNode.Previous = tail;  tail = newNode;  }  count++;  } |
| **Paste your code for the GetCount:**  public int Count => count; |
| **Paste your code for the InsertAtRandomLocation**:  public void InsertAtRandomLocation(T item)  {  var newNode = new DoublyLinkedListNode<T>(item);  var random = new Random();  int position = random.Next(count + 1); // +1 to include the possibility of insertion at the end.  if (position == 0)  {  AddFront(item);  }  else if (position == count)  {  AddLast(item);  }  else  {  var current = head;  for (int i = 0; i < position - 1; i++) // Move to the node just before the insertion point.  {  current = current.Next;  }  // Insert the new node.  newNode.Next = current.Next;  newNode.Previous = current;  current.Next.Previous = newNode;  current.Next = newNode;  count++;  }  } |
| **Paste your code for the Merge**  public void Merge(DoublyLinkedList<T> otherList)  {  if (otherList.count == 0) return; // Nothing to merge if the other list is empty.  if (count == 0)  {  // If the current list is empty, just set head and tail to the other list's.  head = otherList.head;  tail = otherList.tail;  }  else  {  // Connect the tail of this list to the head of the other list.  tail.Next = otherList.head;  otherList.head.Previous = tail;  tail = otherList.tail;  }  count += otherList.count;  // Optionally clear the other list if it should not be used after merging.  otherList.head = null;  otherList.tail = null;  otherList.count = 0;  } |
| **Paste your code for the FindClosest**  // This method is conceptual and needs to be adjusted based on the actual properties of T.  public T FindClosest(Position position)  {  DoublyLinkedListNode<T> closest = null;  double minDistance = double.MaxValue;  var current = head;  while (current != null)  {  double distance = FindDistance(position, current.Value); // Assumes a method to calculate distance exists.  if (distance < minDistance)  {  minDistance = distance;  closest = current;  }  current = current.Next;  }  return closest?.Value;  } |
| **Paste your code for the FindDistance**  // Conceptual method, adjust based on the actual properties of T.  public double FindDistance(Position position, T item)  {  // Example calculation assuming T has X, Y, Z properties or similar.  // You'll need to access the actual position properties of item.  var itemPosition = /\* obtain position from item \*/;  return Math.Sqrt(Math.Pow(position.X - itemPosition.X, 2) + Math.Pow(position.Y - itemPosition.Y, 2) + Math.Pow(position.Z - itemPosition.Z, 2));  } |
| **Paste your code for the DeleteFirst**  public void DeleteFirst()  {  if (head == null) return; // List is empty.  // If there's only one item, clear the list.  if (head == tail)  {  head = null;  tail = null;  }  else  {  head = head.Next;  head.Previous = null;  }  count--;  } |
| **Paste your code for the DeleteLast**  public void DeleteLast()  {  if (tail == null) return; // List is empty.  // If there's only one item, clear the list.  if (head == tail)  {  head = null;  tail = null;  }  else  {  tail = tail.Previous;  tail.Next = null;  }  count--;  } |
| **Paste your code for the GetEaten**  public void GetEaten(T target)  {  DoublyLinkedListNode<T> current = head;  while (current != null)  {  if (Equals(current.Value, target)) // Assumes T implements Equals appropriately.  {  if (current == head) { DeleteFirst(); return; }  if (current == tail) { DeleteLast(); return; }  // Link the previous and next nodes together, effectively removing 'current' from the chain.  current.Previous.Next = current.Next;  current.Next.Previous = current.Previous;  count--;  return;  }  current = current.Next;  }  } |
| **Paste your code for the RotateLeft**  public void RotateLeft()  {  if (count <= 1) return; // No need to rotate if the list has 0 or 1 element.  DoublyLinkedListNode<T> formerHead = head;    // Move head to the next element.  head = head.Next;  head.Previous = null;  // Move the former head to the tail.  tail.Next = formerHead;  formerHead.Previous = tail;  formerHead.Next = null;  tail = formerHead;  } |
| **Paste your code for the RotateRight**  public void RotateRight()  {  if (count <= 1) return; // No need to rotate if the list has 0 or 1 element.  DoublyLinkedListNode<T> formerTail = tail;  // Move tail to the previous element.  tail = tail.Previous;  tail.Next = null;  // Move the former tail to the head.  formerTail |
| **Paste your code for the StringPrintAllForward**  // Returns a string representation of the DoublyLinkedList from beginning to end.  public string StringPrintAllForward()  {  if (head == null) return "The list is empty.";  var builder = new StringBuilder();  var current = head;  while (current != null)  {  builder.Append(current.Value.ToString() + (current.Next != null ? ", " : ""));  current = current.Next;  }  return builder.ToString();  } |
| **Paste your code for the StringPrintAllReverse**  // Returns a string representation of the DoublyLinkedList from end to beginning.  public string StringPrintAllReverse()  {  if (tail == null) return "The list is empty.";  var builder = new StringBuilder();  var current = tail;  while (current != null)  {  builder.Append(current.Value.ToString() + (current.Previous != null ? ", " : ""));  current = current.Previous;  }  return builder.ToString();  } |
| **Paste your code for the DeleteAll**  // Deletes all elements from the DoublyLinkedList.  public void DeleteAll()  {  head = null;  tail = null;  count = 0;  } |

1. Main:

|  |
| --- |
| **Paste your code for creating two Arraylists and merging them**  using System;  using System.Collections.Generic;  class Program  {  static void Main()  {  // Create ArrayLists for Cats and Snakes  var cats = new ArrayList<Animal>();  var snakes = new ArrayList<Animal>();  // Populate Cats ArrayList  for (int i = 0; i < 3; i++)  {  var cat = new Cat();  cat.GenerateRandomPosition(-25, 25); // Generate random position  cats.AddFront(cat);  }  // Populate Snakes ArrayList  for (int i = 0; i < 3; i++)  {  var snake = new Snake();  snake.GenerateRandomPosition(-25, 25); // Generate random position  snakes.AddLast(snake);  }  // Merge the two ArrayLists  var mergedList = ArrayList<Animal>.Merge(cats, snakes);  // Test PrintAllForward on the merged ArrayList  Console.WriteLine("Merged ArrayList (Forward):");  Console.WriteLine(mergedList.StringPrintAllForward());  // Test PrintAllReverse on the merged ArrayList  Console.WriteLine("\nMerged ArrayList (Reverse):");  Console.WriteLine(mergedList.StringPrintAllReverse());  }  } |
| **Paste your code for creating two LinkedLists and merging them**  using System;  class Program  {  static void Main()  {  // Create DoublyLinkedLists for the first 5 birds and the remaining 5 birds  var firstFiveBirds = new DoublyLinkedList<Bird>();  var remainingBirds = new DoublyLinkedList<Bird>();  // Populate the first DoublyLinkedList with the first 5 birds  string[] birdNames = { "Tweety", "Zazu", "Iago", "Hula", "Manu" };  for (int i = 0; i < 5; i++)  {  var bird = new Bird(birdNames[i]);  // Add bird to the front of the list  firstFiveBirds.AddFront(bird);  }  // Populate the second DoublyLinkedList with the remaining 5 birds  for (int i = 5; i < 10; i++)  {  var bird = new Bird(birdNames[i - 5]); // Using same bird names as the first 5  // Add bird to the front of the list  remainingBirds.AddFront(bird);  }  // Merge the second DoublyLinkedList onto the first one  firstFiveBirds.Merge(remainingBirds);  // Print the contents of the merged list  Console.WriteLine("Merged DoublyLinkedList:");  Console.WriteLine(firstFiveBirds.StringPrintAllForward());  }  } |
| **Paste your code for the while loop and any other method you have created to serve the while loop procedure (if any)**  using System;  class Program  {  static void Main()  {  // Create ArrayLists for Cats, Snakes, and Birds  var cats = new ArrayList<Animal>();  var snakes = new ArrayList<Animal>();  var birds = new ArrayList<Animal>();  // Populate Cats ArrayList  for (int i = 0; i < 3; i++)  {  var cat = new Cat();  cat.GenerateRandomPosition(-25, 25); // Generate random position  cats.AddFront(cat);  }  // Populate Snakes ArrayList  for (int i = 0; i < 3; i++)  {  var snake = new Snake();  snake.GenerateRandomPosition(-25, 25); // Generate random position  snakes.AddLast(snake);  }  // Populate Birds ArrayList  string[] birdNames = { "Tweety", "Zazu", "Iago", "Hula", "Manu", "Couscous", "Roo", "Tookie", "Plucky", "Jay" };  for (int i = 0; i < 10; i++)  {  var bird = new Bird(birdNames[i]);  bird.GenerateRandomPosition(-100, 100, 0, 10); // Generate random position  birds.AddLast(bird);  }  // Merge the Cats and Snakes ArrayLists  var animals = ArrayList<Animal>.Merge(cats, snakes);  // Merge the Birds ArrayList onto the merged Cats and Snakes ArrayList  animals.Merge(birds);  // Counter for the number of rounds  int roundCount = 0;  // While loop to simulate the procedure  while (birds.GetCount() > 0)  {  roundCount++;  // Iterate over the animals list  var current = animals.GetFirstNode();  while (current != null)  {  // If the current animal is a Cat or a Snake  if (current.Value is Cat || current.Value is Snake)  {  // Check for nearby birds to eat  foreach (var bird in birds)  {  double distance = CalculateDistance(current.Value.Position, bird.Position);  if (current.Value is Cat && distance <= 8)  {  Console.WriteLine($"{current.Value.Name} is eating {bird.Name}");  birds.Delete(bird);  }  else if (current.Value is Snake && distance <= 3)  {  Console.WriteLine($"{current.Value.Name} is eating {bird.Name}");  birds.Delete(bird);  }  }  // If no birds are nearby, move towards the nearest bird  // Cats move at a speed of 16, and Snakes move at a speed of 14  var nearestBird = birds.FindClosest(current.Value.Position);  if (nearestBird != null)  {  double distanceToBird = CalculateDistance(current.Value.Position, nearestBird.Position);  double deltaX = (nearestBird.Position.X - current.Value.Position.X) / distanceToBird \* (current.Value is Cat ? 16 : 14);  double deltaY = (nearestBird.Position.Y - current.Value.Position.Y) / distanceToBird \* (current.Value is Cat ? 16 : 14);  current.Value.Position.X += deltaX;  current.Value.Position.Y += deltaY;  }  }  // If the current animal is a Bird, move randomly  else if (current.Value is Bird)  {  RandomMove(current.Value);  }  current = current.Next;  }  // Print the list every fifth iteration  if (roundCount % 5 == 0)  {  Console.WriteLine($"\nAfter {roundCount} rounds:");  Console.WriteLine(animals.StringPrintAllForward());  }  }  Console.WriteLine($"\nAll birds have been eaten in {roundCount} rounds.");  }  // Method to calculate the distance between two positions  static double CalculateDistance(Position pos1, Position pos2)  {  return Math.Sqrt(Math.Pow(pos1.X - pos2.X, 2) + Math.Pow(pos1.Y - pos2.Y, 2) + Math.Pow(pos1.Z - pos2.Z, 2));  }  // Method to move an animal randomly  static void RandomMove(Animal animal)  {  Random random = new Random();  double deltaX = random.Next(-10, 11);  double deltaY = random.Next(-10, 11);  double deltaZ = random.Next(-2, 3);  // Clamping movement within the specified range  animal.Position.X = Math.Max(-100, Math.Min(100, animal.Position.X + deltaX));  animal.Position.Y = Math.Max(-100, Math.Min(100, animal.Position.Y + deltaY));  animal.Position.Z = Math.Max(0, Math.Min(10, animal.Position.Z + deltaZ));  }  } |
| **Screenshot of the output of** PrintallForward**. Make sure to provide the results before the while loop, and during the while loop. [**You don’t have to show all PrintallForward results, one before the while, one at the end of the loop, and 2-5 within the while loop is enough**]** |

After filling out the table, save this document as a word file AND as a pdf file. Then Submit BOTH files along with your C# project directory, as described in the main assignment document.