|  |  |
| --- | --- |
| Name:Vraj Chauhan | Username: vrajchauhan |

Part 1: Arraylist. Code + Testing for your methods, you may want to group together some of these. You can test this with primitives or some simple object.

|  |
| --- |
| **FindIndex(T item)**  /// <summary>  ///Gets the index of the first occurrence of the specified item in the list.  /// </summary>  ///  /// <param name="item">The item to find.</param>  ///  /// <returns>The index of the first occurrence of the specified item, or -1 if the item is not found.</returns>  public int FindIndex(T item)  {  for (int i = 0; i < Count; i++)  {  if (buffer[i].Equals(item))  {  return i;  }  }  return -1;  } |
| **RemoveAt(int index)**  /// <summary>  /// Removes the item at the specified index. All items after the specified index should be shifted one position to the left.  /// </summary>  ///  /// <param name="index">The index of the item to remove.</param>  ///  /// <returns>The item that was removed.</returns>  ///  /// <exception cref="ArgumentOutOfRangeException">  /// If <paramref name="index"/> is less than zero or greater than or equal to <see cref="Count"/>.  /// </exception>  public T RemoveAt(int index)  {  if (index < 0 || index >= Count)  throw new ArgumentOutOfRangeException(nameof(index));  T remove = buffer[index];  for (int i = index; i < Count - 1; i++)  {  buffer[i] = buffer[i + 1];  }  Count--;  return remove;  } |
| **RemoveFirst()** and **RemoveLast()**.  /// <summary>  /// Removes the first item from the list. If the item is not found, the list should remain unchanged.  /// </summary>  ///  /// <param name="item">The item to remove.</param>  ///  public T RemoveFirst()  {  return RemoveAt(0);  }  /// <summary>  /// Removes the last item from the list. If the item is not found, the list should remain unchanged.  /// </summary>  ///  public T RemoveLast()  {  return RemoveAt(Count - 1);  } |
| **Clear()**.  /// <summary>  /// clears the list of all items.  /// </summary>  public void Clear()  {  for (int i = 0; i < Count; i++)  {  buffer[i] = default!;  }  Count = 0;  } |
| **Swap(int i, int j)**.  /// <summary>  /// Swaps the items at the specified indices.  /// </summary>  ///  /// <param name="index1">The index of the first item to swap.</param>  /// <param name="index2">The index of the second item to swap.</param>  ///  /// <exception cref="ArgumentOutOfRangeException">  /// If <paramref name="index1"/> is less than zero or greater than or equal to <see cref="Count"/>.  /// </exception>  ///  public void Swap(int index1, int index2)  {  if (index1 < 0 || index1 >= Count)  throw new ArgumentOutOfRangeException(nameof(index1));  if (index2 < 0 || index2 >= Count)  throw new ArgumentOutOfRangeException(nameof(index2));  T temp = buffer[index1];  buffer[index1] = buffer[index2];  buffer[index2] = temp;  } |
| **RotateLeft()** and **RotateRight()**.  /// <summary>  /// rotates the list to the left by one position.  /// </summary>  public void RotateLeft()  {  if (Count <= 1)  return;  T firstItem = RemoveAt(0);  AddBack(firstItem);  }  /// <summary>  /// rotates the list to the right by one position.  /// </summary>  public void RotateRight()  {  if (Count <= 1)  return;  T lastItem = RemoveAt(Count - 1);  AddFront(lastItem);  } |
| **Sort()**. Make sure to discuss here, above or below your code what sorting method you used, why, and anything about how you chose parameters etc. Sorting Method Chosen: QuickSort QuickSort is chosen because it is a highly efficient and widely used sorting algorithm, particularly well-suited for large datasets due to its average-case time complexity of O( n log(n)). Also I wanted to test and see If I’m able to implement it because it was on those algorithms which I was not so comfortable at first. It is also an in-place sort that has good space complexity() and has good cache performance.  To accommodate sorting in both ascending and descending order, I added a parameter called reverse that has a default value of false so that without this parameter the list would be sorted in ascending order.  /// <summary>  /// Sorts the list in ascending order by default. If reverse is true, sorts the list in descending order.  /// </summary>  /// <param name="reverse">If true, the list should be sorted in descending order.</param>  public void Sort(bool reverse = false)  {  if (Count <= 1)  return;  QuickSort(0, Count - 1, reverse);  }  private void QuickSort(int left, int right, bool reverse)  {  if (left < right)  {  int pivot = Partition(left, right, reverse);  QuickSort(left, pivot - 1, reverse);  QuickSort(pivot + 1, right, reverse);  }  }  private int Partition(int left, int right, bool reverse)  {  T pivot = buffer[right];  int i = left - 1;  for (int j = left; j < right; j++)  {  if (reverse)  {  if (buffer[j].CompareTo(pivot) > 0)  {  i++;  Swap(i, j);  }  }  else  {  if (buffer[j].CompareTo(pivot) < 0)  {  i++;  Swap(i, j);  }  }  }  Swap(i + 1, right);  return i + 1;  } |

Part 2: The main Animating Wizards and Goblins

You will want a video of it running. The windows snipping tool (in windows 11) will take short videos but OBS is probably better. You may want to clearly label what you’re showing in certain parts of the testing.

|  |
| --- |
| Code for Goblin to attack a wizard in range  In goblin class properties:  public float AttackRange { get; set; } = 1.0f; // Goblins have a 1.0f attack range.  In game.cs  protected override void Update()  {  foreach (Goblin goblin in Goblins)  {  updateGoblin(goblin);  }  protected void updateGoblin(Goblin goblin)  {  if (goblin.CanAttack(CurrentTimestamp)) // If the goblin can attack  {  for (int i = 0; i < Wizards.Count; i++) // Loop through all wizards  {  Wizard wizard = Wizards.Get(i); // Get the wizard  if (isLeastHP(wizard) && goblin.DistanceTo(wizard) <= goblin.AttackRange) // If the wizard has the least HP and the distance between the goblin and the wizard is less than or equal to the goblin's attack range  {  LogMessage($"Goblin {goblin} is attacking Wizard {wizard} with attack power {goblin.AttackPower}."); // Log the message  goblin.Attack(wizard, goblin.AttackPower, CurrentTimestamp); // Goblin attacks the wizard  //additional goblin powers because they are too weak  if (goblin.HP < 0.25 \* goblin.MaxHP) // If the goblin's HP is less than 0.25 of the goblin's max HP  {  goblin.HP += (wizard.MaxHP - wizard.HP) / 2; // Add half of the difference between the wizard's max HP and the wizard's HP to the goblin's HP  }  }  }  }  }  protected bool isLeastHP(Wizard wizard) // Is least HP method  {  foreach (Wizard w in Wizards) // Loop through all wizards  {  if (w.HP < wizard.HP) // If the HP of the wizard is less than the HP of the wizard  {  return false; // Return false if the HP of the wizard is not the least  }  }  return true; // Return true if the HP of the wizard is the least  } |
| Wizard casting AOE spell on goblins in range (edit: correction in future there will be targeting logic. Sri got ahead of himself if you read the previous version of this document)  In update method:  foreach (Wizard wizard in Wizards)  {  updateWizard(wizard);  }  protected void updateWizard(Wizard wizard) // Update wizard method  {  if (wizard.CanAttack(CurrentTimestamp)) // If wizard can attack  {  for (int i = 0; i < Goblins.Count; i++) // Loop through all goblins  {  Goblin goblin = Goblins.Get(i); // Get the goblin  if (wizard.DistanceTo(goblin) <= wizard.SpellRange) // If the distance between the wizard and the goblin is less than or equal to the wizard's spell range  {  // LogMessage($"Wizard {wizard} is attacking Goblin {goblin} with spell level {wizard.SpellLevel}."); // Log the message  wizard.Attack(goblin, (int)Math.Ceiling(wizard.SpellLevel / wizard.DistanceTo(goblin)), CurrentTimestamp); // Wizard attacks the goblin |
| Knockback/pushback logic and testing  In updateWizard method after attacking:  goblin.PushAwayFrom(wizard, 1.50f); // Push the goblin away from the wizard  goblin.ClampPosition(EntityXRange, EntityYRange); // Clamp the goblin's position  In updateGoblin method after attacking:  wizard.PushAwayFrom(goblin, 1.50f); // Push the wizard away from the goblin wizard.ClampPosition(EntityXRange, EntityYRange); // Clamp the wizard's position |
| Code for moving goblins and wizards (testing will be in the video)  In updateWizard method after if the wizard hasn’t attacked in this frame. 2 possiblities so 2 else statements:  else // If the distance between the wizard and the goblin is greater than the wizard's spell range  {  wizard.MoveTowards(getClosestGoblin(wizard, Goblins), 0.50f); // Move the wizard towards the closest goblin  wizard.ClampPosition(EntityXRange, EntityYRange); // Clamp the wizard's position  }  }  }  else  {  wizard.MoveTowards(getClosestGoblin(wizard, Goblins), 0.50f); // Move the wizard towards the closest goblin  wizard.ClampPosition(EntityXRange, EntityYRange); // Clamp the wizard's position  }  }  protected Goblin getClosestGoblin(Wizard wizard, ArrayList<Goblin> goblins) // Get the closest goblin method  {  Goblin closestGoblin = goblins.Get(0); // Get the first goblin  foreach (Goblin goblin in goblins) // Loop through all goblins  {  if (wizard.DistanceTo(goblin) < wizard.DistanceTo(closestGoblin)) // If the distance between the wizard and the goblin is less than the distance between the wizard and the closest goblin  {  closestGoblin = goblin; // Set the closest goblin to the goblin  }  }  return closestGoblin; // Return the closest goblin  }  In updateGoblin method:  protected void updateGoblin(Goblin goblin)  {  moveGoblinRandomly(goblin); // Move the goblin randomly  }  protected void moveGoblinRandomly(Goblin goblin) //Move the goblin randomly method  {  float dx = (RNG.NextSingle() \* 2 - 1) \* 0.60f; // Randomly generate a float between -1 and 1 and multiply it by the speed of the goblin  float dy = (RNG.NextSingle() \* 2 - 1) \* 0.60f; // Randomly generate a float between -1 and 1 and multiply it by the speed of the goblin  goblin.Move(dx, dy); // Move the goblin  goblin.ClampPosition(EntityXRange, EntityYRange); // Clamp the goblin's position  } |
| Code that generates new Goblins/Wizards and inserts them in sorted order without resorting the lists  AddGoblinAfter15Ticks();  AddWizardAfter50Ticks();  protected void AddGoblinAfter15Ticks() // Add goblin after 15 ticks method  {  if (CurrentTimestamp % 15 == 0 && !isBossGoblin) // If the current timestamp is divisible by 15 and we don't have a boss goblin  {  Goblins.AddBack(new Goblin()); // Add a new goblin  Goblin goblin = Goblins.Get(Goblins.Count - 1); // Get the goblin  AllEntities.InsertAt(getSortedIndex(goblin), goblin); // Insert the goblin into the allEntities arrayList  }  }  protected void AddWizardAfter50Ticks() // Add wizard after 50 ticks method  {  if (CurrentTimestamp % 50 == 0) // If the current timestamp is divisible by 50  {  Wizards.AddBack(new Wizard()); // Add a new wizard  Wizard wizard = Wizards.Get(Wizards.Count - 1); // Get the wizard  AllEntities.InsertAt(getSortedIndex(wizard), wizard); // Insert the wizard into the allEntities arrayList  }  }  protected int getSortedIndex(CombatEntity entity) // Get the sorted index method  {  // Binary search to find the index where the entity should be inserted.  int left = 0; // Left index  int right = AllEntities.Count - 1; // Right index  while (left <= right) // While the left index is less than or equal to the right index  {  int mid = (left + right) / 2; // Calculate the middle index  if (entity.MaxHP >= AllEntities.Get(mid).MaxHP) // If maxHP of the entity is greater than or equal to the maxHP of the entity at the middle index  {  right = mid - 1; // Set the right index to the middle index - 1  }  else  {  left = mid + 1; // Set the left index to the middle index + 1  }  }  return left; // Return the left index  } |
| Code to finish the game.  checkEntities(Goblins, Wizards);  checkGameOver();  protected void checkEntities(ArrayList<Goblin> Goblins, ArrayList<Wizard> Wizards) // Check entities method  {  for (int i = 0; i < Goblins.Count; i++) // Loop through all goblins  {  if (Goblins.Get(i).HP <= 0) // If the goblin's HP is less than or equal to 0  {  AllEntities.RemoveAt(AllEntities.FindIndex(Goblins.RemoveAt(i))); // Remove the goblin from the allEntities arrayList and the goblins arrayList  if (Goblins.Count == 0 && !isBossGoblin) // If there are no goblins left and we don't have a boss goblin  {  LogMessage("BOSS TIME"); // Log the message  Goblins.AddBack(new Goblin(isBossGoblin: true)); // Add a new boss goblin  Goblin goblin = Goblins.Get(Goblins.Count - 1); // Get the boss goblin  AllEntities.InsertAt(getSortedIndex(goblin), goblin); // Insert the boss goblin into the allEntities arrayList  isBossGoblin = true; // Set the boss goblin flag to true  }  }  }  for (int i = 0; i < Wizards.Count; i++) // Loop through all wizards  {  if (Wizards.Get(i).HP <= 0) // If the wizard's HP is less than or equal to 0  {  AllEntities.RemoveAt(AllEntities.FindIndex(Wizards.RemoveAt(i))); // Remove the wizard from the allEntities arrayList and the wizards arrayList  }  }  }  protected void checkGameOver() // Check game over method  {  if (Wizards.Count == 0 || Goblins.Count == 0) // If there are no wizards left or no goblins left  {  Stop(); // Stop the game  }  } |

Part 2.5. Theory questions. I’m not expecting huge novels here. 200 words per is probably about right (so anywhere from 100-300 is probably good).

|  |
| --- |
| Reflections on the inefficiency of removing from ISriList  When we are trying to remove the elements from our Array List implementation of ISriList, because of the nature of this array List , the removal method involves shifting of all the elements after the removed element and thus is an efficiency as this is an O(n) operation as when the list size will grow the time to loop through all the elements to shift would increase thus making the overall program slow and inefficient.  To make it more efficient we can essentially use a method that doesn’t involve shifting of all elements rather just removing it directly. One way to do can be by simply having a LinkedList type interface that supports making non-array-based collections. Instead of a buffer array, count, capacity and grow, we can have pointers, head and tail and nodes that will make the removal of an element much faster, especially when we use a switch to last and remove tail approach.  When using it in array List example we must loop over each element after the removed one and then switch to the one before, making it inefficient. If had large number of entities in our main game (vampire survivors), then removing would be extremely slow the game would run with immense lag. |

|  |
| --- |
| Reflections on other inefficient methods in ISriList  There definitely are some inefficient methods in the interface other than the removeAt.  Firstly, InsertAt follows a very similar pattern of switching positions after the added element to make space for a new one. This Is also O(n) and bad. If we had a LinkedList it wouldn’t work in this case, as we will need the nodes, head and tail pointers as well. Also, if we do insertAt(index i) for a linkedList even then we will have to loop through the collection till the location to add it is found and then simply it is added there.  Also, the Get (index i ) method in an ISriList arrayList implementation would be simple O(1), but a problem comes when we try to implement linkedList using the same interface, as to get a specific item in a linkedlist we will have to loop and make our head change until we reach the element.  Lastly the Swap() method can have potential problems when trying to make it work in non array based lists as swapping involves complex changing of head pointers in a linkedList making it O(n) and highly complicated to implement. |