Національний технічний університет України «КПІ»

Факультет інформатики та обчислювальної техніки

Кафедра Інформаційних систем та технологій

Лабораторна робота №1

з дисципліни « Сучасні технології розробки WEB-застосувань на платформі Microsoft.NET»

на тему: « Узагальнені типи (Generic) з підтримкою подій. Колекції»

Виконав:

студент гр. ІП-11

Веремчук Ігор

Викладач:

Бардін В.

2023 рік

**Мета**: навчитися проектувати та реалізовувати узагальнені типи, а також типи з підтримкою подій.

**Завдання:**

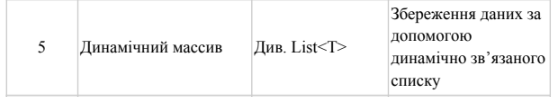
1. Розробити клас власної узагальненої колекції, використовуючи стандартні інтерфейси колекцій із бібліотек System.Collections та System.Collections.Generic. Стандартні колекції при розробці власної не застосовувати. Для колекції передбачити методи внесення даних будь-якого типу, видалення, пошуку та ін. (відповідно до типу колекції).

2. Додати до класу власної узагальненої колекції підтримку подій та обробку виключних ситуацій.

3. Опис класу колекції та всіх необхідних для роботи з колекцією типів зберегти у динамічній бібліотеці.

4. Створити консольний додаток, в якому продемонструвати використання розробленої власної колекції, підписку на події колекції.

**Варіант 5:**



**Посилання на код GitHub:**

https://github.com/vrmchk/.NET  
Код основних класів:

**NodeEnumerator:**

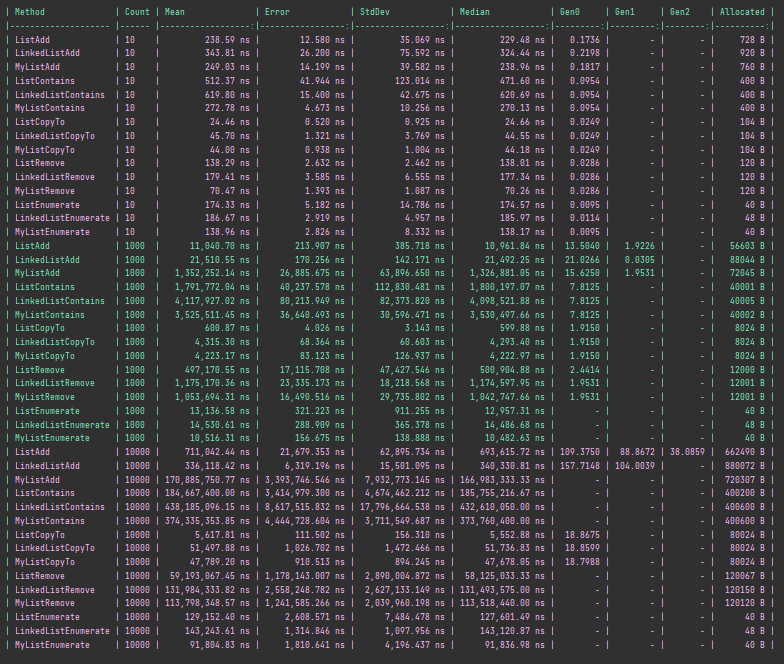
public class NodeEnumerator<T> : IEnumerator<T>  
{  
 private int \_position = -1;  
 private readonly Node<T>? \_rootNode;  
 private Node<T>? \_currentNode;  
  
 public NodeEnumerator(Node<T>? rootNode)  
 {  
 \_rootNode = rootNode;  
 \_currentNode = \_rootNode;  
 }  
  
 public T Current => \_currentNode != null ? \_currentNode.Value : default;  
  
 object IEnumerator.Current => Current;  
  
 public bool MoveNext()  
 {  
 if (\_currentNode == null)  
 return false;  
  
 if (\_position == -1)  
 {  
 \_position++;  
 return true;  
 }  
  
 if (\_currentNode.Next == null)  
 return false;  
  
 \_currentNode = \_currentNode.Next;  
 \_position++;  
 return true;  
 }  
  
 public void Reset()  
 {  
 \_currentNode = \_rootNode;  
 \_position = -1;  
 }  
  
 public void Dispose() { }  
}

**MyList:**

public class MyList<T> : IList<T>, ICloneable  
{  
 private Node<T>? \_head;  
  
 public int Count { get; private set; }  
 public bool IsReadOnly => false;  
 private bool ShouldNotify => CollectionChanged != null;  
  
 public T this[int index]  
 {  
 get => GetNode(index).Value;  
 set  
 {  
 if (!ShouldNotify)  
 {  
 GetNode(index).Value = value;  
 return;  
 }  
  
 var node = GetNode(index);  
  
 var oldItem = node.Value;  
 var oldCollection = (MyList<T>)Clone();  
  
 node.Value = value;  
  
 OnCollectionChanged(CollectionChangeType.**Update**, oldItem, value, oldCollection);  
 }  
 }  
  
 public event Action<CollectionChangedEventArgs<T>>? CollectionChanged;  
  
 public void Add(T item)  
 {  
 if (!ShouldNotify)  
 {  
 PerformAdd();  
 return;  
 }  
  
 var oldCollection = (MyList<T>)Clone();  
 PerformAdd();  
 OnCollectionChanged(CollectionChangeType.**Add**, newItem: item, oldCollection: oldCollection);  
 return;  
  
 void PerformAdd()  
 {  
 if (\_head == null)  
 {  
 \_head = new Node<T>(item);  
 Count++;  
 return;  
 }  
  
 var current = \_head;  
 while (current is { Next: not null })  
 {  
 current = current.Next;  
 }  
  
 current.Next = new Node<T>(item);  
 Count++;  
 }  
 }  
  
 public void Clear()  
 {  
 if (!ShouldNotify)  
 {  
 PerformClear();  
 return;  
 }  
  
 var oldCollection = (MyList<T>)Clone();  
 PerformClear();  
  
 OnCollectionChanged(CollectionChangeType.**Clear**, oldCollection: oldCollection);  
 return;  
  
 void PerformClear()  
 {  
 \_head = null;  
 Count = 0;  
 }  
 }  
  
 public bool Contains(T item)  
 {  
 return IndexOf(item) >= 0;  
 }  
  
 public void CopyTo(T[] array, int arrayIndex)  
 {  
 if (array == null)  
 throw new ArgumentNullException(nameof(array));  
  
 if (arrayIndex < 0)  
 throw new ArgumentOutOfRangeException(nameof(arrayIndex));  
  
 if (array.Length - arrayIndex < Count)  
 throw new ArgumentException("Destination array was not long enough.");  
  
 var current = \_head;  
 while (current != null)  
 {  
 array[arrayIndex++] = current.Value;  
 current = current.Next;  
 }  
 }  
  
 public bool Remove(T item)  
 {  
 var current = \_head;  
 Node<T>? previous = null;  
 while (current != null)  
 {  
 if (current.Value != null && current.Value.Equals(item))  
 {  
 if (!ShouldNotify)  
 {  
 Remove(previous, current);  
 return true;  
 }  
  
 var oldCollection = (MyList<T>)Clone();  
 Remove(previous, current);  
  
 OnCollectionChanged(CollectionChangeType.**Remove**, item, oldCollection: oldCollection);  
 return true;  
 }  
  
 previous = current;  
 current = current.Next;  
 }  
  
 return false;  
 }  
  
 public int IndexOf(T item)  
 {  
 var current = \_head;  
 var index = 0;  
 while (current != null)  
 {  
 if (current.Value != null && current.Value.Equals(item))  
 return index;  
  
 current = current.Next;  
 index++;  
 }  
  
 return -1;  
 }  
  
 public void Insert(int index, T item)  
 {  
 if (!ShouldNotify)  
 {  
 PerformInsert();  
 return;  
 }  
  
 var oldCollection = (MyList<T>)Clone();  
 PerformInsert();  
 OnCollectionChanged(CollectionChangeType.**Add**, newItem: item, oldCollection: oldCollection);  
 return;  
  
 void PerformInsert()  
 {  
 if (index == 0)  
 {  
 \_head = new Node<T>(item) { Next = \_head };  
 Count++;  
 return;  
 }  
  
 if (index == Count)  
 {  
 Add(item);  
 return;  
 }  
  
 var current = GetNode(index - 1);  
 var newNode = new Node<T>(item) { Next = current.Next };  
 current.Next = newNode;  
 Count++;  
 }  
 }  
  
 public void RemoveAt(int index)  
 {  
 var previous = index != 0 ? GetNode(index - 1) : null;  
 var toRemove = previous?.Next ?? \_head;  
 if (toRemove == null)  
 return;  
  
 if (!ShouldNotify)  
 {  
 Remove(previous, toRemove);  
 return;  
 }  
  
 var oldCollection = (MyList<T>)Clone();  
 var oldItem = toRemove.Value;  
 Remove(previous, toRemove);  
 OnCollectionChanged(CollectionChangeType.**Remove**, oldItem, oldCollection: oldCollection);  
 }  
  
 public IEnumerator<T> GetEnumerator()  
 {  
 return new NodeEnumerator<T>(\_head);  
 }  
  
 IEnumerator IEnumerable.GetEnumerator()  
 {  
 return GetEnumerator();  
 }  
  
 public object Clone()  
 {  
 return new MyList<T> { \_head = (Node<T>?)\_head?.Clone() };  
 }  
  
 private Node<T> GetNode(int index)  
 {  
 if (index < 0 || index >= Count)  
 throw new IndexOutOfRangeException();  
  
 var current = \_head;  
 for (int i = 0; i < index; i++)  
 {  
 if (current == null)  
 throw new IndexOutOfRangeException();  
  
 current = current.Next;  
 }  
  
 if (current == null)  
 throw new IndexOutOfRangeException();  
  
 return current;  
 }  
  
 private void Remove(Node<T>? previous, Node<T> current)  
 {  
 if (previous == null)  
 \_head = current.Next;  
 else  
 previous.Next = current.Next;  
 Count--;  
 }  
  
 private void OnCollectionChanged(CollectionChangeType changeType,  
 T? oldItem = default,  
 T? newItem = default,  
 ICollection<T>? oldCollection = null)  
 {  
 CollectionChanged?.Invoke(new CollectionChangedEventArgs<T>  
 {  
 ChangeType = changeType,  
 OldItem = oldItem,  
 NewItem = newItem,  
 OldCollection = oldCollection ?? new List<T>(),  
 NewCollection = this  
 });  
 }  
}

Як додаткове завдання написано бенчмарки за допомогою бібліотеки BenchmarkDotNet, де порівнювалась моя реалізація списку реалізації списку від Microsoft.

Результати:



Generic type – Lazy<Int32>

**Висновки:** під час виконання лабораторної роботи я навчився проектувати та реалізовувати узагальнені колекції, з підтримкою подій та виключних ситуацій, було реалізовано динамічний масив на базі динамічно зв’язного списку.