Deakin University

Data Structures and Algorithms

OnTrack Submission

Vector: A simple list-like collection class

Submitted By:
Peter Stacey
pstacey
2020/07/03 11:47

Tutor:
Richard Dazeley

Outcome	Weight
Complexity	♦◊◊◊◊
Implement Solutions	♦♦♦◊◊
Document solutions	♦♦◊◊◊

This task involves implementing basic algorithms within the method bodies of a Vector class, built on an array as an underlying data structure. This includes addressing the specified requirements to produce a solution that meets the needs of a generic vector class. As a result, this aligns well with ULO2. Similarly, although not directly part of the task, documenting the code is good practice and has been included in my submission. This aligns with ULO3, although the main comments include XML comments as the code is self explanatory. The task does not at all involve investigating memory usage or computational complexity of using an array as the underlying data structure. While algorithms are implemented to adjust the capacity of the array and the copy elements from one array to another, this is part of the provided code and not directly investigated in the task.

July 3, 2020

Produced by Doubtfire

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

namespace Vector
{
    public class Vector<T>
    {
        // This constant determines the default number of elements in a newly created vector.
        // It is also used to extended the capacity of the existing vector
        private const int DEFAULT_CAPACITY = 10;

        // This array represents the internal data structure wrapped by the vector class.
        // In fact, all the elements are to be stored in this private  array.
        // You will just write extra functionality (methods) to make the work with the array more convenient for the user.
        private T[] data;

        // This property represents the number of elements in the vector
        public int Count { get; private set; } = 0;

        // This property represents the maximum number of elements (capacity) in the vector
        public int Capacity { get; private set; } = 0;

        // This is an overloaded constructor
        public Vector(int capacity)
        {
            data = new T[capacity];
            Capacity = capacity;
        }

        // This is the implementation of the default constructor
        public Vector() : this(DEFAULT_CAPACITY) { }

        // An Indexer is a special type of property that allows a class or structure to be accessed the same way as array for its internal collection.
        // For example, introducing the following indexer you may address an element of the vector as vector[i] or vector[0] or ...
        public T this[int index]
        {
            get
            {
                if (index >= Count || index < 0) throw new IndexOutOfRangeException();
                return data[index];
            }
            set
            {
                if (index >= Count || index < 0) throw new
                 ↪  IndexOutOfRangeException();
                data[index] = value;
            }
        }

        // This private method allows extension of the existing capacity of the
         ↪  vector by another 'extraCapacity' elements.
        // The new capacity is equal to the existing one plus 'extraCapacity'.
        // It copies the elements of 'data' (the existing array) to 'newData' (the
         ↪  new array), and then makes data pointing to 'newData'.
        private void ExtendData(int extraCapacity)
        {
            T[] newData = new T[data.Length + extraCapacity];
            for (int i = 0; i < Count; i++) newData[i] = data[i];
            data = newData;
        }

        // This method adds a new element to the existing array.
        // If the internal array is out of capacity, its capacity is first extended
         ↪  to fit the new element.
        public void Add(T element)
        {
            if (Count == data.Length) ExtendData(DEFAULT_CAPACITY);
            data[Count++] = element;
        }

        // This method searches for the specified object and returns the zerobased
         ↪  index of the first occurrence within the entire data structure.
        // This method performs a linear search; therefore, this method is an O(n)
         ↪  runtime complexity operation.
        // If occurrence is not found, then the method returns 1.
        // Note that Equals is the proper method to compare two objects for
         ↪  equality, you must not use operator '=' for this purpose.
        public int IndexOf(T element)
        {
            for (var i = 0; i < Count; i++)
            {
                if (data[i].Equals(element)) return i;
            }
            return -1;
        }

        // TODO:***************************************************************
         ↪  *************************
        // TODO: Your task is to implement all the remaining methods.
        // Read the instruction carefully, study the code examples from above as
         ↪  they should help you to
        // write the rest of the code.

        /// <summary>
        /// Inserts a new element into the vector at the provided index if the
         ↪  index is valid.
        /// </summary>
        /// <param name="index">Position in the vector to add the element</param>
        /// <param name="element">Element to add</param>
        /// <exception cref="System.IndexOutOfRangeException">Thrown when the index
        /// is outside the valid range [0, count of elements]</exception>
        public void Insert(int index, T element)
        {
            if (index < 0 || index > Count) throw new IndexOutOfRangeException();
            if (Count == Capacity) ExtendData(DEFAULT_CAPACITY);
            if (index == Count)
            {
                data[Count++] = element;
                return;
            }
            int current = Count;
            while(current > index)
            {
                data[current] = data[current - 1];
                current--;
            }
            data[index] = element;
            Count++;
        }

        /// <summary>
        /// Clears the vector of all current elements
        /// </summary>
        public void Clear()
        {
            data = new T[Capacity];
            Count = 0;
        }

        /// <summary>
        /// Identifies whether the vector contains a specific element.
        /// </summary>
        /// <param name="element">The element to search for</param>
        /// <returns>True if the element is present and falso otherwise</returns>
        public bool Contains(T element)
        {
            return IndexOf(element) >= 0;
        }

        /// <summary>
        /// Removes the first occurence of a given element, if it exists.
        /// </summary>
        /// <param name="element">The element to remove the first occurence of,
        /// if present</param>
        /// <returns>True if the element was found and removed,
        /// and false otherwise</returns>
        public bool Remove(T element)
        {
            int index = IndexOf(element);
            if (index is -1) return false;
            RemoveAt(index);
            return true;
        }

        /// <summary>
        /// Removes an element from a specific index position in the vector
        /// </summary>
        /// <param name="index">The index position to delete the elelent of</param>
        /// <exception cref="System.IndexOutOfRangeException">Thrown if the provided
        /// index is not in the valid range, [0, Count - 1]</exception>
        public void RemoveAt(int index)
        {
            if (index < 0 || index >= Count) throw new IndexOutOfRangeException();
            if (Count is 1)
            {
                data = new T[Capacity];
                Count = 0;
                return;
            }
            for (int i = index + 1; i < Count; i++) data[i-1] = data[i];
            Count--;
        }

        /// <summary>
        /// Returns a string representation of the Vector in the form
        /// [ #, #, #, #, #] where each # is one element of the vector
        /// commencing from index 0
        /// </summary>
        /// <returns>A string representation of the Vector</returns>
        public override string ToString()
        {
            if (Count is 0) return "[]";
            return "[" + string.Join(",", data).Substring(0, Count * 2 - 1) + "]";
        }
    }
}
```