Deakin University

Data Structures and Algorithms

OnTrack Submission

Iteration and Search

Submitted By:
Peter Stacey
pstacey
2020/08/11 13:10

Tutor:
Maksym Slavnenko

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

The task involves implementing a recursive Binary Search, with it's improved performance compared to linear search approaches implemented in previous tasks. This is related to ULO1 and ULO2. The task also involves implementing the Iterator pattern to facilitate iteration over a vector. As a standard Design Pattern, this provides practice and knowledge in implementing the pattern and enables foreach to be used on the Vector. This pattern aligns with ULO2, in particular the implementation of programs that address specific requirements.

August 11, 2020

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

namespace Task_4_1
{
    public class Vector<T> : IEnumerable<T> where T : IComparable<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 { return data.Length; }
        }

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

```
46              set
47              {
48                  if (index >= Count || index < 0) throw new
                    ↪  IndexOutOfRangeException();
49                  data[index] = value;
50              }
51          }
52
53          // This private method allows extension of the existing capacity of the
            ↪  vector by another 'extraCapacity' elements.
54          // The new capacity is equal to the existing one plus 'extraCapacity'.
55          // It copies the elements of 'data' (the existing array) to 'newData' (the
            ↪  new array), and then makes data pointing to 'newData'.
56          private void ExtendData(int extraCapacity)
57          {
58              T[] newData = new T[Capacity + extraCapacity];
59              for (int i = 0; i < Count; i++) newData[i] = data[i];
60              data = newData;
61          }
62
63          // This method adds a new element to the existing array.
64          // If the internal array is out of capacity, its capacity is first extended
            ↪  to fit the new element.
65          public void Add(T element)
66          {
67              if (Count == Capacity) ExtendData(DEFAULT_CAPACITY);
68              data[Count++] = element;
69          }
70
71          // This method searches for the specified object and returns the zerobased
            ↪  index of the first occurrence within the entire data structure.
72          // This method performs a linear search; therefore, this method is an O(n)
            ↪  runtime complexity operation.
73          // If occurrence is not found, then the method returns 1.
74          // Note that Equals is the proper method to compare two objects for
            ↪  equality, you must not use operator '=' for this purpose.
75          public int IndexOf(T element)
76          {
77              for (var i = 0; i < Count; i++)
78              {
79                  if (data[i].Equals(element)) return i;
80              }
81              return -1;
82          }
83
84          // Returns a string representation of the elements of the Vector
85          public override string ToString() => "[" + string.Join(", ",
            ↪  data[0..Count]) + "]";
86
87          public ISorter Sorter { set; get; } = new DefaultSorter();
88
89          internal class DefaultSorter : ISorter
90          {
```

```
91              public void Sort<K>(K[] sequence, IComparer<K> comparer) where K :
                ↪ IComparable<K>
92              {
93                  if (comparer == null) comparer = Comparer<K>.Default;
94                  Array.Sort(sequence, comparer);
95              }
96          }
97
98          public void Sort()
99          {
100             if (Sorter == null) Sorter = new DefaultSorter();
101             Array.Resize(ref data, Count);
102             Sorter.Sort(data, null);
103         }
104
105         public void Sort(IComparer<T> comparer)
106         {
107             if (Sorter == null) Sorter = new DefaultSorter();
108             Array.Resize(ref data, Count);
109             if (comparer == null) Sorter.Sort(data, null);
110             else Sorter.Sort(data, comparer);
111         }
112
113         // TODO: Your task is to implement all the remaining methods.
114         // Read the instruction carefully, study the code examples from above as
                ↪ they should help you to write the rest of the code.
115
116         /// <summary>
117         /// Conducts a BinarySearch on the sorted data using a default comparerand
                ↪ returns
118         /// the index of the element or -1 if the element does not exist in the data
119         /// </summary>
120         /// <param name="element">The element to search for</param>
121         /// <returns>The 0 based index of the element, or -1 if it doesn't
                ↪ exist</returns>
122         public int BinarySearch(T element)
123         {
124             IComparer<T> comparer = Comparer<T>.Default;
125             return BinarySearch(element, comparer);
126         }
127
128         /// <summary>
129         /// Conducts a BinarySearch on the sorted data and returns the index of the
                ↪ element or -1 if the
130         /// element does not exist in the data
131         /// </summary>
132         /// <param name="element">The element to search for</param>
133         /// <param name="comparer">The IComparer to use for comparison</param>
134         /// <returns>The 0 based index of the element, or -1 if it doesn't
                ↪ exist</returns>
135         public int BinarySearch(T element, IComparer<T> comparer)
136         {
137             if (Count is 0) return -1;
```

```
138             if (comparer is null) comparer = Comparer<T>.Default;
139             return BinarySearch(element, comparer, 0, data.Length - 1);
140         }
141 
142         /// <summary>
143         /// Recursively searches the vector for the required element by dividing
            ↪ the data into
144         /// smaller and smaller sections until the element is found. If the element
            ↪ is not
145         /// present, -1 is returned
146         /// </summary>
147         /// <param name="element">The element to search for</param>
148         /// <param name="comparer">The IComparer to use for comparison</param>
149         /// <param name="lower">The lower index of the section to search</param>
150         /// <param name="upper">The upper index of the section to search</param>
151         /// <returns>The 0 based index of the element, or -1 if it does not
            ↪ exist</returns>
152         private int BinarySearch(T element, IComparer<T> comparer, int lower, int
            ↪ upper)
153         {
154             if (lower > upper) return -1;
155             int mid = (int)(upper + lower) / 2;
156             if (comparer.Compare(element, data[mid]) < 0) return
                ↪ BinarySearch(element, comparer, lower, mid - 1);
157             else if (comparer.Compare(element, data[mid]) == 0) return mid;
158             else return BinarySearch(element, comparer, mid + 1, upper);
159         }
160 
161         /// <summary>
162         /// Returns a new IEnumerable<T> for the Vector
163         /// </summary>
164         /// <returns>Iterator for the Vector</returns>
165         public IEnumerator<T> GetEnumerator() => new Iterator(this);
166 
167         /// <summary>
168         /// Returns the IEnumerable implemented within IEnumerable<T>
169         /// </summary>
170         /// <returns>Enumerable implemented within IEnumerable</returns>
171         IEnumerator IEnumerable.GetEnumerator() => GetEnumerator();
172 
173         /// <summary>
174         /// Implementation of the IEnumerator<T> interface to
175         /// facilitate iteration over a vector of elements of
176         /// type T
177         /// </summary>
178         private class Iterator : IEnumerator<T>
179         {
180             private Vector<T> _v;
181             private int _currentIndex = -1;
182             public Iterator(Vector<T> v) => _v = v;
183 
184             public T Current
185             {
```

```
186                 get
187                 {
188                     try
189                     {
190                         return _v.data[_currentIndex];
191                     }
192                     catch (IndexOutOfRangeException)
193                     {
194                         return default(T);
195                     }
196                 }
197             }
198 
199             object IEnumerator.Current
200             {
201                 get
202                 {
203                     try
204                     {
205                         return _v.data[_currentIndex];
206                     }
207                     catch (IndexOutOfRangeException)
208                     {
209                         return default(T);
210                     }
211                 }
212             }
213 
214             /// <summary>
215             /// Advances the iterator cursor to the next position and
216             /// returns whether the iterator has fully
217             /// iterated the vector.
218             /// </summary>
219             /// <returns>Boolean whether the Vector is fully iterated</returns>
220             public bool MoveNext()
221             {
222                 _currentIndex++;
223                 return _currentIndex < _v.Count;
224             }
225 
226             /// <summary>
227             /// Returns the iterator cursor to the start to allow another iteration
228             /// </summary>
229             public void Reset() => _currentIndex = -1;
230 
231             /// <summary>
232             /// Currently unused. Previously required in the IEnumerator
233             /// implementation, but no longer the case.
234             /// </summary>
235             public void Dispose() { }
236         }
237     }
238 }
```