

LinksPlatform's Platform.Collections.Methods Class Library

1.1 ./csharp/Platform.Collections.Methods/GenericCollectionMethodsBase.cs

```

1  using System.Collections.Generic;
2  using System.Runtime.CompilerServices;
3  using Platform.Numbers;
4
5  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6
7  namespace Platform.Collections.Methods
8  {
9      /// <summary>
10     /// <para>Represents a base implementation of methods for a collection of elements of type
11     ↪ TElement.</para>
12     /// <para>Представляет базовую реализацию методов коллекции элементов типа TElement.</para>
13     /// </summary>
14     /// <typeparam name="TElement"><para>Source type of conversion.</para><para>Исходный тип
15     ↪ конверсии.</para></typeparam>
16     public abstract class GenericCollectionMethodsBase<TElement>
17     {
18         /// <summary>
19         /// <para>Returns a null constant of type <see cref="TElement" />.</para>
20         /// <para>Возвращает нулевую константу типа <see cref="TElement" />.</para>
21         /// </summary>
22         /// <returns><para>A null constant of type <see cref="TElement" />.</para><para>Нулевую
23         ↪ константу типа <see cref="TElement" />.</para></returns>
24         [MethodImpl(MethodImplOptions.AggressiveInlining)]
25         protected virtual TElement GetZero() => default;
26
27         /// <summary>
28         /// <para>Determines whether the specified value is equal to zero type <see
29         ↪ cref="TElement" />.</para>
30         /// <para>Определяет равно ли нулю указанное значение типа <see cref="TElement"
31         ↪ />.</para>
32         /// </summary>
33         /// <returns><para></para>Is the specified value equal to zero type <see cref="TElement"
34         ↪ /><para>Равно ли нулю указанное значение типа <see cref="TElement"
35         ↪ /></para></returns>
36         [MethodImpl(MethodImplOptions.AggressiveInlining)]
37         protected virtual bool EqualToZero(TElement value) => EqualityComparer.Equals(value,
38         ↪ Zero);
39
40         /// <summary>
41         /// <para>Presents the Range in readable format.</para>
42         /// <para>Представляет диапазон в удобном для чтения формате.</para>
43         /// </summary>
44         /// <returns><para>String representation of the Range.</para><para>Строковое
45         ↪ представление диапазона.</para></returns>
46         [MethodImpl(MethodImplOptions.AggressiveInlining)]
47         protected virtual bool AreEqual(TElement first, TElement second) =>
48         ↪ EqualityComparer.Equals(first, second);
49
50         /// <summary>
51         /// <para>Presents the Range in readable format.</para>
52         /// <para>Представляет диапазон в удобном для чтения формате.</para>
53         /// </summary>
54         /// <returns><para>String representation of the Range.</para><para>Строковое
55         ↪ представление диапазона.</para></returns>
56         [MethodImpl(MethodImplOptions.AggressiveInlining)]
57         protected virtual bool GreaterThanZero(TElement value) => Comparer.Compare(value, Zero)
58         ↪ > 0;
59
60         /// <summary>
61         /// <para>Presents the Range in readable format.</para>
62         /// <para>Представляет диапазон в удобном для чтения формате.</para>
63         /// </summary>
64         /// <returns><para>String representation of the Range.</para><para>Строковое
65         ↪ представление диапазона.</para></returns>
66         [MethodImpl(MethodImplOptions.AggressiveInlining)]
67         protected virtual bool GreaterThan(TElement first, TElement second) =>
68         ↪ Comparer.Compare(first, second) > 0;
69
70         /// <summary>
71         /// <para>Presents the Range in readable format.</para>
72         /// <para>Представляет диапазон в удобном для чтения формате.</para>
73         /// </summary>
74         /// <returns><para>String representation of the Range.</para><para>Строковое
75         ↪ представление диапазона.</para></returns>

```

```

61 [MethodImpl(MethodImplOptions.AggressiveInlining)]
62 protected virtual bool GreaterOrEqualThanZero(TElement value) => Comparer.Compare(value,
    ↪ Zero) >= 0;
63
64 /// <summary>
65 /// <para>Presents the Range in readable format.</para>
66 /// <para>Представляет диапазон в удобном для чтения формате.</para>
67 /// </summary>
68 /// <returns><para>String representation of the Range.</para><para>Строковое
    ↪ представление диапазона.</para></returns>
69 [MethodImpl(MethodImplOptions.AggressiveInlining)]
70 protected virtual bool GreaterOrEqualThan(TElement first, TElement second) =>
    ↪ Comparer.Compare(first, second) >= 0;
71
72 /// <summary>
73 /// <para>Presents the Range in readable format.</para>
74 /// <para>Представляет диапазон в удобном для чтения формате.</para>
75 /// </summary>
76 /// <returns><para>String representation of the Range.</para><para>Строковое
    ↪ представление диапазона.</para></returns>
77 [MethodImpl(MethodImplOptions.AggressiveInlining)]
78 protected virtual bool LessOrEqualThanZero(TElement value) => Comparer.Compare(value,
    ↪ Zero) <= 0;
79
80 /// <summary>
81 /// <para>Presents the Range in readable format.</para>
82 /// <para>Представляет диапазон в удобном для чтения формате.</para>
83 /// </summary>
84 /// <returns><para>String representation of the Range.</para><para>Строковое
    ↪ представление диапазона.</para></returns>
85 [MethodImpl(MethodImplOptions.AggressiveInlining)]
86 protected virtual bool LessOrEqualThan(TElement first, TElement second) =>
    ↪ Comparer.Compare(first, second) <= 0;
87
88 /// <summary>
89 /// <para>Presents the Range in readable format.</para>
90 /// <para>Представляет диапазон в удобном для чтения формате.</para>
91 /// </summary>
92 /// <returns><para>String representation of the Range.</para><para>Строковое
    ↪ представление диапазона.</para></returns>
93 [MethodImpl(MethodImplOptions.AggressiveInlining)]
94 protected virtual bool LessThanZero(TElement value) => Comparer.Compare(value, Zero) < 0;
95
96 /// <summary>
97 /// <para>Presents the Range in readable format.</para>
98 /// <para>Представляет диапазон в удобном для чтения формате.</para>
99 /// </summary>
100 /// <returns><para>String representation of the Range.</para><para>Строковое
    ↪ представление диапазона.</para></returns>
101 [MethodImpl(MethodImplOptions.AggressiveInlining)]
102 protected virtual bool LessThan(TElement first, TElement second) =>
    ↪ Comparer.Compare(first, second) < 0;
103
104 /// <summary>
105 /// <para>Presents the Range in readable format.</para>
106 /// <para>Представляет диапазон в удобном для чтения формате.</para>
107 /// </summary>
108 /// <returns><para>String representation of the Range.</para><para>Строковое
    ↪ представление диапазона.</para></returns>
109 [MethodImpl(MethodImplOptions.AggressiveInlining)]
110 protected virtual TElement Increment(TElement value) =>
    ↪ Arithmetic<TElement>.Increment(value);
111
112 /// <summary>
113 /// <para>Presents the Range in readable format.</para>
114 /// <para>Представляет диапазон в удобном для чтения формате.</para>
115 /// </summary>
116 /// <returns><para>String representation of the Range.</para><para>Строковое
    ↪ представление диапазона.</para></returns>
117 [MethodImpl(MethodImplOptions.AggressiveInlining)]
118 protected virtual TElement Decrement(TElement value) =>
    ↪ Arithmetic<TElement>.Decrement(value);
119
120 /// <summary>
121 /// <para>Presents the Range in readable format.</para>
122 /// <para>Представляет диапазон в удобном для чтения формате.</para>
123 /// </summary>

```

```

124    /// <returns><para>String representation of the Range.</para><para>Строковое
    ↪ представление диапазона.</para></returns>
125    [MethodImpl(MethodImplOptions.AggressiveInlining)]
126    protected virtual TElement Add(TElement first, TElement second) =>
    ↪ Arithmetic<TElement>.Add(first, second);

127
128    /// <summary>
129    /// <para>Presents the Range in readable format.</para>
130    /// <para>Представляет диапазон в удобном для чтения формате.</para>
131    /// </summary>
132    /// <returns><para>String representation of the Range.</para><para>Строковое
    ↪ представление диапазона.</para></returns>
133    [MethodImpl(MethodImplOptions.AggressiveInlining)]
134    protected virtual TElement Subtract(TElement first, TElement second) =>
    ↪ Arithmetic<TElement>.Subtract(first, second);

135
136    /// <summary>
137    /// <para>Returns minimum value of the range.</para>
138    /// <para>Возвращает минимальное значение диапазона.</para>
139    /// </summary>
140    protected readonly TElement Zero;

141
142    /// <summary>
143    /// <para>Returns minimum value of the range.</para>
144    /// <para>Возвращает минимальное значение диапазона.</para>
145    /// </summary>
146    protected readonly TElement One;

147
148    /// <summary>
149    /// <para>Returns minimum value of the range.</para>
150    /// <para>Возвращает минимальное значение диапазона.</para>
151    /// </summary>
152    protected readonly TElement Two;

153
154    /// <summary>
155    /// <para>Returns minimum value of the range.</para>
156    /// <para>Возвращает минимальное значение диапазона.</para>
157    /// </summary>
158    protected readonly EqualityComparer<TElement> EqualityComparer;

159
160    /// <summary>
161    /// <para>Returns minimum value of the range.</para>
162    /// <para>Возвращает минимальное значение диапазона.</para>
163    /// </summary>
164    protected readonly Comparer<TElement> Comparer;

165
166    /// <summary>
167    /// <para>Presents the Range in readable format.</para>
168    /// <para>Представляет диапазон в удобном для чтения формате.</para>
169    /// </summary>
170    /// <returns><para>String representation of the Range.</para><para>Строковое
    ↪ представление диапазона.</para></returns>
171    protected GenericCollectionMethodsBase()
172    {
173        EqualityComparer = EqualityComparer<TElement>.Default;
174        Comparer = Comparer<TElement>.Default;
175        Zero = GetZero(); //-V3068
176        One = Increment(Zero); //-V3068
177        Two = Increment(One); //-V3068
178    }
179 }
180 }

```

1.2 ./csharp/Platform.Collections.Methods/Lists/AbsoluteCircularDoublyLinkedListMethods.cs

```

1  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
2
3  namespace Platform.Collections.Methods.Lists
4  {
5      /// <summary>
6      /// <para>
7      /// Represents the absolute circular doubly linked list methods.
8      /// </para>
9      /// <para></para>
10     /// </summary>
11     /// <seealso cref="AbsoluteDoublyLinkedListMethodsBase{TElement}">
12     public abstract class AbsoluteCircularDoublyLinkedListMethods<TElement> :
    ↪ AbsoluteDoublyLinkedListMethodsBase<TElement>
13     {
14         /// <summary>

```

```

15     /// <para>
16     /// Attaches the before using the specified base element.
17     /// </para>
18     /// <para></para>
19     /// </summary>
20     /// <param name="baseElement">
21     /// <para>The base element.</para>
22     /// <para></para>
23     /// </param>
24     /// <param name="newElement">
25     /// <para>The new element.</para>
26     /// <para></para>
27     /// </param>
28 public void AttachBefore(TElement baseElement, TElement newElement)
29 {
30     var baseElementPrevious = GetPrevious(baseElement);
31     SetPrevious(newElement, baseElementPrevious);
32     SetNext(newElement, baseElement);
33     if (AreEqual(baseElement, GetFirst()))
34     {
35         SetFirst(newElement);
36     }
37     SetNext(baseElementPrevious, newElement);
38     SetPrevious(baseElement, newElement);
39     IncrementSize();
40 }
41
42     /// <summary>
43     /// <para>
44     /// Attaches the after using the specified base element.
45     /// </para>
46     /// <para></para>
47     /// </summary>
48     /// <param name="baseElement">
49     /// <para>The base element.</para>
50     /// <para></para>
51     /// </param>
52     /// <param name="newElement">
53     /// <para>The new element.</para>
54     /// <para></para>
55     /// </param>
56 public void AttachAfter(TElement baseElement, TElement newElement)
57 {
58     var baseElementNext = GetNext(baseElement);
59     SetPrevious(newElement, baseElement);
60     SetNext(newElement, baseElementNext);
61     if (AreEqual(baseElement, GetLast()))
62     {
63         SetLast(newElement);
64     }
65     SetPrevious(baseElementNext, newElement);
66     SetNext(baseElement, newElement);
67     IncrementSize();
68 }
69
70     /// <summary>
71     /// <para>
72     /// Attaches the as first using the specified element.
73     /// </para>
74     /// <para></para>
75     /// </summary>
76     /// <param name="element">
77     /// <para>The element.</para>
78     /// <para></para>
79     /// </param>
80 public void AttachAsFirst(TElement element)
81 {
82     var first = GetFirst();
83     if (EqualToZero(first))
84     {
85         SetFirst(element);
86         SetLast(element);
87         SetPrevious(element, element);
88         SetNext(element, element);
89         IncrementSize();
90     }
91     else
92     {

```

```

93         AttachBefore(first, element);
94     }
95 }
96
97 /// <summary>
98 /// <para>
99 /// Attaches the as last using the specified element.
100 /// </para>
101 /// <para></para>
102 /// </summary>
103 /// <param name="element">
104 /// <para>The element.</para>
105 /// <para></para>
106 /// </param>
107 public void AttachAsLast(TElement element)
108 {
109     var last = GetLast();
110     if (EqualToZero(last))
111     {
112         AttachAsFirst(element);
113     }
114     else
115     {
116         AttachAfter(last, element);
117     }
118 }
119
120 /// <summary>
121 /// <para>
122 /// Detaches the element.
123 /// </para>
124 /// <para></para>
125 /// </summary>
126 /// <param name="element">
127 /// <para>The element.</para>
128 /// <para></para>
129 /// </param>
130 public void Detach(TElement element)
131 {
132     var elementPrevious = GetPrevious(element);
133     var elementNext = GetNext(element);
134     if (AreEqual(elementNext, element))
135     {
136         SetFirst(Zero);
137         SetLast(Zero);
138     }
139     else
140     {
141         SetNext(elementPrevious, elementNext);
142         SetPrevious(elementNext, elementPrevious);
143         if (AreEqual(element, GetFirst()))
144         {
145             SetFirst(elementNext);
146         }
147         if (AreEqual(element, GetLast()))
148         {
149             SetLast(elementPrevious);
150         }
151     }
152     SetPrevious(element, Zero);
153     SetNext(element, Zero);
154     DecrementSize();
155 }
156 }
157 }

```

1.3 ./csharp/Platform.Collections.Methods/Lists/AbsoluteDoublyLinkedListMethodsBase.cs

```

1 using System.Runtime.CompilerServices;
2
3 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
5 namespace Platform.Collections.Methods.Lists
6 {
7     /// <summary>
8     /// <para>
9     /// Represents the absolute doubly linked list methods base.
10    /// </para>
11    /// <para></para>

```

```

12  /// </summary>
13  /// <seealso cref="DoublyLinkedListMethodsBase{TElement}"/>
14  public abstract class AbsoluteDoublyLinkedListMethodsBase<TElement> :
    ↳ DoublyLinkedListMethodsBase<TElement>
15  {
16      /// <summary>
17      /// <para>
18      /// Gets the first.
19      /// </para>
20      /// <para></para>
21      /// </summary>
22      /// <returns>
23      /// <para>The element</para>
24      /// <para></para>
25      /// </returns>
26      [MethodImpl(MethodImplOptions.AggressiveInlining)]
27      protected abstract TElement GetFirst();
28
29      /// <summary>
30      /// <para>
31      /// Gets the last.
32      /// </para>
33      /// <para></para>
34      /// </summary>
35      /// <returns>
36      /// <para>The element</para>
37      /// <para></para>
38      /// </returns>
39      [MethodImpl(MethodImplOptions.AggressiveInlining)]
40      protected abstract TElement GetLast();
41
42      /// <summary>
43      /// <para>
44      /// Gets the size.
45      /// </para>
46      /// <para></para>
47      /// </summary>
48      /// <returns>
49      /// <para>The element</para>
50      /// <para></para>
51      /// </returns>
52      [MethodImpl(MethodImplOptions.AggressiveInlining)]
53      protected abstract TElement GetSize();
54
55      /// <summary>
56      /// <para>
57      /// Sets the first using the specified element.
58      /// </para>
59      /// <para></para>
60      /// </summary>
61      /// <param name="element">
62      /// <para>The element.</para>
63      /// <para></para>
64      /// </param>
65      [MethodImpl(MethodImplOptions.AggressiveInlining)]
66      protected abstract void SetFirst(TElement element);
67
68      /// <summary>
69      /// <para>
70      /// Sets the last using the specified element.
71      /// </para>
72      /// <para></para>
73      /// </summary>
74      /// <param name="element">
75      /// <para>The element.</para>
76      /// <para></para>
77      /// </param>
78      [MethodImpl(MethodImplOptions.AggressiveInlining)]
79      protected abstract void SetLast(TElement element);
80
81      /// <summary>
82      /// <para>
83      /// Sets the size using the specified size.
84      /// </para>
85      /// <para></para>
86      /// </summary>
87      /// <param name="size">
88      /// <para>The size.</para>

```

```

89     /// <para></para>
90     /// </param>
91     [MethodImpl(MethodImplOptions.AggressiveInlining)]
92     protected abstract void SetSize(TElement size);
93
94     /// <summary>
95     /// <para>
96     /// Increments the size.
97     /// </para>
98     /// <para></para>
99     /// </summary>
100    [MethodImpl(MethodImplOptions.AggressiveInlining)]
101    protected void IncrementSize() => SetSize(Increment(GetSize()));
102
103    /// <summary>
104    /// <para>
105    /// Decrements the size.
106    /// </para>
107    /// <para></para>
108    /// </summary>
109    [MethodImpl(MethodImplOptions.AggressiveInlining)]
110    protected void DecrementSize() => SetSize(Decrement(GetSize()));
111 }
112 }

```

1.4 ./csharp/Platform.Collections.Methods/Lists/AbsoluteOpenDoublyLinkedListMethods.cs

```

1  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
2
3  namespace Platform.Collections.Methods.Lists
4  {
5      /// <summary>
6      /// <para>
7      /// Represents the absolute open doubly linked list methods.
8      /// </para>
9      /// <para></para>
10     /// </summary>
11     /// <seealso cref="AbsoluteDoublyLinkedListMethodsBase{TElement}"/>
12     public abstract class AbsoluteOpenDoublyLinkedListMethods<TElement> :
13         ↳ AbsoluteDoublyLinkedListMethodsBase<TElement>
14     {
15         /// <summary>
16         /// <para>
17         /// Attaches the before using the specified base element.
18         /// </para>
19         /// <para></para>
20         /// </summary>
21         /// <param name="baseElement">
22         /// <para>The base element.</para>
23         /// <para></para>
24         /// </param>
25         /// <param name="newElement">
26         /// <para>The new element.</para>
27         /// <para></para>
28         /// </param>
29         public void AttachBefore(TElement baseElement, TElement newElement)
30         {
31             var baseElementPrevious = GetPrevious(baseElement);
32             SetPrevious(newElement, baseElementPrevious);
33             SetNext(newElement, baseElement);
34             if (EqualToZero(baseElementPrevious))
35             {
36                 SetFirst(newElement);
37             }
38             else
39             {
40                 SetNext(baseElementPrevious, newElement);
41             }
42             SetPrevious(baseElement, newElement);
43             IncrementSize();
44         }
45
46         /// <summary>
47         /// <para>
48         /// Attaches the after using the specified base element.
49         /// </para>
50         /// <para></para>
51         /// </summary>
52         /// <param name="baseElement">

```

```

52  /// <para>The base element.</para>
53  /// <para></para>
54  /// </param>
55  /// <param name="newElement">
56  /// <para>The new element.</para>
57  /// <para></para>
58  /// </param>
59  public void AttachAfter(TElement baseElement, TElement newElement)
60  {
61      var baseElementNext = GetNext(baseElement);
62      SetPrevious(newElement, baseElement);
63      SetNext(newElement, baseElementNext);
64      if (EqualToZero(baseElementNext))
65      {
66          SetLast(newElement);
67      }
68      else
69      {
70          SetPrevious(baseElementNext, newElement);
71      }
72      SetNext(baseElement, newElement);
73      IncrementSize();
74  }
75
76  /// <summary>
77  /// <para>
78  /// Attaches the as first using the specified element.
79  /// </para>
80  /// <para></para>
81  /// </summary>
82  /// <param name="element">
83  /// <para>The element.</para>
84  /// <para></para>
85  /// </param>
86  public void AttachAsFirst(TElement element)
87  {
88      var first = GetFirst();
89      if (EqualToZero(first))
90      {
91          SetFirst(element);
92          SetLast(element);
93          SetPrevious(element, Zero);
94          SetNext(element, Zero);
95          IncrementSize();
96      }
97      else
98      {
99          AttachBefore(first, element);
100     }
101 }
102
103 /// <summary>
104 /// <para>
105 /// Attaches the as last using the specified element.
106 /// </para>
107 /// <para></para>
108 /// </summary>
109 /// <param name="element">
110 /// <para>The element.</para>
111 /// <para></para>
112 /// </param>
113 public void AttachAsLast(TElement element)
114 {
115     var last = GetLast();
116     if (EqualToZero(last))
117     {
118         AttachAsFirst(element);
119     }
120     else
121     {
122         AttachAfter(last, element);
123     }
124 }
125
126 /// <summary>
127 /// <para>
128 /// Detaches the element.
129 /// </para>

```



```

130     /// <para></para>
131     /// </summary>
132     /// <param name="element">
133     /// <para>The element.</para>
134     /// <para></para>
135     /// </param>
136     public void Detach(TElement element)
137     {
138         var elementPrevious = GetPrevious(element);
139         var elementNext = GetNext(element);
140         if (EqualToZero(elementPrevious))
141         {
142             SetFirst(elementNext);
143         }
144         else
145         {
146             SetNext(elementPrevious, elementNext);
147         }
148         if (EqualToZero(elementNext))
149         {
150             SetLast(elementPrevious);
151         }
152         else
153         {
154             SetPrevious(elementNext, elementPrevious);
155         }
156         SetPrevious(element, Zero);
157         SetNext(element, Zero);
158         DecrementSize();
159     }
160 }
161 }

```

1.5 ./csharp/Platform.Collections.Methods/Lists/DoublyLinkedListMethodsBase.cs

```

1  using System.Runtime.CompilerServices;
2
3  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
5  namespace Platform.Collections.Methods.Lists
6  {
7      /// <remarks>
8      /// Based on <a href="https://en.wikipedia.org/wiki/Doubly_linked_list">doubly linked
9      ↪ list</a> implementation.
10     /// </remarks>
11     public abstract class DoublyLinkedListMethodsBase<TElement> :
12     ↪ GenericCollectionMethodsBase<TElement>
13     {
14         /// <summary>
15         /// <para>
16         /// Gets the previous using the specified element.
17         /// </para>
18         /// <para></para>
19         /// </summary>
20         /// <param name="element">
21         /// <para>The element.</para>
22         /// <para></para>
23         /// </param>
24         /// <returns>
25         /// <para>The element</para>
26         /// <para></para>
27         /// </returns>
28         [MethodImpl(MethodImplOptions.AggressiveInlining)]
29         protected abstract TElement GetPrevious(TElement element);
30
31         /// <summary>
32         /// <para>
33         /// Gets the next using the specified element.
34         /// </para>
35         /// <para></para>
36         /// </summary>
37         /// <param name="element">
38         /// <para>The element.</para>
39         /// <para></para>
40         /// </param>
41         /// <returns>
42         /// <para>The element</para>
43         /// <para></para>
44         /// </returns>

```

```

43     [MethodImpl(MethodImplOptions.AggressiveInlining)]
44     protected abstract TElement GetNext(TElement element);
45
46     /// <summary>
47     /// <para>
48     /// Sets the previous using the specified element.
49     /// </para>
50     /// <para></para>
51     /// </summary>
52     /// <param name="element">
53     /// <para>The element.</para>
54     /// <para></para>
55     /// </param>
56     /// <param name="previous">
57     /// <para>The previous.</para>
58     /// <para></para>
59     /// </param>
60     [MethodImpl(MethodImplOptions.AggressiveInlining)]
61     protected abstract void SetPrevious(TElement element, TElement previous);
62
63     /// <summary>
64     /// <para>
65     /// Sets the next using the specified element.
66     /// </para>
67     /// <para></para>
68     /// </summary>
69     /// <param name="element">
70     /// <para>The element.</para>
71     /// <para></para>
72     /// </param>
73     /// <param name="next">
74     /// <para>The next.</para>
75     /// <para></para>
76     /// </param>
77     [MethodImpl(MethodImplOptions.AggressiveInlining)]
78     protected abstract void SetNext(TElement element, TElement next);
79 }
80 }

```

1.6 ./csharp/Platform.Collections.Methods/Lists/RelativeCircularDoublyLinkedListMethods.cs

```

1  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
2
3  namespace Platform.Collections.Methods.Lists
4  {
5      /// <summary>
6      /// <para>
7      /// Represents the relative circular doubly linked list methods.
8      /// </para>
9      /// <para></para>
10     /// </summary>
11     /// <seealso cref="RelativeDoublyLinkedListMethodsBase{TElement}"/>
12     public abstract class RelativeCircularDoublyLinkedListMethods<TElement> :
13         ↳ RelativeDoublyLinkedListMethodsBase<TElement>
14     {
15         /// <summary>
16         /// <para>
17         /// Attaches the before using the specified head element.
18         /// </para>
19         /// <para></para>
20         /// </summary>
21         /// <param name="headElement">
22         /// <para>The head element.</para>
23         /// <para></para>
24         /// </param>
25         /// <param name="baseElement">
26         /// <para>The base element.</para>
27         /// <para></para>
28         /// </param>
29         /// <param name="newElement">
30         /// <para>The new element.</para>
31         /// <para></para>
32         /// </param>
33         public void AttachBefore(TElement headElement, TElement baseElement, TElement newElement)
34         {
35             var baseElementPrevious = GetPrevious(baseElement);
36             SetPrevious(newElement, baseElementPrevious);
37             SetNext(newElement, baseElement);
38             if (AreEqual(baseElement, GetFirst(headElement)))

```

```

38     {
39         SetFirst(headElement, newElement);
40     }
41     SetNext(baseElementPrevious, newElement);
42     SetPrevious(baseElement, newElement);
43     IncrementSize(headElement);
44 }
45
46 /// <summary>
47 /// <para>
48 /// Attaches the after using the specified head element.
49 /// </para>
50 /// <para></para>
51 /// </summary>
52 /// <param name="headElement">
53 /// <para>The head element.</para>
54 /// <para></para>
55 /// </param>
56 /// <param name="baseElement">
57 /// <para>The base element.</para>
58 /// <para></para>
59 /// </param>
60 /// <param name="newElement">
61 /// <para>The new element.</para>
62 /// <para></para>
63 /// </param>
64 public void AttachAfter(TElement headElement, TElement baseElement, TElement newElement)
65 {
66     var baseElementNext = GetNext(baseElement);
67     SetPrevious(newElement, baseElement);
68     SetNext(newElement, baseElementNext);
69     if (AreEqual(baseElement, GetLast(headElement)))
70     {
71         SetLast(headElement, newElement);
72     }
73     SetPrevious(baseElementNext, newElement);
74     SetNext(baseElement, newElement);
75     IncrementSize(headElement);
76 }
77
78 /// <summary>
79 /// <para>
80 /// Attaches the as first using the specified head element.
81 /// </para>
82 /// <para></para>
83 /// </summary>
84 /// <param name="headElement">
85 /// <para>The head element.</para>
86 /// <para></para>
87 /// </param>
88 /// <param name="element">
89 /// <para>The element.</para>
90 /// <para></para>
91 /// </param>
92 public void AttachAsFirst(TElement headElement, TElement element)
93 {
94     var first = GetFirst(headElement);
95     if (EqualToZero(first))
96     {
97         SetFirst(headElement, element);
98         SetLast(headElement, element);
99         SetPrevious(element, element);
100        SetNext(element, element);
101        IncrementSize(headElement);
102    }
103    else
104    {
105        AttachBefore(headElement, first, element);
106    }
107 }
108
109 /// <summary>
110 /// <para>
111 /// Attaches the as last using the specified head element.
112 /// </para>
113 /// <para></para>
114 /// </summary>
115 /// <param name="headElement">

```

```

116     /// <para>The head element.</para>
117     /// <para></para>
118     /// </param>
119     /// <param name="element">
120     /// <para>The element.</para>
121     /// <para></para>
122     /// </param>
123     public void AttachAsLast(TElement headElement, TElement element)
124     {
125         var last = GetLast(headElement);
126         if (EqualToZero(last))
127         {
128             AttachAsFirst(headElement, element);
129         }
130         else
131         {
132             AttachAfter(headElement, last, element);
133         }
134     }
135
136     /// <summary>
137     /// <para>
138     /// Detaches the head element.
139     /// </para>
140     /// <para></para>
141     /// </summary>
142     /// <param name="headElement">
143     /// <para>The head element.</para>
144     /// <para></para>
145     /// </param>
146     /// <param name="element">
147     /// <para>The element.</para>
148     /// <para></para>
149     /// </param>
150     public void Detach(TElement headElement, TElement element)
151     {
152         var elementPrevious = GetPrevious(element);
153         var elementNext = GetNext(element);
154         if (AreEqual(elementNext, element))
155         {
156             SetFirst(headElement, Zero);
157             SetLast(headElement, Zero);
158         }
159         else
160         {
161             SetNext(elementPrevious, elementNext);
162             SetPrevious(elementNext, elementPrevious);
163             if (AreEqual(element, GetFirst(headElement)))
164             {
165                 SetFirst(headElement, elementNext);
166             }
167             if (AreEqual(element, GetLast(headElement)))
168             {
169                 SetLast(headElement, elementPrevious);
170             }
171         }
172         SetPrevious(element, Zero);
173         SetNext(element, Zero);
174         DecrementSize(headElement);
175     }
176 }
177 }

```

1.7 ./csharp/Platform.Collections.Methods/Lists/RelativeDoublyLinkedListMethodsBase.cs

```

1 using System.Runtime.CompilerServices;
2
3 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
5 namespace Platform.Collections.Methods.Lists
6 {
7     /// <summary>
8     /// <para>
9     /// Represents the relative doubly linked list methods base.
10    /// </para>
11    /// <para></para>
12    /// </summary>
13    /// <seealso cref="DoublyLinkedListMethodsBase{TElement}"/>
14    public abstract class RelativeDoublyLinkedListMethodsBase<TElement> :
15        ↳ DoublyLinkedListMethodsBase<TElement>

```

```

15 {
16     /// <summary>
17     /// <para>
18     /// Gets the first using the specified head element.
19     /// </para>
20     /// <para></para>
21     /// </summary>
22     /// <param name="headElement">
23     /// <para>The head element.</para>
24     /// <para></para>
25     /// </param>
26     /// <returns>
27     /// <para>The element</para>
28     /// <para></para>
29     /// </returns>
30     [MethodImpl(MethodImplOptions.AggressiveInlining)]
31     protected abstract TElement GetFirst(TElement headElement);
32
33     /// <summary>
34     /// <para>
35     /// Gets the last using the specified head element.
36     /// </para>
37     /// <para></para>
38     /// </summary>
39     /// <param name="headElement">
40     /// <para>The head element.</para>
41     /// <para></para>
42     /// </param>
43     /// <returns>
44     /// <para>The element</para>
45     /// <para></para>
46     /// </returns>
47     [MethodImpl(MethodImplOptions.AggressiveInlining)]
48     protected abstract TElement GetLast(TElement headElement);
49
50     /// <summary>
51     /// <para>
52     /// Gets the size using the specified head element.
53     /// </para>
54     /// <para></para>
55     /// </summary>
56     /// <param name="headElement">
57     /// <para>The head element.</para>
58     /// <para></para>
59     /// </param>
60     /// <returns>
61     /// <para>The element</para>
62     /// <para></para>
63     /// </returns>
64     [MethodImpl(MethodImplOptions.AggressiveInlining)]
65     protected abstract TElement GetSize(TElement headElement);
66
67     /// <summary>
68     /// <para>
69     /// Sets the first using the specified head element.
70     /// </para>
71     /// <para></para>
72     /// </summary>
73     /// <param name="headElement">
74     /// <para>The head element.</para>
75     /// <para></para>
76     /// </param>
77     /// <param name="element">
78     /// <para>The element.</para>
79     /// <para></para>
80     /// </param>
81     [MethodImpl(MethodImplOptions.AggressiveInlining)]
82     protected abstract void SetFirst(TElement headElement, TElement element);
83
84     /// <summary>
85     /// <para>
86     /// Sets the last using the specified head element.
87     /// </para>
88     /// <para></para>
89     /// </summary>
90     /// <param name="headElement">
91     /// <para>The head element.</para>
92     /// <para></para>

```

```

93     /// </param>
94     /// <param name="element">
95     /// <para>The element.</para>
96     /// <para></para>
97     /// </param>
98     [MethodImpl(MethodImplOptions.AggressiveInlining)]
99     protected abstract void SetLast(TElement headElement, TElement element);
100
101     /// <summary>
102     /// <para>
103     /// Sets the size using the specified head element.
104     /// </para>
105     /// <para></para>
106     /// </summary>
107     /// <param name="headElement">
108     /// <para>The head element.</para>
109     /// <para></para>
110     /// </param>
111     /// <param name="size">
112     /// <para>The size.</para>
113     /// <para></para>
114     /// </param>
115     [MethodImpl(MethodImplOptions.AggressiveInlining)]
116     protected abstract void SetSize(TElement headElement, TElement size);
117
118     /// <summary>
119     /// <para>
120     /// Increments the size using the specified head element.
121     /// </para>
122     /// <para></para>
123     /// </summary>
124     /// <param name="headElement">
125     /// <para>The head element.</para>
126     /// <para></para>
127     /// </param>
128     [MethodImpl(MethodImplOptions.AggressiveInlining)]
129     protected void IncrementSize(TElement headElement) => SetSize(headElement,
130         ↪ Increment(GetSize(headElement)));
131
132     /// <summary>
133     /// <para>
134     /// Decrements the size using the specified head element.
135     /// </para>
136     /// <para></para>
137     /// </summary>
138     /// <param name="headElement">
139     /// <para>The head element.</para>
140     /// <para></para>
141     /// </param>
142     [MethodImpl(MethodImplOptions.AggressiveInlining)]
143     protected void DecrementSize(TElement headElement) => SetSize(headElement,
144         ↪ Decrement(GetSize(headElement)));
145 }
146 }

```

1.8 ./csharp/Platform.Collections.Methods/Lists/RelativeOpenDoublyLinkedListMethods.cs

```

1  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
2
3  namespace Platform.Collections.Methods.Lists
4  {
5      /// <summary>
6      /// <para>
7      /// Represents the relative open doubly linked list methods.
8      /// </para>
9      /// <para></para>
10     /// </summary>
11     /// <seealso cref="RelativeDoublyLinkedListMethodsBase{TElement}"/>
12     public abstract class RelativeOpenDoublyLinkedListMethods<TElement> :
13         ↪ RelativeDoublyLinkedListMethodsBase<TElement>
14     {
15         /// <summary>
16         /// <para>
17         /// Attaches the before using the specified head element.
18         /// </para>
19         /// <para></para>
20         /// </summary>
21         /// <param name="headElement">
22         /// <para>The head element.</para>
23         /// <para></para>
24         /// </param>

```

```

22     /// <para></para>
23     /// </param>
24     /// <param name="baseElement">
25     /// <para>The base element.</para>
26     /// <para></para>
27     /// </param>
28     /// <param name="newElement">
29     /// <para>The new element.</para>
30     /// <para></para>
31     /// </param>
32     public void AttachBefore(TElement headElement, TElement baseElement, TElement newElement)
33     {
34         var baseElementPrevious = GetPrevious(baseElement);
35         SetPrevious(newElement, baseElementPrevious);
36         SetNext(newElement, baseElement);
37         if (EqualToZero(baseElementPrevious))
38         {
39             SetFirst(headElement, newElement);
40         }
41         else
42         {
43             SetNext(baseElementPrevious, newElement);
44         }
45         SetPrevious(baseElement, newElement);
46         IncrementSize(headElement);
47     }
48
49     /// <summary>
50     /// <para>
51     /// Attaches the after using the specified head element.
52     /// </para>
53     /// <para></para>
54     /// </summary>
55     /// <param name="headElement">
56     /// <para>The head element.</para>
57     /// <para></para>
58     /// </param>
59     /// <param name="baseElement">
60     /// <para>The base element.</para>
61     /// <para></para>
62     /// </param>
63     /// <param name="newElement">
64     /// <para>The new element.</para>
65     /// <para></para>
66     /// </param>
67     public void AttachAfter(TElement headElement, TElement baseElement, TElement newElement)
68     {
69         var baseElementNext = GetNext(baseElement);
70         SetPrevious(newElement, baseElement);
71         SetNext(newElement, baseElementNext);
72         if (EqualToZero(baseElementNext))
73         {
74             SetLast(headElement, newElement);
75         }
76         else
77         {
78             SetPrevious(baseElementNext, newElement);
79         }
80         SetNext(baseElement, newElement);
81         IncrementSize(headElement);
82     }
83
84     /// <summary>
85     /// <para>
86     /// Attaches the as first using the specified head element.
87     /// </para>
88     /// <para></para>
89     /// </summary>
90     /// <param name="headElement">
91     /// <para>The head element.</para>
92     /// <para></para>
93     /// </param>
94     /// <param name="element">
95     /// <para>The element.</para>
96     /// <para></para>
97     /// </param>
98     public void AttachAsFirst(TElement headElement, TElement element)
99     {

```

```

100     var first = GetFirst(headElement);
101     if (EqualToZero(first))
102     {
103         SetFirst(headElement, element);
104         SetLast(headElement, element);
105         SetPrevious(element, Zero);
106         SetNext(element, Zero);
107         IncrementSize(headElement);
108     }
109     else
110     {
111         AttachBefore(headElement, first, element);
112     }
113 }
114
115 /// <summary>
116 /// <para>
117 /// Attaches the as last using the specified head element.
118 /// </para>
119 /// <para></para>
120 /// </summary>
121 /// <param name="headElement">
122 /// <para>The head element.</para>
123 /// <para></para>
124 /// </param>
125 /// <param name="element">
126 /// <para>The element.</para>
127 /// <para></para>
128 /// </param>
129 public void AttachAsLast(TElement headElement, TElement element)
130 {
131     var last = GetLast(headElement);
132     if (EqualToZero(last))
133     {
134         AttachAsFirst(headElement, element);
135     }
136     else
137     {
138         AttachAfter(headElement, last, element);
139     }
140 }
141
142 /// <summary>
143 /// <para>
144 /// Detaches the head element.
145 /// </para>
146 /// <para></para>
147 /// </summary>
148 /// <param name="headElement">
149 /// <para>The head element.</para>
150 /// <para></para>
151 /// </param>
152 /// <param name="element">
153 /// <para>The element.</para>
154 /// <para></para>
155 /// </param>
156 public void Detach(TElement headElement, TElement element)
157 {
158     var elementPrevious = GetPrevious(element);
159     var elementNext = GetNext(element);
160     if (EqualToZero(elementPrevious))
161     {
162         SetFirst(headElement, elementNext);
163     }
164     else
165     {
166         SetNext(elementPrevious, elementNext);
167     }
168     if (EqualToZero(elementNext))
169     {
170         SetLast(headElement, elementPrevious);
171     }
172     else
173     {
174         SetPrevious(elementNext, elementPrevious);
175     }
176     SetPrevious(element, Zero);
177     SetNext(element, Zero);

```



```

178         DecrementSize(headElement);
179     }
180 }
181 }

```

1.9 ./csharp/Platform.Collections.Methods/Trees/RecursionlessSizeBalancedTreeMethods.cs

```

1  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
2
3  namespace Platform.Collections.Methods.Trees
4  {
5      /// <summary>
6      /// <para>
7      /// Represents the recursionless size balanced tree methods.
8      /// </para>
9      /// <para></para>
10     /// </summary>
11     /// <seealso cref="SizedBinaryTreeMethodsBase{TElement}"/>
12     public abstract class RecursionlessSizeBalancedTreeMethods<TElement> :
13         ↳ SizedBinaryTreeMethodsBase<TElement>
14     {
15         /// <summary>
16         /// <para>
17         /// Attaches the core using the specified root.
18         /// </para>
19         /// <para></para>
20         /// </summary>
21         /// <param name="root">
22         /// <para>The root.</para>
23         /// <para></para>
24         /// </param>
25         /// <param name="node">
26         /// <para>The node.</para>
27         /// <para></para>
28         /// </param>
29         protected override void AttachCore(ref TElement root, TElement node)
30         {
31             while (true)
32             {
33                 ref var left = ref GetLeftReference(root);
34                 var leftSize = GetSizeOrZero(left);
35                 ref var right = ref GetRightReference(root);
36                 var rightSize = GetSizeOrZero(right);
37                 if (FirstIsToTheLeftOfSecond(node, root)) // node.Key less than root.Key
38                 {
39                     if (EqualToZero(left))
40                     {
41                         IncrementSize(root);
42                         SetSize(node, One);
43                         left = node;
44                         return;
45                     }
46                     if (FirstIsToTheLeftOfSecond(node, left)) // node.Key less than left.Key
47                     {
48                         if (GreaterThan(Increment(leftSize), rightSize))
49                         {
50                             RightRotate(ref root);
51                         }
52                         else
53                         {
54                             IncrementSize(root);
55                             root = ref left;
56                         }
57                     }
58                     else // node.Key greater than left.Key
59                     {
60                         var leftRightSize = GetSizeOrZero(GetRight(left));
61                         if (GreaterThan(Increment(leftRightSize), rightSize))
62                         {
63                             if (EqualToZero(leftRightSize) && EqualToZero(rightSize))
64                             {
65                                 SetLeft(node, left);
66                                 SetRight(node, root);
67                                 SetSize(node, Add(leftSize, Two)); // Two (2) - node the size of
68                                 ↳ root and a node itself
69                                 SetLeft(root, Zero);
70                                 SetSize(root, One);
71                                 root = node;
72                                 return;

```

```

71         }
72         LeftRotate(ref left);
73         RightRotate(ref root);
74     }
75     else
76     {
77         IncrementSize(root);
78         root = ref left;
79     }
80 }
81 }
82 else // node.Key greater than root.Key
83 {
84     if (EqualToZero(right))
85     {
86         IncrementSize(root);
87         SetSize(node, One);
88         right = node;
89         return;
90     }
91     if (FirstIsToTheRightOfSecond(node, right)) // node.Key greater than
92     ↪ right.Key
93     {
94         if (GreaterThan(Increment(rightSize), leftSize))
95         {
96             LeftRotate(ref root);
97         }
98         else
99         {
100             IncrementSize(root);
101             root = ref right;
102         }
103     }
104     else // node.Key less than right.Key
105     {
106         var rightLeftSize = GetSizeOrZero(GetLeft(right));
107         if (GreaterThan(Increment(rightLeftSize), leftSize))
108         {
109             if (EqualToZero(rightLeftSize) && EqualToZero(leftSize))
110             {
111                 SetLeft(node, root);
112                 SetRight(node, right);
113                 SetSize(node, Add(rightSize, Two)); // Two (2) - node the size
114                 ↪ of root and a node itself
115                 SetRight(root, Zero);
116                 SetSize(root, One);
117                 root = node;
118                 return;
119             }
120             RightRotate(ref right);
121             LeftRotate(ref root);
122         }
123         else
124         {
125             IncrementSize(root);
126             root = ref right;
127         }
128     }
129 }
130 }
131
132 /// <summary>
133 /// <para>
134 /// Detaches the core using the specified root.
135 /// </para>
136 /// <para></para>
137 /// </summary>
138 /// <param name="root">
139 /// <para>The root.</para>
140 /// <para></para>
141 /// </param>
142 /// <param name="node">
143 /// <para>The node.</para>
144 /// <para></para>
145 /// </param>
146 protected override void DetachCore(ref TElement root, TElement node)
147 {

```

```

147 while (true)
148 {
149     ref var left = ref GetLeftReference(root);
150     var leftSize = GetSizeOrZero(left);
151     ref var right = ref GetRightReference(root);
152     var rightSize = GetSizeOrZero(right);
153     if (FirstIsToTheLeftOfSecond(node, root)) // node.Key less than root.Key
154     {
155         var decrementedLeftSize = Decrement(leftSize);
156         if (GreaterThan(GetSizeOrZero(GetRightOrDefault(right)),
157             ↪ decrementedLeftSize))
158         {
159             LeftRotate(ref root);
160         }
161         else if (GreaterThan(GetSizeOrZero(GetLeftOrDefault(right)),
162             ↪ decrementedLeftSize))
163         {
164             RightRotate(ref right);
165             LeftRotate(ref root);
166         }
167         else
168         {
169             DecrementSize(root);
170             root = ref left;
171         }
172     }
173     else if (FirstIsToTheRightOfSecond(node, root)) // node.Key greater than root.Key
174     {
175         var decrementedRightSize = Decrement(rightSize);
176         if (GreaterThan(GetSizeOrZero(GetLeftOrDefault(left)), decrementedRightSize))
177         {
178             RightRotate(ref root);
179         }
180         else if (GreaterThan(GetSizeOrZero(GetRightOrDefault(left)),
181             ↪ decrementedRightSize))
182         {
183             LeftRotate(ref left);
184             RightRotate(ref root);
185         }
186         else
187         {
188             DecrementSize(root);
189             root = ref right;
190         }
191     }
192     else // key equals to root.Key
193     {
194         if (GreaterThanZero(leftSize) && GreaterThanZero(rightSize))
195         {
196             TElement replacement;
197             if (GreaterThan(leftSize, rightSize))
198             {
199                 replacement = GetRighttest(left);
200                 DetachCore(ref left, replacement);
201             }
202             else
203             {
204                 replacement = GetLefttest(right);
205                 DetachCore(ref right, replacement);
206             }
207             SetLeft(replacement, left);
208             SetRight(replacement, right);
209             SetSize(replacement, Add(leftSize, rightSize));
210             root = replacement;
211         }
212         else if (GreaterThanZero(leftSize))
213         {
214             root = left;
215         }
216         else if (GreaterThanZero(rightSize))
217         {
218             root = right;
219         }
220         else
221         {
222             root = Zero;
223         }
224     }
225     ClearNode(node);

```

```

222         return;
223     }
224 }
225 }
226 }
227 }

```

1.10 ./csharp/Platform.Collections.Methods/Trees/SizeBalancedTreeMethods.cs

```

1  using System;
2
3  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
5  namespace Platform.Collections.Methods.Trees
6  {
7      /// <summary>
8      /// <para>
9      /// Represents the size balanced tree methods.
10     /// </para>
11     /// <para></para>
12     /// </summary>
13     /// <seealso cref="SizedBinaryTreeMethodsBase{TElement}"/>
14     public abstract class SizeBalancedTreeMethods<TElement> :
15         ↳ SizedBinaryTreeMethodsBase<TElement>
16     {
17         /// <summary>
18         /// <para>
19         /// Attaches the core using the specified root.
20         /// </para>
21         /// <para></para>
22         /// </summary>
23         /// <param name="root">
24         /// <para>The root.</para>
25         /// <para></para>
26         /// </param>
27         /// <param name="node">
28         /// <para>The node.</para>
29         /// <para></para>
30         /// </param>
31         protected override void AttachCore(ref TElement root, TElement node)
32         {
33             if (EqualToZero(root))
34             {
35                 root = node;
36                 IncrementSize(root);
37             }
38             else
39             {
40                 IncrementSize(root);
41                 if (FirstIsToTheLeftOfSecond(node, root))
42                 {
43                     AttachCore(ref GetLeftReference(root), node);
44                     LeftMaintain(ref root);
45                 }
46                 else
47                 {
48                     AttachCore(ref GetRightReference(root), node);
49                     RightMaintain(ref root);
50                 }
51             }
52         }
53
54         /// <summary>
55         /// <para>
56         /// Detaches the core using the specified root.
57         /// </para>
58         /// <para></para>
59         /// </summary>
60         /// <param name="root">
61         /// <para>The root.</para>
62         /// <para></para>
63         /// </param>
64         /// <param name="nodeToDetach">
65         /// <para>The node to detach.</para>
66         /// <para></para>
67         /// </param>
68         /// <exception cref="InvalidOperationException">
69         /// <para>Duplicate link found in the tree.</para>
70         /// <para></para>

```

```

70  /// </exception>
71  protected override void DetachCore(ref TElement root, TElement nodeToDetach)
72  {
73      ref var currentNode = ref root;
74      ref var parent = ref root;
75      var replacementNode = Zero;
76      while (!AreEqual(currentNode, nodeToDetach))
77      {
78          DecrementSize(currentNode);
79          if (FirstIsToTheLeftOfSecond(nodeToDetach, currentNode))
80          {
81              parent = ref currentNode;
82              currentNode = ref GetLeftReference(currentNode);
83          }
84          else if (FirstIsToTheRightOfSecond(nodeToDetach, currentNode))
85          {
86              parent = ref currentNode;
87              currentNode = ref GetRightReference(currentNode);
88          }
89          else
90          {
91              throw new InvalidOperationException("Duplicate link found in the tree.");
92          }
93      }
94      var nodeToDetachLeft = GetLeft(nodeToDetach);
95      var node = GetRight(nodeToDetach);
96      if (!EqualToZero(nodeToDetachLeft) && !EqualToZero(node))
97      {
98          var lefttestNode = GetLefttest(node);
99          DetachCore(ref GetRightReference(nodeToDetach), lefttestNode);
100         SetLeft(lefttestNode, nodeToDetachLeft);
101         node = GetRight(nodeToDetach);
102         if (!EqualToZero(node))
103         {
104             SetRight(lefttestNode, node);
105             SetSize(lefttestNode, Increment(Add(GetSize(nodeToDetachLeft),
106                 ↪ GetSize(node))));
107         }
108         else
109         {
110             SetSize(lefttestNode, Increment(GetSize(nodeToDetachLeft)));
111         }
112         replacementNode = lefttestNode;
113     }
114     else if (!EqualToZero(nodeToDetachLeft))
115     {
116         replacementNode = nodeToDetachLeft;
117     }
118     else if (!EqualToZero(node))
119     {
120         replacementNode = node;
121     }
122     if (AreEqual(root, nodeToDetach))
123     {
124         root = replacementNode;
125     }
126     else if (AreEqual(GetLeft(parent), nodeToDetach))
127     {
128         SetLeft(parent, replacementNode);
129     }
130     else if (AreEqual(GetRight(parent), nodeToDetach))
131     {
132         SetRight(parent, replacementNode);
133     }
134     ClearNode(nodeToDetach);
135 }
136
137 /// <summary>
138 /// <para>
139 /// Lefts the maintain using the specified root.
140 /// </para>
141 /// <para></para>
142 /// </summary>
143 /// <param name="root">
144 /// <para>The root.</para>
145 /// </param>
146 private void LeftMaintain(ref TElement root)

```

```

147 {
148     if (!EqualToZero(root))
149     {
150         var rootLeftNode = GetLeft(root);
151         if (!EqualToZero(rootLeftNode))
152         {
153             var rootRightNode = GetRight(root);
154             var rootRightNodeSize = GetSize(rootRightNode);
155             var rootLeftNodeLeftNode = GetLeft(rootLeftNode);
156             if (!EqualToZero(rootLeftNodeLeftNode) &&
157                 (EqualToZero(rootRightNode) ||
158                  ⇨ GreaterThan(GetSize(rootLeftNodeLeftNode), rootRightNodeSize)))
159             {
160                 RightRotate(ref root);
161             }
162             else
163             {
164                 var rootLeftNodeRightNode = GetRight(rootLeftNode);
165                 if (!EqualToZero(rootLeftNodeRightNode) &&
166                     (EqualToZero(rootRightNode) ||
167                      ⇨ GreaterThan(GetSize(rootLeftNodeRightNode), rootRightNodeSize)))
168                 {
169                     LeftRotate(ref GetLeftReference(root));
170                     RightRotate(ref root);
171                 }
172                 else
173                 {
174                     return;
175                 }
176             }
177             LeftMaintain(ref GetLeftReference(root));
178             RightMaintain(ref GetRightReference(root));
179             LeftMaintain(ref root);
180             RightMaintain(ref root);
181         }
182     }
183     /// <summary>
184     /// <para>
185     /// Rights the maintain using the specified root.
186     /// </para>
187     /// <para></para>
188     /// </summary>
189     /// <param name="root">
190     /// <para>The root.</para>
191     /// <para></para>
192     /// </param>
193     private void RightMaintain(ref TElement root)
194     {
195         if (!EqualToZero(root))
196         {
197             var rootRightNode = GetRight(root);
198             if (!EqualToZero(rootRightNode))
199             {
200                 var rootLeftNode = GetLeft(root);
201                 var rootLeftNodeSize = GetSize(rootLeftNode);
202                 var rootRightNodeRightNode = GetRight(rootRightNode);
203                 if (!EqualToZero(rootRightNodeRightNode) &&
204                     (EqualToZero(rootLeftNode) ||
205                      ⇨ GreaterThan(GetSize(rootRightNodeRightNode), rootLeftNodeSize)))
206                 {
207                     LeftRotate(ref root);
208                 }
209                 else
210                 {
211                     var rootRightNodeLeftNode = GetLeft(rootRightNode);
212                     if (!EqualToZero(rootRightNodeLeftNode) &&
213                         (EqualToZero(rootLeftNode) ||
214                          ⇨ GreaterThan(GetSize(rootRightNodeLeftNode), rootLeftNodeSize)))
215                     {
216                         RightRotate(ref GetRightReference(root));
217                         LeftRotate(ref root);
218                     }
219                     else
220                     {
221                         return;
222                     }
223                 }
224             }
225         }
226     }

```

```

221         }
222         LeftMaintain(ref GetLeftReference(root));
223         RightMaintain(ref GetRightReference(root));
224         LeftMaintain(ref root);
225         RightMaintain(ref root);
226     }
227 }
228 }
229 }
230 }

```

1.11 ./csharp/Platform.Collections.Methods/Trees/SizedAndThreadedAVLBalancedTreeMethods.cs

```

1  using System;
2  using System.Runtime.CompilerServices;
3  using System.Text;
4  #if USEARRAYPOOL
5  using Platform.Collections;
6  #endif
7  using Platform.Reflection;
8
9  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
10
11 namespace Platform.Collections.Methods.Trees
12 {
13     /// <summary>
14     /// Combination of Size, Height (AVL), and threads.
15     /// </summary>
16     /// <remarks>
17     /// Based on: <a href="https://github.com/programatom/TreeLib/blob/master/TreeLib/TreeLib/G_
18     ↪ enerated/AVLTreeList.cs">TreeLib.AVLTreeList</a>.
19     /// Which itself based on: <a
20     ↪ href="https://github.com/GNOME/glib/blob/master/glib/gtree.c">GNOME/glib/gtree</a>.
21     /// </remarks>
22     public abstract class SizedAndThreadedAVLBalancedTreeMethods<TElement> :
23     ↪ SizedBinaryTreeMethodsBase<TElement>
24     {
25         /// <summary>
26         /// <para>
27         /// The bytes size.
28         /// </para>
29         /// <para></para>
30         /// </summary>
31         private static readonly int _maxPath = 11 * NumericType<TElement>.BytesSize + 4;
32
33         /// <summary>
34         /// <para>
35         /// Gets the rightest using the specified current.
36         /// </para>
37         /// <para></para>
38         /// </summary>
39         /// <param name="current">
40         /// <para>The current.</para>
41         /// <para></para>
42         /// </param>
43         /// <returns>
44         /// <para>The current.</para>
45         /// <para></para>
46         /// </returns>
47         [MethodImpl(MethodImplOptions.AggressiveInlining)]
48         protected override TElement GetRightest(TElement current)
49         {
50             var currentRight = GetRightOrDefault(current);
51             while (!EqualToZero(currentRight))
52             {
53                 current = currentRight;
54                 currentRight = GetRightOrDefault(current);
55             }
56             return current;
57         }
58
59         /// <summary>
60         /// <para>
61         /// Gets the leftest using the specified current.
62         /// </para>
63         /// <para></para>
64         /// </summary>
65         /// <param name="current">
66         /// <para>The current.</para>
67         /// <para></para>
68         /// </param>
69         /// <returns>
70         /// <para>The current.</para>
71         /// <para></para>
72         /// </returns>
73         [MethodImpl(MethodImplOptions.AggressiveInlining)]
74         protected override TElement GetLeftest(TElement current)
75         {
76             var currentLeft = GetLeftOrDefault(current);
77             while (!EqualToZero(currentLeft))
78             {
79                 current = currentLeft;
80                 currentLeft = GetLeftOrDefault(current);
81             }
82             return current;
83         }
84     }
85 }

```

```

65     /// </param>
66     /// <returns>
67     /// <para>The current.</para>
68     /// <para></para>
69     /// </returns>
70     [MethodImpl(MethodImplOptions.AggressiveInlining)]
71     protected override TElement GetLefttest(TElement current)
72     {
73         var currentLeft = GetLeftOrDefault(current);
74         while (!EqualToZero(currentLeft))
75         {
76             current = currentLeft;
77             currentLeft = GetLeftOrDefault(current);
78         }
79         return current;
80     }
81
82     /// <summary>
83     /// <para>
84     /// Determines whether this instance contains.
85     /// </para>
86     /// <para></para>
87     /// </summary>
88     /// <param name="node">
89     /// <para>The node.</para>
90     /// <para></para>
91     /// </param>
92     /// <param name="root">
93     /// <para>The root.</para>
94     /// <para></para>
95     /// </param>
96     /// <returns>
97     /// <para>The bool</para>
98     /// <para></para>
99     /// </returns>
100     public override bool Contains(TElement node, TElement root)
101     {
102         while (!EqualToZero(root))
103         {
104             if (FirstIsToTheLeftOfSecond(node, root)) // node.Key < root.Key
105             {
106                 root = GetLeftOrDefault(root);
107             }
108             else if (FirstIsToTheRightOfSecond(node, root)) // node.Key > root.Key
109             {
110                 root = GetRightOrDefault(root);
111             }
112             else // node.Key == root.Key
113             {
114                 return true;
115             }
116         }
117         return false;
118     }
119
120     /// <summary>
121     /// <para>
122     /// Prints the node using the specified node.
123     /// </para>
124     /// <para></para>
125     /// </summary>
126     /// <param name="node">
127     /// <para>The node.</para>
128     /// <para></para>
129     /// </param>
130     /// <param name="sb">
131     /// <para>The sb.</para>
132     /// <para></para>
133     /// </param>
134     /// <param name="level">
135     /// <para>The level.</para>
136     /// <para></para>
137     /// </param>
138     protected override void PrintNode(TElement node, StringBuilder sb, int level)
139     {
140         base.PrintNode(node, sb, level);
141         sb.Append(' ');
142         sb.Append(GetLeftIsChild(node) ? 'L' : 'L');

```



```

143         sb.Append(GetRightIsChild(node) ? 'r' : 'R');
144         sb.Append(' ');
145         sb.Append(GetBalance(node));
146     }
147
148     /// <summary>
149     /// <para>
150     /// Increments the balance using the specified node.
151     /// </para>
152     /// <para></para>
153     /// </summary>
154     /// <param name="node">
155     /// <para>The node.</para>
156     /// <para></para>
157     /// </param>
158     [MethodImpl(MethodImplOptions.AggressiveInlining)]
159     protected void IncrementBalance(TElement node) => SetBalance(node,
160         ↪ (sbyte)(GetBalance(node) + 1));
161
162     /// <summary>
163     /// <para>
164     /// Decrements the balance using the specified node.
165     /// </para>
166     /// <para></para>
167     /// </summary>
168     /// <param name="node">
169     /// <para>The node.</para>
170     /// <para></para>
171     /// </param>
172     [MethodImpl(MethodImplOptions.AggressiveInlining)]
173     protected void DecrementBalance(TElement node) => SetBalance(node,
174         ↪ (sbyte)(GetBalance(node) - 1));
175
176     /// <summary>
177     /// <para>
178     /// Gets the left or default using the specified node.
179     /// </para>
180     /// <para></para>
181     /// </summary>
182     /// <param name="node">
183     /// <para>The node.</para>
184     /// <para></para>
185     /// </param>
186     /// <returns>
187     /// <para>The element</para>
188     /// <para></para>
189     /// </returns>
190     [MethodImpl(MethodImplOptions.AggressiveInlining)]
191     protected override TElement GetLeftOrDefault(TElement node) => GetLeftIsChild(node) ?
192         ↪ GetLeft(node) : default;
193
194     /// <summary>
195     /// <para>
196     /// Gets the right or default using the specified node.
197     /// </para>
198     /// <para></para>
199     /// </summary>
200     /// <param name="node">
201     /// <para>The node.</para>
202     /// <para></para>
203     /// </param>
204     /// <returns>
205     /// <para>The element</para>
206     /// <para></para>
207     /// </returns>
208     [MethodImpl(MethodImplOptions.AggressiveInlining)]
209     protected override TElement GetRightOrDefault(TElement node) => GetRightIsChild(node) ?
210         ↪ GetRight(node) : default;
211
212     /// <summary>
213     /// <para>
214     /// Determines whether this instance get left is child.
215     /// </para>
216     /// <para></para>
217     /// </summary>
218     /// <param name="node">
219     /// <para>The node.</para>
220     /// <para></para>
221     /// </param>

```

```

217     /// </param>
218     /// <returns>
219     /// <para>The bool</para>
220     /// <para></para>
221     /// </returns>
222     [MethodImpl(MethodImplOptions.AggressiveInlining)]
223     protected abstract bool GetLeftIsChild(TElement node);
224
225     /// <summary>
226     /// <para>
227     /// Sets the left is child using the specified node.
228     /// </para>
229     /// <para></para>
230     /// </summary>
231     /// <param name="node">
232     /// <para>The node.</para>
233     /// <para></para>
234     /// </param>
235     /// <param name="value">
236     /// <para>The value.</para>
237     /// <para></para>
238     /// </param>
239     [MethodImpl(MethodImplOptions.AggressiveInlining)]
240     protected abstract void SetLeftIsChild(TElement node, bool value);
241
242     /// <summary>
243     /// <para>
244     /// Determines whether this instance get right is child.
245     /// </para>
246     /// <para></para>
247     /// </summary>
248     /// <param name="node">
249     /// <para>The node.</para>
250     /// <para></para>
251     /// </param>
252     /// <returns>
253     /// <para>The bool</para>
254     /// <para></para>
255     /// </returns>
256     [MethodImpl(MethodImplOptions.AggressiveInlining)]
257     protected abstract bool GetRightIsChild(TElement node);
258
259     /// <summary>
260     /// <para>
261     /// Sets the right is child using the specified node.
262     /// </para>
263     /// <para></para>
264     /// </summary>
265     /// <param name="node">
266     /// <para>The node.</para>
267     /// <para></para>
268     /// </param>
269     /// <param name="value">
270     /// <para>The value.</para>
271     /// <para></para>
272     /// </param>
273     [MethodImpl(MethodImplOptions.AggressiveInlining)]
274     protected abstract void SetRightIsChild(TElement node, bool value);
275
276     /// <summary>
277     /// <para>
278     /// Gets the balance using the specified node.
279     /// </para>
280     /// <para></para>
281     /// </summary>
282     /// <param name="node">
283     /// <para>The node.</para>
284     /// <para></para>
285     /// </param>
286     /// <returns>
287     /// <para>The sbyte</para>
288     /// <para></para>
289     /// </returns>
290     [MethodImpl(MethodImplOptions.AggressiveInlining)]
291     protected abstract sbyte GetBalance(TElement node);
292
293     /// <summary>
294     /// <para>

```



```

372         currentNode = GetRight(currentNode);
373     }
374     else
375     {
376         // Threads
377         SetRight(node, GetRight(currentNode));
378         SetLeft(node, currentNode);
379         SetRight(currentNode, node);
380         SetRightIsChild(currentNode, true);
381         IncrementBalance(currentNode);
382         SetSize(node, One);
383         FixSize(currentNode); // Should be incremented already
384         break;
385     }
386 }
387 else
388 {
389     throw new InvalidOperationException("Node with the same key already
390     ↳ attached to a tree.");
391 }
392 // Restore balance. This is the goodness of a non-recursive
393 // implementation, when we are done with balancing we 'break'
394 // the loop and we are done.
395 while (true)
396 {
397     var parent = path[--pathPosition];
398     var isLeftNode = !AreEqual(parent, default) && AreEqual(currentNode,
399     ↳ GetLeft(parent));
400     var currentNodeBalance = GetBalance(currentNode);
401     if (currentNodeBalance < -1 || currentNodeBalance > 1)
402     {
403         currentNode = Balance(currentNode);
404         if (AreEqual(parent, default))
405         {
406             root = currentNode;
407         }
408         else if (isLeftNode)
409         {
410             SetLeft(parent, currentNode);
411             FixSize(parent);
412         }
413         else
414         {
415             SetRight(parent, currentNode);
416             FixSize(parent);
417         }
418     }
419     currentNodeBalance = GetBalance(currentNode);
420     if (currentNodeBalance == 0 || AreEqual(parent, default))
421     {
422         break;
423     }
424     if (isLeftNode)
425     {
426         DecrementBalance(parent);
427     }
428     else
429     {
430         IncrementBalance(parent);
431     }
432     currentNode = parent;
433 }
434 #if USEARRAYPOOL
435     ArrayPool.Free(path);
436 #endif
437 }
438 }
439 /// <summary>
440 /// <para>
441 /// Balances the node.
442 /// </para>
443 /// <para></para>
444 /// </summary>
445 /// <param name="node">
446 /// <para>The node.</para>
447 /// <para></para>

```

```

448     /// </param>
449     /// <returns>
450     /// <para>The element</para>
451     /// <para></para>
452     /// </returns>
453 private TElement Balance(TElement node)
454 {
455     unchecked
456     {
457         var rootBalance = GetBalance(node);
458         if (rootBalance < -1)
459         {
460             var left = GetLeft(node);
461             if (GetBalance(left) > 0)
462             {
463                 SetLeft(node, LeftRotateWithBalance(left));
464                 FixSize(node);
465             }
466             node = RightRotateWithBalance(node);
467         }
468         else if (rootBalance > 1)
469         {
470             var right = GetRight(node);
471             if (GetBalance(right) < 0)
472             {
473                 SetRight(node, RightRotateWithBalance(right));
474                 FixSize(node);
475             }
476             node = LeftRotateWithBalance(node);
477         }
478         return node;
479     }
480 }
481
482     /// <summary>
483     /// <para>
484     /// Lefts the rotate with balance using the specified node.
485     /// </para>
486     /// <para></para>
487     /// </summary>
488     /// <param name="node">
489     /// <para>The node.</para>
490     /// <para></para>
491     /// </param>
492     /// <returns>
493     /// <para>The element</para>
494     /// <para></para>
495     /// </returns>
496 protected TElement LeftRotateWithBalance(TElement node)
497 {
498     unchecked
499     {
500         var right = GetRight(node);
501         if (GetLeftIsChild(right))
502         {
503             SetRight(node, GetLeft(right));
504         }
505         else
506         {
507             SetRightIsChild(node, false);
508             SetLeftIsChild(right, true);
509         }
510         SetLeft(right, node);
511         // Fix size
512         SetSize(right, GetSize(node));
513         FixSize(node);
514         // Fix balance
515         var rootBalance = GetBalance(node);
516         var rightBalance = GetBalance(right);
517         if (rightBalance <= 0)
518         {
519             if (rootBalance >= 1)
520             {
521                 SetBalance(right, (sbyte)(rightBalance - 1));
522             }
523             else
524             {
525                 SetBalance(right, (sbyte)(rootBalance + rightBalance - 2));

```

```

526     }
527     SetBalance(node, (sbyte)(rootBalance - 1));
528 }
529 else
530 {
531     if (rootBalance <= rightBalance)
532     {
533         SetBalance(right, (sbyte)(rootBalance - 2));
534     }
535     else
536     {
537         SetBalance(right, (sbyte)(rightBalance - 1));
538     }
539     SetBalance(node, (sbyte)(rootBalance - rightBalance - 1));
540 }
541 return right;
542 }
543 }
544
545 /// <summary>
546 /// <para>
547 /// Rights the rotate with balance using the specified node.
548 /// </para>
549 /// <para></para>
550 /// </summary>
551 /// <param name="node">
552 /// <para>The node.</para>
553 /// <para></para>
554 /// </param>
555 /// <returns>
556 /// <para>The element</para>
557 /// <para></para>
558 /// </returns>
559 protected TElement RightRotateWithBalance(TElement node)
560 {
561     unchecked
562     {
563         var left = GetLeft(node);
564         if (GetRightIsChild(left))
565         {
566             SetLeft(node, GetRight(left));
567         }
568         else
569         {
570             SetLeftIsChild(node, false);
571             SetRightIsChild(left, true);
572         }
573         SetRight(left, node);
574         // Fix size
575         SetSize(left, GetSize(node));
576         FixSize(node);
577         // Fix balance
578         var rootBalance = GetBalance(node);
579         var leftBalance = GetBalance(left);
580         if (leftBalance <= 0)
581         {
582             if (leftBalance > rootBalance)
583             {
584                 SetBalance(left, (sbyte)(leftBalance + 1));
585             }
586             else
587             {
588                 SetBalance(left, (sbyte)(rootBalance + 2));
589             }
590             SetBalance(node, (sbyte)(rootBalance - leftBalance + 1));
591         }
592         else
593         {
594             if (rootBalance <= -1)
595             {
596                 SetBalance(left, (sbyte)(leftBalance + 1));
597             }
598             else
599             {
600                 SetBalance(left, (sbyte)(rootBalance + leftBalance + 2));
601             }
602             SetBalance(node, (sbyte)(rootBalance + 1));
603         }

```

```

604         return left;
605     }
606 }
607
608 /// <summary>
609 /// <para>
610 /// Gets the next using the specified node.
611 /// </para>
612 /// <para></para>
613 /// </summary>
614 /// <param name="node">
615 /// <para>The node.</para>
616 /// <para></para>
617 /// </param>
618 /// <returns>
619 /// <para>The current.</para>
620 /// <para></para>
621 /// </returns>
622 [MethodImpl(MethodImplOptions.AggressiveInlining)]
623 protected override TElement GetNext(TElement node)
624 {
625     var current = GetRight(node);
626     if (GetRightIsChild(node))
627     {
628         return GetLefttest(current);
629     }
630     return current;
631 }
632
633 /// <summary>
634 /// <para>
635 /// Gets the previous using the specified node.
636 /// </para>
637 /// <para></para>
638 /// </summary>
639 /// <param name="node">
640 /// <para>The node.</para>
641 /// <para></para>
642 /// </param>
643 /// <returns>
644 /// <para>The current.</para>
645 /// <para></para>
646 /// </returns>
647 [MethodImpl(MethodImplOptions.AggressiveInlining)]
648 protected override TElement GetPrevious(TElement node)
649 {
650     var current = GetLeft(node);
651     if (GetLeftIsChild(node))
652     {
653         return GetRighttest(current);
654     }
655     return current;
656 }
657
658 /// <summary>
659 /// <para>
660 /// Detaches the core using the specified root.
661 /// </para>
662 /// <para></para>
663 /// </summary>
664 /// <param name="root">
665 /// <para>The root.</para>
666 /// <para></para>
667 /// </param>
668 /// <param name="node">
669 /// <para>The node.</para>
670 /// <para></para>
671 /// </param>
672 /// <exception cref="InvalidOperationException">
673 /// <para>Cannot find a node.</para>
674 /// <para></para>
675 /// </exception>
676 /// <exception cref="InvalidOperationException">
677 /// <para>Cannot find a node.</para>
678 /// <para></para>
679 /// </exception>
680 protected override void DetachCore(ref TElement root, TElement node)
681 {

```

```

682         unchecked
683     {
684         #if USEARRAYPOOL
685             var path = ArrayPool.Allocate<TElement>(MaxPath);
686             var pathPosition = 0;
687             path[pathPosition++] = default;
688         #else
689             var path = new TElement[_maxPath];
690             var pathPosition = 1;
691         #endif
692         var currentNode = root;
693         while (true)
694         {
695             if (FirstIsToTheLeftOfSecond(node, currentNode))
696             {
697                 if (!GetLeftIsChild(currentNode))
698                 {
699                     throw new InvalidOperationException("Cannot find a node.");
700                 }
701                 DecrementSize(currentNode);
702                 path[pathPosition++] = currentNode;
703                 currentNode = GetLeft(currentNode);
704             }
705             else if (FirstIsToTheRightOfSecond(node, currentNode))
706             {
707                 if (!GetRightIsChild(currentNode))
708                 {
709                     throw new InvalidOperationException("Cannot find a node.");
710                 }
711                 DecrementSize(currentNode);
712                 path[pathPosition++] = currentNode;
713                 currentNode = GetRight(currentNode);
714             }
715             else
716             {
717                 break;
718             }
719         }
720         var parent = path[--pathPosition];
721         var balanceNode = parent;
722         var isLeftNode = !AreEqual(parent, default) && AreEqual(currentNode,
723             ↪ GetLeft(parent));
724         if (!GetLeftIsChild(currentNode))
725         {
726             if (!GetRightIsChild(currentNode)) // node has no children
727             {
728                 if (AreEqual(parent, default))
729                 {
730                     root = Zero;
731                 }
732                 else if (isLeftNode)
733                 {
734                     SetLeftIsChild(parent, false);
735                     SetLeft(parent, GetLeft(currentNode));
736                     IncrementBalance(parent);
737                 }
738                 else
739                 {
740                     SetRightIsChild(parent, false);
741                     SetRight(parent, GetRight(currentNode));
742                     DecrementBalance(parent);
743                 }
744             }
745             else // node has a right child
746             {
747                 var successor = GetNext(currentNode);
748                 SetLeft(successor, GetLeft(currentNode));
749                 var right = GetRight(currentNode);
750                 if (AreEqual(parent, default))
751                 {
752                     root = right;
753                 }
754                 else if (isLeftNode)
755                 {
756                     SetLeft(parent, right);
757                     IncrementBalance(parent);
758                 }
759                 else
760                 {

```



```

760         SetRight(parent, right);
761         DecrementBalance(parent);
762     }
763 }
764 }
765 else // node has a left child
766 {
767     if (!GetRightIsChild(currentNode))
768     {
769         var predecessor = GetPrevious(currentNode);
770         SetRight(predecessor, GetRight(currentNode));
771         var leftValue = GetLeft(currentNode);
772         if (AreEqual(parent, default))
773         {
774             root = leftValue;
775         }
776         else if (isLeftNode)
777         {
778             SetLeft(parent, leftValue);
779             IncrementBalance(parent);
780         }
781         else
782         {
783             SetRight(parent, leftValue);
784             DecrementBalance(parent);
785         }
786     }
787     else // node has a both children (left and right)
788     {
789         var predecessor = GetLeft(currentNode);
790         var successor = GetRight(currentNode);
791         var successorParent = currentNode;
792         int previousPathPosition = ++pathPosition;
793         // find the immediately next node (and its parent)
794         while (GetLeftIsChild(successor))
795         {
796             path[++pathPosition] = successorParent = successor;
797             successor = GetLeft(successor);
798             if (!AreEqual(successorParent, currentNode))
799             {
800                 DecrementSize(successorParent);
801             }
802         }
803         path[previousPathPosition] = successor;
804         balanceNode = path[pathPosition];
805         // remove 'successor' from the tree
806         if (!AreEqual(successorParent, currentNode))
807         {
808             if (!GetRightIsChild(successor))
809             {
810                 SetLeftIsChild(successorParent, false);
811             }
812             else
813             {
814                 SetLeft(successorParent, GetRight(successor));
815             }
816             IncrementBalance(successorParent);
817             SetRightIsChild(successor, true);
818             SetRight(successor, GetRight(currentNode));
819         }
820         else
821         {
822             DecrementBalance(currentNode);
823         }
824         // set the predecessor's successor link to point to the right place
825         while (GetRightIsChild(predecessor))
826         {
827             predecessor = GetRight(predecessor);
828         }
829         SetRight(predecessor, successor);
830         // prepare 'successor' to replace 'node'
831         var left = GetLeft(currentNode);
832         SetLeftIsChild(successor, true);
833         SetLeft(successor, left);
834         SetBalance(successor, GetBalance(currentNode));
835         FixSize(successor);
836         if (AreEqual(parent, default))
837         {

```

```

838         root = successor;
839     }
840     else if (isLeftNode)
841     {
842         SetLeft(parent, successor);
843     }
844     else
845     {
846         SetRight(parent, successor);
847     }
848 }
849 }
850 // restore balance
851 if (!AreEqual(balanceNode, default))
852 {
853     while (true)
854     {
855         var balanceParent = path[--pathPosition];
856         isLeftNode = !AreEqual(balanceParent, default) && AreEqual(balanceNode,
857             ↪ GetLeft(balanceParent));
858         var currentNodeBalance = GetBalance(balanceNode);
859         if (currentNodeBalance < -1 || currentNodeBalance > 1)
860         {
861             balanceNode = Balance(balanceNode);
862             if (AreEqual(balanceParent, default))
863             {
864                 root = balanceNode;
865             }
866             else if (isLeftNode)
867             {
868                 SetLeft(balanceParent, balanceNode);
869             }
870             else
871             {
872                 SetRight(balanceParent, balanceNode);
873             }
874             currentNodeBalance = GetBalance(balanceNode);
875             if (currentNodeBalance != 0 || AreEqual(balanceParent, default))
876             {
877                 break;
878             }
879             if (isLeftNode)
880             {
881                 IncrementBalance(balanceParent);
882             }
883             else
884             {
885                 DecrementBalance(balanceParent);
886             }
887             balanceNode = balanceParent;
888         }
889     }
890     ClearNode(node);
891 #if USEARRAYPOOL
892     ArrayPool.Free(path);
893 #endif
894 }
895 }
896
897 /// <summary>
898 /// <para>
899 /// Clears the node using the specified node.
900 /// </para>
901 /// <para></para>
902 /// </summary>
903 /// <param name="node">
904 /// <para>The node.</para>
905 /// <para></para>
906 /// </param>
907 [MethodImpl(MethodImplOptions.AggressiveInlining)]
908 protected override void ClearNode(TElement node)
909 {
910     SetLeft(node, Zero);
911     SetRight(node, Zero);
912     SetSize(node, Zero);
913     SetLeftIsChild(node, false);
914     SetRightIsChild(node, false);

```

```

915         SetBalance(node, 0);
916     }
917 }
918 }

```

1.12 ./csharp/Platform.Collections.Methods/Trees/SizedBinaryTreeMethodsBase.cs

```

1  ///#define ENABLE_TREE_AUTO_DEBUG_AND_VALIDATION
2
3  using System;
4  using System.Diagnostics;
5  using System.Runtime.CompilerServices;
6  using System.Text;
7  using Platform.Numbers;
8
9  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
10
11 namespace Platform.Collections.Methods.Trees
12 {
13     /// <summary>
14     /// <para>
15     /// Represents the sized binary tree methods base.
16     /// </para>
17     /// <para></para>
18     /// </summary>
19     /// <seealso cref="GenericCollectionMethodsBase{TElement}">/>
20     public abstract class SizedBinaryTreeMethodsBase<TElement> :
21         ↳ GenericCollectionMethodsBase<TElement>
22     {
23         /// <summary>
24         /// <para>
25         /// Gets the left reference using the specified node.
26         /// </para>
27         /// <para></para>
28         /// </summary>
29         /// <param name="node">
30         /// <para>The node.</para>
31         /// <para></para>
32         /// </param>
33         /// <returns>
34         /// <para>The ref element</para>
35         /// <para></para>
36         /// </returns>
37         [MethodImpl(MethodImplOptions.AggressiveInlining)]
38         protected abstract ref TElement GetLeftReference(TElement node);
39
40         /// <summary>
41         /// <para>
42         /// Gets the right reference using the specified node.
43         /// </para>
44         /// <para></para>
45         /// </summary>
46         /// <param name="node">
47         /// <para>The node.</para>
48         /// <para></para>
49         /// </param>
50         /// <returns>
51         /// <para>The ref element</para>
52         /// <para></para>
53         /// </returns>
54         [MethodImpl(MethodImplOptions.AggressiveInlining)]
55         protected abstract ref TElement GetRightReference(TElement node);
56
57         /// <summary>
58         /// <para>
59         /// Gets the left using the specified node.
60         /// </para>
61         /// <para></para>
62         /// </summary>
63         /// <param name="node">
64         /// <para>The node.</para>
65         /// <para></para>
66         /// </param>
67         /// <returns>
68         /// <para>The element</para>
69         /// <para></para>
70         /// </returns>
71         [MethodImpl(MethodImplOptions.AggressiveInlining)]
72         protected abstract TElement GetLeft(TElement node);

```

```

73     /// <summary>
74     /// <para>
75     /// Gets the right using the specified node.
76     /// </para>
77     /// <para></para>
78     /// </summary>
79     /// <param name="node">
80     /// <para>The node.</para>
81     /// <para></para>
82     /// </param>
83     /// <returns>
84     /// <para>The element</para>
85     /// <para></para>
86     /// </returns>
87     [MethodImpl(MethodImplOptions.AggressiveInlining)]
88     protected abstract TElement GetRight(TElement node);
89
90     /// <summary>
91     /// <para>
92     /// Gets the size using the specified node.
93     /// </para>
94     /// <para></para>
95     /// </summary>
96     /// <param name="node">
97     /// <para>The node.</para>
98     /// <para></para>
99     /// </param>
100    /// <returns>
101    /// <para>The element</para>
102    /// <para></para>
103    /// </returns>
104    [MethodImpl(MethodImplOptions.AggressiveInlining)]
105    protected abstract TElement GetSize(TElement node);
106
107    /// <summary>
108    /// <para>
109    /// Sets the left using the specified node.
110    /// </para>
111    /// <para></para>
112    /// </summary>
113    /// <param name="node">
114    /// <para>The node.</para>
115    /// <para></para>
116    /// </param>
117    /// <param name="left">
118    /// <para>The left.</para>
119    /// <para></para>
120    /// </param>
121    [MethodImpl(MethodImplOptions.AggressiveInlining)]
122    protected abstract void SetLeft(TElement node, TElement left);
123
124    /// <summary>
125    /// <para>
126    /// Sets the right using the specified node.
127    /// </para>
128    /// <para></para>
129    /// </summary>
130    /// <param name="node">
131    /// <para>The node.</para>
132    /// <para></para>
133    /// </param>
134    /// <param name="right">
135    /// <para>The right.</para>
136    /// <para></para>
137    /// </param>
138    [MethodImpl(MethodImplOptions.AggressiveInlining)]
139    protected abstract void SetRight(TElement node, TElement right);
140
141    /// <summary>
142    /// <para>
143    /// Sets the size using the specified node.
144    /// </para>
145    /// <para></para>
146    /// </summary>
147    /// <param name="node">
148    /// <para>The node.</para>
149    /// <para></para>
150    /// </param>

```

```

151     /// <param name="size">
152     /// <para>The size.</para>
153     /// <para></para>
154     /// </param>
155     [MethodImpl(MethodImplOptions.AggressiveInlining)]
156     protected abstract void SetSize(TElement node, TElement size);
157
158     /// <summary>
159     /// <para>
160     /// Determines whether this instance first is to the left of second.
161     /// </para>
162     /// <para></para>
163     /// </summary>
164     /// <param name="first">
165     /// <para>The first.</para>
166     /// <para></para>
167     /// </param>
168     /// <param name="second">
169     /// <para>The second.</para>
170     /// <para></para>
171     /// </param>
172     /// <returns>
173     /// <para>The bool</para>
174     /// <para></para>
175     /// </returns>
176     [MethodImpl(MethodImplOptions.AggressiveInlining)]
177     protected abstract bool FirstIsToLeftOfSecond(TElement first, TElement second);
178
179     /// <summary>
180     /// <para>
181     /// Determines whether this instance first is to the right of second.
182     /// </para>
183     /// <para></para>
184     /// </summary>
185     /// <param name="first">
186     /// <para>The first.</para>
187     /// <para></para>
188     /// </param>
189     /// <param name="second">
190     /// <para>The second.</para>
191     /// <para></para>
192     /// </param>
193     /// <returns>
194     /// <para>The bool</para>
195     /// <para></para>
196     /// </returns>
197     [MethodImpl(MethodImplOptions.AggressiveInlining)]
198     protected abstract bool FirstIsToTheRightOfSecond(TElement first, TElement second);
199
200     /// <summary>
201     /// <para>
202     /// Gets the left or default using the specified node.
203     /// </para>
204     /// <para></para>
205     /// </summary>
206     /// <param name="node">
207     /// <para>The node.</para>
208     /// <para></para>
209     /// </param>
210     /// <returns>
211     /// <para>The element</para>
212     /// <para></para>
213     /// </returns>
214     [MethodImpl(MethodImplOptions.AggressiveInlining)]
215     protected virtual TElement GetLeftOrDefault(TElement node) => AreEqual(node, default) ?
        ↳ default : GetLeft(node);
216
217     /// <summary>
218     /// <para>
219     /// Gets the right or default using the specified node.
220     /// </para>
221     /// <para></para>
222     /// </summary>
223     /// <param name="node">
224     /// <para>The node.</para>
225     /// <para></para>
226     /// </param>
227     /// </returns>

```

```

228    /// <para>The element</para>
229    /// <para></para>
230    /// </returns>
231    [MethodImpl(MethodImplOptions.AggressiveInlining)]
232    protected virtual TElement GetRightOrDefault(TElement node) => AreEqual(node, default) ?
    ↪ default : GetRight(node);

233
234    /// <summary>
235    /// <para>
236    /// Increments the size using the specified node.
237    /// </para>
238    /// <para></para>
239    /// </summary>
240    /// <param name="node">
241    /// <para>The node.</para>
242    /// <para></para>
243    /// </param>
244    [MethodImpl(MethodImplOptions.AggressiveInlining)]
245    protected void IncrementSize(TElement node) => SetSize(node, Increment(GetSize(node)));
246
247    /// <summary>
248    /// <para>
249    /// Decrements the size using the specified node.
250    /// </para>
251    /// <para></para>
252    /// </summary>
253    /// <param name="node">
254    /// <para>The node.</para>
255    /// <para></para>
256    /// </param>
257    [MethodImpl(MethodImplOptions.AggressiveInlining)]
258    protected void DecrementSize(TElement node) => SetSize(node, Decrement(GetSize(node)));
259
260    /// <summary>
261    /// <para>
262    /// Gets the left size using the specified node.
263    /// </para>
264    /// <para></para>
265    /// </summary>
266    /// <param name="node">
267    /// <para>The node.</para>
268    /// <para></para>
269    /// </param>
270    /// <returns>
271    /// <para>The element</para>
272    /// <para></para>
273    /// </returns>
274    [MethodImpl(MethodImplOptions.AggressiveInlining)]
275    protected TElement GetLeftSize(TElement node) => GetSizeOrZero(GetLeftOrDefault(node));
276
277    /// <summary>
278    /// <para>
279    /// Gets the right size using the specified node.
280    /// </para>
281    /// <para></para>
282    /// </summary>
283    /// <param name="node">
284    /// <para>The node.</para>
285    /// <para></para>
286    /// </param>
287    /// <returns>
288    /// <para>The element</para>
289    /// <para></para>
290    /// </returns>
291    [MethodImpl(MethodImplOptions.AggressiveInlining)]
292    protected TElement GetRightSize(TElement node) => GetSizeOrZero(GetRightOrDefault(node));
293
294    /// <summary>
295    /// <para>
296    /// Gets the size or zero using the specified node.
297    /// </para>
298    /// <para></para>
299    /// </summary>
300    /// <param name="node">
301    /// <para>The node.</para>
302    /// <para></para>
303    /// </param>
304    /// <returns>

```

```

305     /// <para>The element</para>
306     /// <para></para>
307     /// </returns>
308     [MethodImpl(MethodImplOptions.AggressiveInlining)]
309     protected TElement GetSizeOrZero(TElement node) => EqualToZero(node) ? Zero :
        ↳ GetSize(node);

310
311     /// <summary>
312     /// <para>
313     /// Fixes the size using the specified node.
314     /// </para>
315     /// <para></para>
316     /// </summary>
317     /// <param name="node">
318     /// <para>The node.</para>
319     /// <para></para>
320     /// </param>
321     [MethodImpl(MethodImplOptions.AggressiveInlining)]
322     protected void FixSize(TElement node) => SetSize(node, Increment(Add(GetLeftSize(node),
        ↳ GetRightSize(node))));

323
324     /// <summary>
325     /// <para>
326     /// Lefts the rotate using the specified root.
327     /// </para>
328     /// <para></para>
329     /// </summary>
330     /// <param name="root">
331     /// <para>The root.</para>
332     /// <para></para>
333     /// </param>
334     [MethodImpl(MethodImplOptions.AggressiveInlining)]
335     protected void LeftRotate(ref TElement root) => root = LeftRotate(root);

336
337     /// <summary>
338     /// <para>
339     /// Lefts the rotate using the specified root.
340     /// </para>
341     /// <para></para>
342     /// </summary>
343     /// <param name="root">
344     /// <para>The root.</para>
345     /// <para></para>
346     /// </param>
347     /// <returns>
348     /// <para>The right.</para>
349     /// <para></para>
350     /// </returns>
351     [MethodImpl(MethodImplOptions.AggressiveInlining)]
352     protected TElement LeftRotate(TElement root)
353     {
354         var right = GetRight(root);
355         #if ENABLE_TREE_AUTO_DEBUG_AND_VALIDATION
356             if (EqualToZero(right))
357             {
358                 throw new InvalidOperationException("Right is null.");
359             }
360         #endif
361         SetRight(root, GetLeft(right));
362         SetLeft(right, root);
363         SetSize(right, GetSize(root));
364         FixSize(root);
365         return right;
366     }

367
368     /// <summary>
369     /// <para>
370     /// Rights the rotate using the specified root.
371     /// </para>
372     /// <para></para>
373     /// </summary>
374     /// <param name="root">
375     /// <para>The root.</para>
376     /// <para></para>
377     /// </param>
378     [MethodImpl(MethodImplOptions.AggressiveInlining)]
379     protected void RightRotate(ref TElement root) => root = RightRotate(root);
380

```

```

381     /// <summary>
382     /// <para>
383     /// Rights the rotate using the specified root.
384     /// </para>
385     /// <para></para>
386     /// </summary>
387     /// <param name="root">
388     /// <para>The root.</para>
389     /// <para></para>
390     /// </param>
391     /// <returns>
392     /// <para>The left.</para>
393     /// <para></para>
394     /// </returns>
395     [MethodImpl(MethodImplOptions.AggressiveInlining)]
396     protected TElement RightRotate(TElement root)
397     {
398         var left = GetLeft(root);
399     #if ENABLE_TREE_AUTO_DEBUG_AND_VALIDATION
400         if (EqualToZero(left))
401         {
402             throw new InvalidOperationException("Left is null.");
403         }
404     #endif
405         SetLeft(root, GetRight(left));
406         SetRight(left, root);
407         SetSize(left, GetSize(root));
408         FixSize(root);
409         return left;
410     }
411
412     /// <summary>
413     /// <para>
414     /// Gets the rightest using the specified current.
415     /// </para>
416     /// <para></para>
417     /// </summary>
418     /// <param name="current">
419     /// <para>The current.</para>
420     /// <para></para>
421     /// </param>
422     /// <returns>
423     /// <para>The current.</para>
424     /// <para></para>
425     /// </returns>
426     [MethodImpl(MethodImplOptions.AggressiveInlining)]
427     protected virtual TElement GetRightest(TElement current)
428     {
429         var currentRight = GetRight(current);
430         while (!EqualToZero(currentRight))
431         {
432             current = currentRight;
433             currentRight = GetRight(current);
434         }
435         return current;
436     }
437
438     /// <summary>
439     /// <para>
440     /// Gets the leftest using the specified current.
441     /// </para>
442     /// <para></para>
443     /// </summary>
444     /// <param name="current">
445     /// <para>The current.</para>
446     /// <para></para>
447     /// </param>
448     /// <returns>
449     /// <para>The current.</para>
450     /// <para></para>
451     /// </returns>
452     [MethodImpl(MethodImplOptions.AggressiveInlining)]
453     protected virtual TElement GetLeftest(TElement current)
454     {
455         var currentLeft = GetLeft(current);
456         while (!EqualToZero(currentLeft))
457         {
458             current = currentLeft;

```



```

459         currentLeft = GetLeft(current);
460     }
461     return current;
462 }
463
464 /// <summary>
465 /// <para>
466 /// Gets the next using the specified node.
467 /// </para>
468 /// <para></para>
469 /// </summary>
470 /// <param name="node">
471 /// <para>The node.</para>
472 /// <para></para>
473 /// </param>
474 /// <returns>
475 /// <para>The element</para>
476 /// <para></para>
477 /// </returns>
478 [MethodImpl(MethodImplOptions.AggressiveInlining)]
479 protected virtual TElement GetNext(TElement node) => GetLefttest(GetRight(node));
480
481 /// <summary>
482 /// <para>
483 /// Gets the previous using the specified node.
484 /// </para>
485 /// <para></para>
486 /// </summary>
487 /// <param name="node">
488 /// <para>The node.</para>
489 /// <para></para>
490 /// </param>
491 /// <returns>
492 /// <para>The element</para>
493 /// <para></para>
494 /// </returns>
495 [MethodImpl(MethodImplOptions.AggressiveInlining)]
496 protected virtual TElement GetPrevious(TElement node) => GetRighttest(GetLeft(node));
497
498 /// <summary>
499 /// <para>
500 /// Determines whether this instance contains.
501 /// </para>
502 /// <para></para>
503 /// </summary>
504 /// <param name="node">
505 /// <para>The node.</para>
506 /// <para></para>
507 /// </param>
508 /// <param name="root">
509 /// <para>The root.</para>
510 /// <para></para>
511 /// </param>
512 /// <returns>
513 /// <para>The bool</para>
514 /// <para></para>
515 /// </returns>
516 [MethodImpl(MethodImplOptions.AggressiveInlining)]
517 public virtual bool Contains(TElement node, TElement root)
518 {
519     while (!EqualToZero(root))
520     {
521         if (FirstIsToTheLeftOfSecond(node, root)) // node.Key < root.Key
522         {
523             root = GetLeft(root);
524         }
525         else if (FirstIsToTheRightOfSecond(node, root)) // node.Key > root.Key
526         {
527             root = GetRight(root);
528         }
529         else // node.Key == root.Key
530         {
531             return true;
532         }
533     }
534     return false;
535 }
536

```

```

537     /// <summary>
538     /// <para>
539     /// Clears the node using the specified node.
540     /// </para>
541     /// <para></para>
542     /// </summary>
543     /// <param name="node">
544     /// <para>The node.</para>
545     /// <para></para>
546     /// </param>
547     [MethodImpl(MethodImplOptions.AggressiveInlining)]
548     protected virtual void ClearNode(TElement node)
549     {
550         SetLeft(node, Zero);
551         SetRight(node, Zero);
552         SetSize(node, Zero);
553     }
554
555     /// <summary>
556     /// <para>
557     /// Attaches the root.
558     /// </para>
559     /// <para></para>
560     /// </summary>
561     /// <param name="root">
562     /// <para>The root.</para>
563     /// <para></para>
564     /// </param>
565     /// <param name="node">
566     /// <para>The node.</para>
567     /// <para></para>
568     /// </param>
569     [MethodImpl(MethodImplOptions.AggressiveInlining)]
570     public void Attach(ref TElement root, TElement node)
571     {
572         #if ENABLE_TREE_AUTO_DEBUG_AND_VALIDATION
573             ValidateSizes(root);
574             Debug.WriteLine("--BeforeAttach--");
575             Debug.WriteLine(PrintNodes(root));
576             Debug.WriteLine("-----");
577             var sizeBefore = GetSize(root);
578         #endif
579         if (EqualToZero(root))
580         {
581             SetSize(node, One);
582             root = node;
583             return;
584         }
585         AttachCore(ref root, node);
586         #if ENABLE_TREE_AUTO_DEBUG_AND_VALIDATION
587             Debug.WriteLine("--AfterAttach--");
588             Debug.WriteLine(PrintNodes(root));
589             Debug.WriteLine("-----");
590             ValidateSizes(root);
591             var sizeAfter = GetSize(root);
592             if (!AreEqual(Arithmetic.Increment(sizeBefore), sizeAfter))
593             {
594                 throw new InvalidOperationException("Tree was broken after attach.");
595             }
596         #endif
597     }
598
599     /// <summary>
600     /// <para>
601     /// Attaches the core using the specified root.
602     /// </para>
603     /// <para></para>
604     /// </summary>
605     /// <param name="root">
606     /// <para>The root.</para>
607     /// <para></para>
608     /// </param>
609     /// <param name="node">
610     /// <para>The node.</para>
611     /// <para></para>
612     /// </param>
613     protected abstract void AttachCore(ref TElement root, TElement node);
614

```

```

615     /// <summary>
616     /// <para>
617     /// Detaches the root.
618     /// </para>
619     /// <para></para>
620     /// </summary>
621     /// <param name="root">
622     /// <para>The root.</para>
623     /// <para></para>
624     /// </param>
625     /// <param name="node">
626     /// <para>The node.</para>
627     /// <para></para>
628     /// </param>
629     [MethodImpl(MethodImplOptions.AggressiveInlining)]
630     public void Detach(ref TElement root, TElement node)
631     {
632         #if ENABLE_TREE_AUTO_DEBUG_AND_VALIDATION
633             ValidateSizes(root);
634             Debug.WriteLine("--BeforeDetach--");
635             Debug.WriteLine(PrintNodes(root));
636             Debug.WriteLine("-----");
637             var sizeBefore = GetSize(root);
638             if (EqualToZero(root))
639             {
640                 throw new InvalidOperationException($"Элемент с {node} не содержится в
641                 ↪ дереве.");
642             }
643             #endif
644             DetachCore(ref root, node);
645             #if ENABLE_TREE_AUTO_DEBUG_AND_VALIDATION
646             Debug.WriteLine("--AfterDetach--");
647             Debug.WriteLine(PrintNodes(root));
648             Debug.WriteLine("-----");
649             ValidateSizes(root);
650             var sizeAfter = GetSize(root);
651             if (!AreEqual(Arithmetic.Decrement(sizeBefore), sizeAfter))
652             {
653                 throw new InvalidOperationException("Tree was broken after detach.");
654             }
655             #endif
656         }
657         /// <summary>
658         /// <para>
659         /// Detaches the core using the specified root.
660         /// </para>
661         /// <para></para>
662         /// </summary>
663         /// <param name="root">
664         /// <para>The root.</para>
665         /// <para></para>
666         /// </param>
667         /// <param name="node">
668         /// <para>The node.</para>
669         /// <para></para>
670         /// </param>
671         protected abstract void DetachCore(ref TElement root, TElement node);
672
673         /// <summary>
674         /// <para>
675         /// Fixes the sizes using the specified node.
676         /// </para>
677         /// <para></para>
678         /// </summary>
679         /// <param name="node">
680         /// <para>The node.</para>
681         /// <para></para>
682         /// </param>
683         public void FixSizes(TElement node)
684         {
685             if (AreEqual(node, default))
686             {
687                 return;
688             }
689             FixSizes(GetLeft(node));
690             FixSizes(GetRight(node));
691             FixSize(node);

```

```

692 }
693
694 /// <summary>
695 /// <para>
696 /// Validates the sizes using the specified node.
697 /// </para>
698 /// <para></para>
699 /// </summary>
700 /// <param name="node">
701 /// <para>The node.</para>
702 /// <para></para>
703 /// </param>
704 /// <exception cref="InvalidOperationException">
705 /// <para>Size of {node} is not valid. Expected size: {expectedSize}, actual size:
    ↪ {size}.</para>
706 /// <para></para>
707 /// </exception>
708 public void ValidateSizes(TElement node)
709 {
710     if (AreEqual(node, default))
711     {
712         return;
713     }
714     var size = GetSize(node);
715     var leftSize = GetLeftSize(node);
716     var rightSize = GetRightSize(node);
717     var expectedSize = Arithmetic.Increment(Arithmetic.Add(leftSize, rightSize));
718     if (!AreEqual(size, expectedSize))
719     {
720         throw new InvalidOperationException($"Size of {node} is not valid. Expected
    ↪ size: {expectedSize}, actual size: {size}.");
721     }
722     ValidateSizes(GetLeft(node));
723     ValidateSizes(GetRight(node));
724 }
725
726 /// <summary>
727 /// <para>
728 /// Validates the size using the specified node.
729 /// </para>
730 /// <para></para>
731 /// </summary>
732 /// <param name="node">
733 /// <para>The node.</para>
734 /// <para></para>
735 /// </param>
736 /// <exception cref="InvalidOperationException">
737 /// <para>Size of {node} is not valid. Expected size: {expectedSize}, actual size:
    ↪ {size}.</para>
738 /// <para></para>
739 /// </exception>
740 public void ValidateSize(TElement node)
741 {
742     var size = GetSize(node);
743     var leftSize = GetLeftSize(node);
744     var rightSize = GetRightSize(node);
745     var expectedSize = Arithmetic.Increment(Arithmetic.Add(leftSize, rightSize));
746     if (!AreEqual(size, expectedSize))
747     {
748         throw new InvalidOperationException($"Size of {node} is not valid. Expected
    ↪ size: {expectedSize}, actual size: {size}.");
749     }
750 }
751
752 /// <summary>
753 /// <para>
754 /// Prints the nodes using the specified node.
755 /// </para>
756 /// <para></para>
757 /// </summary>
758 /// <param name="node">
759 /// <para>The node.</para>
760 /// <para></para>
761 /// </param>
762 /// <returns>
763 /// <para>The string</para>
764 /// <para></para>

```

```

765     /// </returns>
766     public string PrintNodes(TElement node)
767     {
768         var sb = new StringBuilder();
769         PrintNodes(node, sb);
770         return sb.ToString();
771     }
772
773     /// <summary>
774     /// <para>
775     /// Prints the nodes using the specified node.
776     /// </para>
777     /// <para></para>
778     /// </summary>
779     /// <param name="node">
780     /// <para>The node.</para>
781     /// <para></para>
782     /// </param>
783     /// <param name="sb">
784     /// <para>The sb.</para>
785     /// <para></para>
786     /// </param>
787     [MethodImpl(MethodImplOptions.AggressiveInlining)]
788     public void PrintNodes(TElement node, StringBuilder sb) => PrintNodes(node, sb, 0);
789
790     /// <summary>
791     /// <para>
792     /// Prints the nodes using the specified node.
793     /// </para>
794     /// <para></para>
795     /// </summary>
796     /// <param name="node">
797     /// <para>The node.</para>
798     /// <para></para>
799     /// </param>
800     /// <param name="sb">
801     /// <para>The sb.</para>
802     /// <para></para>
803     /// </param>
804     /// <param name="level">
805     /// <para>The level.</para>
806     /// <para></para>
807     /// </param>
808     public void PrintNodes(TElement node, StringBuilder sb, int level)
809     {
810         if (AreEqual(node, default))
811         {
812             return;
813         }
814         PrintNodes(GetLeft(node), sb, level + 1);
815         PrintNode(node, sb, level);
816         sb.AppendLine();
817         PrintNodes(GetRight(node), sb, level + 1);
818     }
819
820     /// <summary>
821     /// <para>
822     /// Prints the node using the specified node.
823     /// </para>
824     /// <para></para>
825     /// </summary>
826     /// <param name="node">
827     /// <para>The node.</para>
828     /// <para></para>
829     /// </param>
830     /// <returns>
831     /// <para>The string</para>
832     /// <para></para>
833     /// </returns>
834     public string PrintNode(TElement node)
835     {
836         var sb = new StringBuilder();
837         PrintNode(node, sb);
838         return sb.ToString();
839     }
840
841     /// <summary>
842     /// <para>

```

```

843     /// Prints the node using the specified node.
844     /// </para>
845     /// <para></para>
846     /// </summary>
847     /// <param name="node">
848     /// <para>The node.</para>
849     /// <para></para>
850     /// </param>
851     /// <param name="sb">
852     /// <para>The sb.</para>
853     /// <para></para>
854     /// </param>
855     [MethodImpl(MethodImplOptions.AggressiveInlining)]
856     protected void PrintNode(TElement node, StringBuilder sb) => PrintNode(node, sb, 0);
857
858     /// <summary>
859     /// <para>
860     /// Prints the node using the specified node.
861     /// </para>
862     /// <para></para>
863     /// </summary>
864     /// <param name="node">
865     /// <para>The node.</para>
866     /// <para></para>
867     /// </param>
868     /// <param name="sb">
869     /// <para>The sb.</para>
870     /// <para></para>
871     /// </param>
872     /// <param name="level">
873     /// <para>The level.</para>
874     /// <para></para>
875     /// </param>
876     protected virtual void PrintNode(TElement node, StringBuilder sb, int level)
877     {
878         sb.Append('\t', level);
879         sb.Append(node);
880         PrintNodeValue(node, sb);
881         sb.Append(' ');
882         sb.Append('s');
883         sb.Append(GetSize(node));
884     }
885
886     /// <summary>
887     /// <para>
888     /// Prints the node value using the specified node.
889     /// </para>
890     /// <para></para>
891     /// </summary>
892     /// <param name="node">
893     /// <para>The node.</para>
894     /// <para></para>
895     /// </param>
896     /// <param name="sb">
897     /// <para>The sb.</para>
898     /// <para></para>
899     /// </param>
900     protected abstract void PrintNodeValue(TElement node, StringBuilder sb);
901 }
902 }

```

1.13 ./csharp/Platform.Collections.Methods.Tests/RecursionlessSizeBalancedTree.cs

```

1  using System;
2  using System.Collections.Generic;
3  using System.Text;
4  using Platform.Numbers;
5  using Platform.Collections.Methods.Trees;
6  using Platform.Converters;
7
8  namespace Platform.Collections.Methods.Tests
9  {
10     /// <summary>
11     /// <para>
12     /// Represents the recursionless size balanced tree.
13     /// </para>
14     /// <para></para>
15     /// </summary>
16     /// <seealso cref="RecursionlessSizeBalancedTreeMethods{TElement}"/>

```

```

17 public class RecursionlessSizeBalancedTree<TElement> :
    ↳ RecursionlessSizeBalancedTreeMethods<TElement>
18 {
19     /// <summary>
20     /// <para>
21     /// The tree element.
22     /// </para>
23     /// <para></para>
24     /// </summary>
25     private struct TreeElement
26     {
27         /// <summary>
28         /// <para>
29         /// The size.
30         /// </para>
31         /// <para></para>
32         /// </summary>
33         public TElement Size;
34         /// <summary>
35         /// <para>
36         /// The left.
37         /// </para>
38         /// <para></para>
39         /// </summary>
40         public TreeElement Left;
41         /// <summary>
42         /// <para>
43         /// The right.
44         /// </para>
45         /// <para></para>
46         /// </summary>
47         public TreeElement Right;
48     }
49
50     /// <summary>
51     /// <para>
52     /// The elements.
53     /// </para>
54     /// <para></para>
55     /// </summary>
56     private readonly TreeElement[] _elements;
57     /// <summary>
58     /// <para>
59     /// The allocated.
60     /// </para>
61     /// <para></para>
62     /// </summary>
63     private TElement _allocated;
64
65     /// <summary>
66     /// <para>
67     /// The root.
68     /// </para>
69     /// <para></para>
70     /// </summary>
71     public TElement Root;
72
73     /// <summary>
74     /// <para>
75     /// Gets the count value.
76     /// </para>
77     /// <para></para>
78     /// </summary>
79     public TElement Count => GetSizeOrZero(Root);
80
81     /// <summary>
82     /// <para>
83     /// Initializes a new <see cref="RecursionlessSizeBalancedTree"/> instance.
84     /// </para>
85     /// <para></para>
86     /// </summary>
87     /// <param name="capacity">
88     /// <para>A capacity.</para>
89     /// <para></para>
90     /// </param>
91     public RecursionlessSizeBalancedTree(int capacity) => (_elements, _allocated) = (new
    ↳ TreeElement[capacity], One);
92

```

```

93     /// <summary>
94     /// <para>
95     /// Allocates this instance.
96     /// </para>
97     /// <para></para>
98     /// </summary>
99     /// <exception cref="InvalidOperationException">
100    /// <para>Allocated tree element is not empty.</para>
101    /// <para></para>
102    /// </exception>
103    /// <returns>
104    /// <para>The element</para>
105    /// <para></para>
106    /// </returns>
107    public TElement Allocate()
108    {
109        var newNode = _allocated;
110        if (IsEmpty(newNode))
111        {
112            _allocated = Arithmetic.Increment(_allocated);
113            return newNode;
114        }
115        else
116        {
117            throw new InvalidOperationException("Allocated tree element is not empty.");
118        }
119    }
120
121    /// <summary>
122    /// <para>
123    /// Frees the node.
124    /// </para>
125    /// <para></para>
126    /// </summary>
127    /// <param name="node">
128    /// <para>The node.</para>
129    /// <para></para>
130    /// </param>
131    public void Free(TElement node)
132    {
133        while (!EqualityComparer.Equals(_allocated, One) && IsEmpty(node))
134        {
135            var lastNode = Arithmetic.Decrement(_allocated);
136            if (EqualityComparer.Equals(lastNode, node))
137            {
138                _allocated = lastNode;
139                node = Arithmetic.Decrement(node);
140            }
141            else
142            {
143                return;
144            }
145        }
146    }
147
148    /// <summary>
149    /// <para>
150    /// Determines whether this instance is empty.
151    /// </para>
152    /// <para></para>
153    /// </summary>
154    /// <param name="node">
155    /// <para>The node.</para>
156    /// <para></para>
157    /// </param>
158    /// <returns>
159    /// <para>The bool</para>
160    /// <para></para>
161    /// </returns>
162    public bool IsEmpty(TElement node) =>
163        ↪ EqualityComparer<TreeElement>.Default.Equals(GetElement(node), default);
164
165    /// <summary>
166    /// <para>
167    /// Determines whether this instance first is to the left of second.
168    /// </para>
169    /// <para></para>
170    /// </summary>

```



```

170    /// <param name="first">
171    /// <para>The first.</para>
172    /// <para></para>
173    /// </param>
174    /// <param name="second">
175    /// <para>The second.</para>
176    /// <para></para>
177    /// </param>
178    /// <returns>
179    /// <para>The bool</para>
180    /// <para></para>
181    /// </returns>
182    protected override bool FirstIsToTheLeftOfSecond(TElement first, TElement second) =>
183        ↪ Comparer.Compare(first, second) < 0;
184
185    /// <summary>
186    /// <para>
187    /// Determines whether this instance first is to the right of second.
188    /// </para>
189    /// <para></para>
190    /// </summary>
191    /// <param name="first">
192    /// <para>The first.</para>
193    /// <para></para>
194    /// </param>
195    /// <param name="second">
196    /// <para>The second.</para>
197    /// <para></para>
198    /// </param>
199    /// <returns>
200    /// <para>The bool</para>
201    /// <para></para>
202    /// </returns>
203    protected override bool FirstIsToTheRightOfSecond(TElement first, TElement second) =>
204        ↪ Comparer.Compare(first, second) > 0;
205
206    /// <summary>
207    /// <para>
208    /// Gets the left reference using the specified node.
209    /// </para>
210    /// <para></para>
211    /// </summary>
212    /// <param name="node">
213    /// <para>The node.</para>
214    /// <para></para>
215    /// </param>
216    /// <returns>
217    /// <para>The ref element</para>
218    /// <para></para>
219    /// </returns>
220    protected override ref TElement GetLeftReference(TElement node) => ref
221        ↪ GetElement(node).Left;
222
223    /// <summary>
224    /// <para>
225    /// Gets the left using the specified node.
226    /// </para>
227    /// <para></para>
228    /// </summary>
229    /// <param name="node">
230    /// <para>The node.</para>
231    /// <para></para>
232    /// </param>
233    /// <returns>
234    /// <para>The element</para>
235    /// <para></para>
236    /// </returns>
237    protected override TElement GetLeft(TElement node) => GetElement(node).Left;
238
239    /// <summary>
240    /// <para>
241    /// Gets the right reference using the specified node.
242    /// </para>
243    /// <para></para>
244    /// </summary>
245    /// <param name="node">
246    /// <para>The node.</para>
247    /// <para></para>
248    /// </param>
249    /// <returns>
250    /// <para>The element</para>
251    /// <para></para>
252    /// </returns>
253    protected override ref TElement GetRightReference(TElement node) => ref
254        ↪ GetElement(node).Right;
255
256    /// <summary>
257    /// <para>
258    /// Gets the right using the specified node.
259    /// </para>
260    /// <para></para>
261    /// </summary>
262    /// <param name="node">
263    /// <para>The node.</para>
264    /// <para></para>
265    /// </param>
266    /// <returns>
267    /// <para>The element</para>
268    /// <para></para>
269    /// </returns>
270    protected override TElement GetRight(TElement node) => GetElement(node).Right;

```

```

245    /// </param>
246    /// <returns>
247    /// <para>The ref element</para>
248    /// <para></para>
249    /// </returns>
250    protected override ref TElement GetRightReference(TElement node) => ref
    ↪ GetElement(node).Right;

251
252    /// <summary>
253    /// <para>
254    /// Gets the right using the specified node.
255    /// </para>
256    /// <para></para>
257    /// </summary>
258    /// <param name="node">
259    /// <para>The node.</para>
260    /// <para></para>
261    /// </param>
262    /// <returns>
263    /// <para>The element</para>
264    /// <para></para>
265    /// </returns>
266    protected override TElement GetRight(TElement node) => GetElement(node).Right;
267
268    /// <summary>
269    /// <para>
270    /// Gets the size using the specified node.
271    /// </para>
272    /// <para></para>
273    /// </summary>
274    /// <param name="node">
275    /// <para>The node.</para>
276    /// <para></para>
277    /// </param>
278    /// <returns>
279    /// <para>The element</para>
280    /// <para></para>
281    /// </returns>
282    protected override TElement GetSize(TElement node) => GetElement(node).Size;
283
284    /// <summary>
285    /// <para>
286    /// Prints the node value using the specified node.
287    /// </para>
288    /// <para></para>
289    /// </summary>
290    /// <param name="node">
291    /// <para>The node.</para>
292    /// <para></para>
293    /// </param>
294    /// <param name="sb">
295    /// <para>The sb.</para>
296    /// <para></para>
297    /// </param>
298    protected override void PrintNodeValue(TElement node, StringBuilder sb) =>
    ↪ sb.Append(node);

299
300    /// <summary>
301    /// <para>
302    /// Sets the left using the specified node.
303    /// </para>
304    /// <para></para>
305    /// </summary>
306    /// <param name="node">
307    /// <para>The node.</para>
308    /// <para></para>
309    /// </param>
310    /// <param name="left">
311    /// <para>The left.</para>
312    /// <para></para>
313    /// </param>
314    protected override void SetLeft(TElement node, TElement left) => GetElement(node).Left =
    ↪ left;

315
316    /// <summary>
317    /// <para>
318    /// Sets the right using the specified node.
319    /// </para>

```

```

320     /// <para></para>
321     /// </summary>
322     /// <param name="node">
323     /// <para>The node.</para>
324     /// <para></para>
325     /// </param>
326     /// <param name="right">
327     /// <para>The right.</para>
328     /// <para></para>
329     /// </param>
330     protected override void SetRight(TElement node, TElement right) =>
        ↪ GetElement(node).Right = right;
331
332     /// <summary>
333     /// <para>
334     /// Sets the size using the specified node.
335     /// </para>
336     /// <para></para>
337     /// </summary>
338     /// <param name="node">
339     /// <para>The node.</para>
340     /// <para></para>
341     /// </param>
342     /// <param name="size">
343     /// <para>The size.</para>
344     /// <para></para>
345     /// </param>
346     protected override void SetSize(TElement node, TElement size) => GetElement(node).Size =
        ↪ size;
347
348     /// <summary>
349     /// <para>
350     /// Gets the element using the specified node.
351     /// </para>
352     /// <para></para>
353     /// </summary>
354     /// <param name="node">
355     /// <para>The node.</para>
356     /// <para></para>
357     /// </param>
358     /// <returns>
359     /// <para>The ref tree element</para>
360     /// <para></para>
361     /// </returns>
362     private ref TreeElement GetElement(TElement node) => ref
        ↪ _elements[UncheckedConverter<TElement, long>.Default.Convert(node)];
363 }
364 }

```

1.14 ./csharp/Platform.Collections.Methods.Tests/SizeBalancedTree.cs

```

1  using System;
2  using System.Collections.Generic;
3  using System.Text;
4  using Platform.Numbers;
5  using Platform.Collections.Methods.Trees;
6  using Platform.Converters;
7
8  namespace Platform.Collections.Methods.Tests
9  {
10     /// <summary>
11     /// <para>
12     /// Represents the size balanced tree.
13     /// </para>
14     /// <para></para>
15     /// </summary>
16     /// <seealso cref="SizeBalancedTreeMethods{TElement}" />
17     public class SizeBalancedTree<TElement> : SizeBalancedTreeMethods<TElement>
18     {
19         /// <summary>
20         /// <para>
21         /// The tree element.
22         /// </para>
23         /// <para></para>
24         /// </summary>
25         private struct TreeElement
26         {
27             /// <summary>
28             /// <para>

```

```

29     /// The size.
30     /// </para>
31     /// <para></para>
32     /// </summary>
33     public TElement Size;
34     /// <summary>
35     /// <para>
36     /// The left.
37     /// </para>
38     /// <para></para>
39     /// </summary>
40     public TElement Left;
41     /// <summary>
42     /// <para>
43     /// The right.
44     /// </para>
45     /// <para></para>
46     /// </summary>
47     public TElement Right;
48 }
49
50     /// <summary>
51     /// <para>
52     /// The elements.
53     /// </para>
54     /// <para></para>
55     /// </summary>
56     private readonly TreeElement[] _elements;
57     /// <summary>
58     /// <para>
59     /// The allocated.
60     /// </para>
61     /// <para></para>
62     /// </summary>
63     private TElement _allocated;
64
65     /// <summary>
66     /// <para>
67     /// The root.
68     /// </para>
69     /// <para></para>
70     /// </summary>
71     public TElement Root;
72
73     /// <summary>
74     /// <para>
75     /// Gets the count value.
76     /// </para>
77     /// <para></para>
78     /// </summary>
79     public TElement Count => GetSizeOrZero(Root);
80
81     /// <summary>
82     /// <para>
83     /// Initializes a new <see cref="SizeBalancedTree"/> instance.
84     /// </para>
85     /// <para></para>
86     /// </summary>
87     /// <param name="capacity">
88     /// <para>A capacity.</para>
89     /// <para></para>
90     /// </param>
91     public SizeBalancedTree(int capacity) => (_elements, _allocated) = (new
    ↪ TreeElement[capacity], One);
92
93     /// <summary>
94     /// <para>
95     /// Allocates this instance.
96     /// </para>
97     /// <para></para>
98     /// </summary>
99     /// <exception cref="InvalidOperationException">
100    /// <para>Allocated tree element is not empty.</para>
101    /// <para></para>
102    /// </exception>
103    /// <returns>
104    /// <para>The element</para>
105    /// <para></para>

```

```

106     /// </returns>
107 public TElement Allocate()
108 {
109     var newNode = _allocated;
110     if (IsEmpty(newNode))
111     {
112         _allocated = Arithmetic.Increment(_allocated);
113         return newNode;
114     }
115     else
116     {
117         throw new InvalidOperationException("Allocated tree element is not empty.");
118     }
119 }
120
121 /// <summary>
122 /// <para>
123 /// Frees the node.
124 /// </para>
125 /// <para></para>
126 /// </summary>
127 /// <param name="node">
128 /// <para>The node.</para>
129 /// <para></para>
130 /// </param>
131 public void Free(TElement node)
132 {
133     while (!EqualityComparer.Equals(_allocated, One) && IsEmpty(node))
134     {
135         var lastNode = Arithmetic.Decrement(_allocated);
136         if (EqualityComparer.Equals(lastNode, node))
137         {
138             _allocated = lastNode;
139             node = Arithmetic.Decrement(node);
140         }
141         else
142         {
143             return;
144         }
145     }
146 }
147
148 /// <summary>
149 /// <para>
150 /// Determines whether this instance is empty.
151 /// </para>
152 /// <para></para>
153 /// </summary>
154 /// <param name="node">
155 /// <para>The node.</para>
156 /// <para></para>
157 /// </param>
158 /// <returns>
159 /// <para>The bool</para>
160 /// <para></para>
161 /// </returns>
162 public bool IsEmpty(TElement node) =>
163     ↪ EqualityComparer<TreeElement>.Default.Equals(GetElement(node), default);
164
165 /// <summary>
166 /// <para>
167 /// Determines whether this instance first is to the left of second.
168 /// </para>
169 /// <para></para>
170 /// </summary>
171 /// <param name="first">
172 /// <para>The first.</para>
173 /// <para></para>
174 /// </param>
175 /// <param name="second">
176 /// <para>The second.</para>
177 /// <para></para>
178 /// </param>
179 /// <returns>
180 /// <para>The bool</para>
181 /// <para></para>
182 /// </returns>

```

```

182     protected override bool FirstIsToTheLeftOfSecond(TElement first, TElement second) =>
183         ↳ Comparer.Compare(first, second) < 0;
184
185     /// <summary>
186     /// <para>
187     /// Determines whether this instance first is to the right of second.
188     /// </para>
189     /// </summary>
190     /// <param name="first">
191     /// <para>The first.</para>
192     /// </param>
193     /// <param name="second">
194     /// <para>The second.</para>
195     /// </param>
196     /// <returns>
197     /// <para>The bool</para>
198     /// </returns>
199     protected override bool FirstIsToTheRightOfSecond(TElement first, TElement second) =>
200         ↳ Comparer.Compare(first, second) > 0;
201
202     /// <summary>
203     /// <para>
204     /// Gets the left reference using the specified node.
205     /// </para>
206     /// </summary>
207     /// <param name="node">
208     /// <para>The node.</para>
209     /// </param>
210     /// <returns>
211     /// <para>The ref element</para>
212     /// </returns>
213     protected override ref TElement GetLeftReference(TElement node) => ref
214         ↳ GetElement(node).Left;
215
216     /// <summary>
217     /// <para>
218     /// Gets the left using the specified node.
219     /// </para>
220     /// </summary>
221     /// <param name="node">
222     /// <para>The node.</para>
223     /// </param>
224     /// <returns>
225     /// <para>The element</para>
226     /// </returns>
227     protected override TElement GetLeft(TElement node) => GetElement(node).Left;
228
229     /// <summary>
230     /// <para>
231     /// Gets the right reference using the specified node.
232     /// </para>
233     /// </summary>
234     /// <param name="node">
235     /// <para>The node.</para>
236     /// </param>
237     /// <returns>
238     /// <para>The ref element</para>
239     /// </returns>
240     protected override ref TElement GetRightReference(TElement node) => ref
241         ↳ GetElement(node).Right;
242
243     /// <summary>
244     /// <para>
245     /// Gets the right using the specified node.
246     /// </para>
247     /// </summary>

```

```

256     /// <para></para>
257     /// </summary>
258     /// <param name="node">
259     /// <para>The node.</para>
260     /// <para></para>
261     /// </param>
262     /// <returns>
263     /// <para>The element</para>
264     /// <para></para>
265     /// </returns>
266     protected override TElement GetRight(TElement node) => GetElement(node).Right;
267
268     /// <summary>
269     /// <para>
270     /// Gets the size using the specified node.
271     /// </para>
272     /// <para></para>
273     /// </summary>
274     /// <param name="node">
275     /// <para>The node.</para>
276     /// <para></para>
277     /// </param>
278     /// <returns>
279     /// <para>The element</para>
280     /// <para></para>
281     /// </returns>
282     protected override TElement GetSize(TElement node) => GetElement(node).Size;
283
284     /// <summary>
285     /// <para>
286     /// Prints the node value using the specified node.
287     /// </para>
288     /// <para></para>
289     /// </summary>
290     /// <param name="node">
291     /// <para>The node.</para>
292     /// <para></para>
293     /// </param>
294     /// <param name="sb">
295     /// <para>The sb.</para>
296     /// <para></para>
297     /// </param>
298     protected override void PrintNodeValue(TElement node, StringBuilder sb) =>
299         ↪ sb.Append(node);
300
301     /// <summary>
302     /// <para>
303     /// Sets the left using the specified node.
304     /// </para>
305     /// <para></para>
306     /// </summary>
307     /// <param name="node">
308     /// <para>The node.</para>
309     /// <para></para>
310     /// </param>
311     /// <param name="left">
312     /// <para>The left.</para>
313     /// <para></para>
314     /// </param>
315     protected override void SetLeft(TElement node, TElement left) => GetElement(node).Left =
316         ↪ left;
317
318     /// <summary>
319     /// <para>
320     /// Sets the right using the specified node.
321     /// </para>
322     /// <para></para>
323     /// </summary>
324     /// <param name="node">
325     /// <para>The node.</para>
326     /// <para></para>
327     /// </param>
328     /// <param name="right">
329     /// <para>The right.</para>
330     /// <para></para>
331     /// </param>
332     protected override void SetRight(TElement node, TElement right) =>
333         ↪ GetElement(node).Right = right;

```

```

331
332     /// <summary>
333     /// <para>
334     /// Sets the size using the specified node.
335     /// </para>
336     /// <para></para>
337     /// </summary>
338     /// <param name="node">
339     /// <para>The node.</para>
340     /// <para></para>
341     /// </param>
342     /// <param name="size">
343     /// <para>The size.</para>
344     /// <para></para>
345     /// </param>
346     protected override void SetSize(TElement node, TElement size) => GetElement(node).Size =
        ↪ size;
347
348     /// <summary>
349     /// <para>
350     /// Gets the element using the specified node.
351     /// </para>
352     /// <para></para>
353     /// </summary>
354     /// <param name="node">
355     /// <para>The node.</para>
356     /// <para></para>
357     /// </param>
358     /// <returns>
359     /// <para>The ref tree element</para>
360     /// <para></para>
361     /// </returns>
362     private ref TreeElement GetElement(TElement node) => ref
        ↪ _elements[UncheckedConverter<TElement, long>.Default.Convert(node)];
363 }
364 }

```

1.15 ./csharp/Platform.Collections.Methods.Tests/SizedAndThreadedAVLBalancedTree.cs

```

1  using System;
2  using System.Collections.Generic;
3  using System.Text;
4  using Platform.Numbers;
5  using Platform.Collections.Methods.Trees;
6  using Platform.Converters;
7
8  namespace Platform.Collections.Methods.Tests
9  {
10     /// <summary>
11     /// <para>
12     /// Represents the sized and threaded avl balanced tree.
13     /// </para>
14     /// <para></para>
15     /// </summary>
16     /// <seealso cref="SizedAndThreadedAVLBalancedTreeMethods{TElement}"/>
17     public class SizedAndThreadedAVLBalancedTree<TElement> :
        ↪ SizedAndThreadedAVLBalancedTreeMethods<TElement>
18     {
19         /// <summary>
20         /// <para>
21         /// The tree element.
22         /// </para>
23         /// <para></para>
24         /// </summary>
25         private struct TreeElement
26         {
27             /// <summary>
28             /// <para>
29             /// The size.
30             /// </para>
31             /// <para></para>
32             /// </summary>
33             public TElement Size;
34             /// <summary>
35             /// <para>
36             /// The left.
37             /// </para>
38             /// <para></para>
39             /// </summary>
40             public TElement Left;

```



```

41     /// <summary>
42     /// <para>
43     /// The right.
44     /// </para>
45     /// <para></para>
46     /// </summary>
47     public TElement Right;
48     /// <summary>
49     /// <para>
50     /// The balance.
51     /// </para>
52     /// <para></para>
53     /// </summary>
54     public sbyte Balance;
55     /// <summary>
56     /// <para>
57     /// The left is child.
58     /// </para>
59     /// <para></para>
60     /// </summary>
61     public bool LeftIsChild;
62     /// <summary>
63     /// <para>
64     /// The right is child.
65     /// </para>
66     /// <para></para>
67     /// </summary>
68     public bool RightIsChild;
69 }
70
71     /// <summary>
72     /// <para>
73     /// The elements.
74     /// </para>
75     /// <para></para>
76     /// </summary>
77     private readonly TreeElement[] _elements;
78     /// <summary>
79     /// <para>
80     /// The allocated.
81     /// </para>
82     /// <para></para>
83     /// </summary>
84     private TElement _allocated;
85
86     /// <summary>
87     /// <para>
88     /// The root.
89     /// </para>
90     /// <para></para>
91     /// </summary>
92     public TElement Root;
93
94     /// <summary>
95     /// <para>
96     /// Gets the count value.
97     /// </para>
98     /// <para></para>
99     /// </summary>
100    public TElement Count => GetSizeOrZero(Root);
101
102    /// <summary>
103    /// <para>
104    /// Initializes a new <see cref="SizedAndThreadedAVLBalancedTree"/> instance.
105    /// </para>
106    /// <para></para>
107    /// </summary>
108    /// <param name="capacity">
109    /// <para>A capacity.</para>
110    /// <para></para>
111    /// </param>
112    public SizedAndThreadedAVLBalancedTree(int capacity) => (_elements, _allocated) = (new
    ↪ TreeElement[capacity], One);
113
114    /// <summary>
115    /// <para>
116    /// Allocates this instance.
117    /// </para>

```

```

118     /// <para></para>
119     /// </summary>
120     /// <exception cref="InvalidOperationException">
121     /// <para>Allocated tree element is not empty.</para>
122     /// <para></para>
123     /// </exception>
124     /// <returns>
125     /// <para>The element</para>
126     /// <para></para>
127     /// </returns>
128     public TElement Allocate()
129     {
130         var newNode = _allocated;
131         if (IsEmpty(newNode))
132         {
133             _allocated = Arithmetic.Increment(_allocated);
134             return newNode;
135         }
136         else
137         {
138             throw new InvalidOperationException("Allocated tree element is not empty.");
139         }
140     }
141
142     /// <summary>
143     /// <para>
144     /// Frees the node.
145     /// </para>
146     /// <para></para>
147     /// </summary>
148     /// <param name="node">
149     /// <para>The node.</para>
150     /// <para></para>
151     /// </param>
152     public void Free(TElement node)
153     {
154         while (!EqualityComparer.Equals(_allocated, One) && IsEmpty(node))
155         {
156             var lastNode = Arithmetic.Decrement(_allocated);
157             if (EqualityComparer.Equals(lastNode, node))
158             {
159                 _allocated = lastNode;
160                 node = Arithmetic.Decrement(node);
161             }
162             else
163             {
164                 return;
165             }
166         }
167     }
168
169     /// <summary>
170     /// <para>
171     /// Determines whether this instance is empty.
172     /// </para>
173     /// <para></para>
174     /// </summary>
175     /// <param name="node">
176     /// <para>The node.</para>
177     /// <para></para>
178     /// </param>
179     /// <returns>
180     /// <para>The bool</para>
181     /// <para></para>
182     /// </returns>
183     public bool IsEmpty(TElement node) =>
184         ⇨ EqualityComparer<TreeElement>.Default.Equals(GetElement(node), default);
185
186     /// <summary>
187     /// <para>
188     /// Determines whether this instance first is to the left of second.
189     /// </para>
190     /// <para></para>
191     /// </summary>
192     /// <param name="first">
193     /// <para>The first.</para>
194     /// <para></para>
195     /// </param>

```

```

195     /// <param name="second">
196     /// <para>The second.</para>
197     /// <para></para>
198     /// </param>
199     /// <returns>
200     /// <para>The bool</para>
201     /// <para></para>
202     /// </returns>
203     protected override bool FirstIsToTheLeftOfSecond(TElement first, TElement second) =>
204         ↪ Comparer.Compare(first, second) < 0;
205
206     /// <summary>
207     /// <para>
208     /// Determines whether this instance first is to the right of second.
209     /// </para>
210     /// <para></para>
211     /// </summary>
212     /// <param name="first">
213     /// <para>The first.</para>
214     /// <para></para>
215     /// </param>
216     /// <param name="second">
217     /// <para>The second.</para>
218     /// <para></para>
219     /// </param>
220     /// <returns>
221     /// <para>The bool</para>
222     /// <para></para>
223     /// </returns>
224     protected override bool FirstIsToTheRightOfSecond(TElement first, TElement second) =>
225         ↪ Comparer.Compare(first, second) > 0;
226
227     /// <summary>
228     /// <para>
229     /// Gets the balance using the specified node.
230     /// </para>
231     /// <para></para>
232     /// </summary>
233     /// <param name="node">
234     /// <para>The node.</para>
235     /// <para></para>
236     /// </param>
237     /// <returns>
238     /// <para>The sbyte</para>
239     /// <para></para>
240     /// </returns>
241     protected override sbyte GetBalance(TElement node) => GetElement(node).Balance;
242
243     /// <summary>
244     /// <para>
245     /// Determines whether this instance get left is child.
246     /// </para>
247     /// <para></para>
248     /// </summary>
249     /// <param name="node">
250     /// <para>The node.</para>
251     /// <para></para>
252     /// </param>
253     /// <returns>
254     /// <para>The bool</para>
255     /// <para></para>
256     /// </returns>
257     protected override bool GetLeftIsChild(TElement node) => GetElement(node).LeftIsChild;
258
259     /// <summary>
260     /// <para>
261     /// Gets the left reference using the specified node.
262     /// </para>
263     /// <para></para>
264     /// </summary>
265     /// <param name="node">
266     /// <para>The node.</para>
267     /// <para></para>
268     /// </param>
269     /// <returns>
270     /// <para>The ref element</para>
271     /// <para></para>
272     /// </returns>

```

```

271 protected override ref TElement GetLeftReference(TElement node) => ref
    ↪ GetElement(node).Left;
272
273 /// <summary>
274 /// <para>
275 /// Gets the left using the specified node.
276 /// </para>
277 /// <para></para>
278 /// </summary>
279 /// <param name="node">
280 /// <para>The node.</para>
281 /// <para></para>
282 /// </param>
283 /// <returns>
284 /// <para>The element</para>
285 /// <para></para>
286 /// </returns>
287 protected override TElement GetLeft(TElement node) => GetElement(node).Left;
288
289 /// <summary>
290 /// <para>
291 /// Determines whether this instance get right is child.
292 /// </para>
293 /// <para></para>
294 /// </summary>
295 /// <param name="node">
296 /// <para>The node.</para>
297 /// <para></para>
298 /// </param>
299 /// <returns>
300 /// <para>The bool</para>
301 /// <para></para>
302 /// </returns>
303 protected override bool GetRightIsChild(TElement node) => GetElement(node).RightIsChild;
304
305 /// <summary>
306 /// <para>
307 /// Gets the right reference using the specified node.
308 /// </para>
309 /// <para></para>
310 /// </summary>
311 /// <param name="node">
312 /// <para>The node.</para>
313 /// <para></para>
314 /// </param>
315 /// <returns>
316 /// <para>The ref element</para>
317 /// <para></para>
318 /// </returns>
319 protected override ref TElement GetRightReference(TElement node) => ref
    ↪ GetElement(node).Right;
320
321 /// <summary>
322 /// <para>
323 /// Gets the right using the specified node.
324 /// </para>
325 /// <para></para>
326 /// </summary>
327 /// <param name="node">
328 /// <para>The node.</para>
329 /// <para></para>
330 /// </param>
331 /// <returns>
332 /// <para>The element</para>
333 /// <para></para>
334 /// </returns>
335 protected override TElement GetRight(TElement node) => GetElement(node).Right;
336
337 /// <summary>
338 /// <para>
339 /// Gets the size using the specified node.
340 /// </para>
341 /// <para></para>
342 /// </summary>
343 /// <param name="node">
344 /// <para>The node.</para>
345 /// <para></para>
346 /// </param>

```

```

347     /// <returns>
348     /// <para>The element</para>
349     /// <para></para>
350     /// </returns>
351     protected override TElement GetSize(TElement node) => GetElement(node).Size;
352
353     /// <summary>
354     /// <para>
355     /// Prints the node value using the specified node.
356     /// </para>
357     /// <para></para>
358     /// </summary>
359     /// <param name="node">
360     /// <para>The node.</para>
361     /// <para></para>
362     /// </param>
363     /// <param name="sb">
364     /// <para>The sb.</para>
365     /// <para></para>
366     /// </param>
367     protected override void PrintNodeValue(TElement node, StringBuilder sb) =>
368         ↪ sb.Append(node);
369
370     /// <summary>
371     /// <para>
372     /// Sets the balance using the specified node.
373     /// </para>
374     /// <para></para>
375     /// </summary>
376     /// <param name="node">
377     /// <para>The node.</para>
378     /// <para></para>
379     /// </param>
380     /// <param name="value">
381     /// <para>The value.</para>
382     /// <para></para>
383     /// </param>
384     protected override void SetBalance(TElement node, sbyte value) =>
385         ↪ GetElement(node).Balance = value;
386
387     /// <summary>
388     /// <para>
389     /// Sets the left using the specified node.
390     /// </para>
391     /// <para></para>
392     /// </summary>
393     /// <param name="node">
394     /// <para>The node.</para>
395     /// <para></para>
396     /// </param>
397     /// <param name="left">
398     /// <para>The left.</para>
399     /// <para></para>
400     /// </param>
401     protected override void SetLeft(TElement node, TElement left) => GetElement(node).Left =
402         ↪ left;
403
404     /// <summary>
405     /// <para>
406     /// Sets the left is child using the specified node.
407     /// </para>
408     /// <para></para>
409     /// </summary>
410     /// <param name="node">
411     /// <para>The node.</para>
412     /// <para></para>
413     /// </param>
414     /// <param name="value">
415     /// <para>The value.</para>
416     /// <para></para>
417     /// </param>
418     protected override void SetLeftIsChild(TElement node, bool value) =>
419         ↪ GetElement(node).LeftIsChild = value;
420
421     /// <summary>
422     /// <para>
423     /// Sets the right using the specified node.
424     /// </para>

```

```

421     /// <para></para>
422     /// </summary>
423     /// <param name="node">
424     /// <para>The node.</para>
425     /// <para></para>
426     /// </param>
427     /// <param name="right">
428     /// <para>The right.</para>
429     /// <para></para>
430     /// </param>
431     protected override void SetRight(TElement node, TElement right) =>
432     ↪ GetElement(node).Right = right;
433
434     /// <summary>
435     /// <para>
436     /// Sets the right is child using the specified node.
437     /// </para>
438     /// <para></para>
439     /// </summary>
440     /// <param name="node">
441     /// <para>The node.</para>
442     /// <para></para>
443     /// </param>
444     /// <param name="value">
445     /// <para>The value.</para>
446     /// <para></para>
447     /// </param>
448     protected override void SetRightIsChild(TElement node, bool value) =>
449     ↪ GetElement(node).RightIsChild = value;
450
451     /// <summary>
452     /// <para>
453     /// Sets the size using the specified node.
454     /// </para>
455     /// <para></para>
456     /// </summary>
457     /// <param name="node">
458     /// <para>The node.</para>
459     /// <para></para>
460     /// </param>
461     /// <param name="size">
462     /// <para>The size.</para>
463     /// <para></para>
464     /// </param>
465     protected override void SetSize(TElement node, TElement size) => GetElement(node).Size =
466     ↪ size;
467
468     /// <summary>
469     /// <para>
470     /// Gets the element using the specified node.
471     /// </para>
472     /// <para></para>
473     /// </summary>
474     /// <param name="node">
475     /// <para>The node.</para>
476     /// <para></para>
477     /// </param>
478     /// <returns>
479     /// <para>The ref tree element</para>
480     /// <para></para>
481     /// </returns>
482     private ref TreeElement GetElement(TElement node) => ref
483     ↪ _elements[UncheckedConverter<TElement, long>.Default.Convert(node)];
484 }
485 }

```

1.16 ./csharp/Platform.Collections.Methods.Tests/TestExtensions.cs

```

1  using System;
2  using System.Collections.Generic;
3  using Xunit;
4  using Platform.Collections.Methods.Trees;
5  using Platform.Converters;
6
7  namespace Platform.Collections.Methods.Tests
8  {
9      /// <summary>
10     /// <para>
11     /// Represents the test extensions.

```

```

12  /// </para>
13  /// <para></para>
14  /// </summary>
15  public static class TestExtensions
16  {
17      /// <summary>
18      /// <para>
19      /// Tests the multiple creations and deletions using the specified tree.
20      /// </para>
21      /// <para></para>
22      /// </summary>
23      /// <typeparam name="TElement">
24      /// <para>The element.</para>
25      /// <para></para>
26      /// </typeparam>
27      /// <param name="tree">
28      /// <para>The tree.</para>
29      /// <para></para>
30      /// </param>
31      /// <param name="allocate">
32      /// <para>The allocate.</para>
33      /// <para></para>
34      /// </param>
35      /// <param name="free">
36      /// <para>The free.</para>
37      /// <para></para>
38      /// </param>
39      /// <param name="root">
40      /// <para>The root.</para>
41      /// <para></para>
42      /// </param>
43      /// <param name="treeCount">
44      /// <para>The tree count.</para>
45      /// <para></para>
46      /// </param>
47      /// <param name="maximumOperationsPerCycle">
48      /// <para>The maximum operations per cycle.</para>
49      /// <para></para>
50      /// </param>
51  public static void TestMultipleCreationsAndDeletions<TElement>(this
    ↳ SizedBinaryTreeMethodsBase<TElement> tree, Func<TElement> allocate, Action<TElement>
    ↳ free, ref TElement root, Func<TElement> treeCount, int maximumOperationsPerCycle)
52  {
53      for (var N = 1; N < maximumOperationsPerCycle; N++)
54      {
55          var currentCount = 0;
56          for (var i = 0; i < N; i++)
57          {
58              var node = allocate();
59              tree.Attach(ref root, node);
60              currentCount++;
61              Assert.Equal(currentCount, (int)UncheckedConverter<TElement,
    ↳ int>.Default.Convert(treeCount()));
62          }
63          for (var i = 1; i <= N; i++)
64          {
65              TElement node = UncheckedConverter<int, TElement>.Default.Convert(i);
66              if (tree.Contains(node, root))
67              {
68                  tree.Detach(ref root, node);
69                  free(node);
70                  currentCount--;
71                  Assert.Equal(currentCount, (int)UncheckedConverter<TElement,
    ↳ int>.Default.Convert(treeCount()));
72              }
73          }
74      }
75  }
76
77  /// <summary>
78  /// <para>
79  /// Tests the multiple random creations and deletions using the specified tree.
80  /// </para>
81  /// <para></para>
82  /// </summary>
83  /// <typeparam name="TElement">
84  /// <para>The element.</para>
85  /// <para></para>

```

```

86     /// </typeparam>
87     /// <param name="tree">
88     /// <para>The tree.</para>
89     /// <para></para>
90     /// </param>
91     /// <param name="root">
92     /// <para>The root.</para>
93     /// <para></para>
94     /// </param>
95     /// <param name="treeCount">
96     /// <para>The tree count.</para>
97     /// <para></para>
98     /// </param>
99     /// <param name="maximumOperationsPerCycle">
100    /// <para>The maximum operations per cycle.</para>
101    /// <para></para>
102    /// </param>
103    public static void TestMultipleRandomCreationsAndDeletions<TElement>(this
    ↪ SizedBinaryTreeMethodsBase<TElement> tree, ref TElement root, Func<TElement>
    ↪ treeCount, int maximumOperationsPerCycle)
104    {
105        var random = new System.Random(0);
106        var added = new HashSet<TElement>();
107        var currentCount = 0;
108        for (var N = 1; N < maximumOperationsPerCycle; N++)
109        {
110            for (var i = 0; i < N; i++)
111            {
112                var node = UncheckedConverter<int, TElement>.Default.Convert(random.Next(1,
    ↪ N));
113                if (added.Add(node))
114                {
115                    tree.Attach(ref root, node);
116                    currentCount++;
117                    Assert.Equal(currentCount, (int)UncheckedConverter<TElement,
    ↪ int>.Default.Convert(treeCount()));
118                }
119            }
120            for (var i = 1; i <= N; i++)
121            {
122                TElement node = UncheckedConverter<int,
    ↪ TElement>.Default.Convert(random.Next(1, N));
123                if (tree.Contains(node, root))
124                {
125                    tree.Detach(ref root, node);
126                    currentCount--;
127                    Assert.Equal(currentCount, (int)UncheckedConverter<TElement,
    ↪ int>.Default.Convert(treeCount()));
128                    added.Remove(node);
129                }
130            }
131        }
132    }
133 }
134 }

```

1.17 ./csharp/Platform.Collections.Methods.Tests/TreesTests.cs

```

1  using Xunit;
2
3  namespace Platform.Collections.Methods.Tests
4  {
5      /// <summary>
6      /// <para>
7      /// Represents the trees tests.
8      /// </para>
9      /// <para></para>
10     /// </summary>
11     public static class TreesTests
12     {
13         /// <summary>
14         /// <para>
15         /// The .
16         /// </para>
17         /// <para></para>
18         /// </summary>
19         private const int _n = 500;
20
21         /// <summary>

```



```

22     /// <para>
23     /// Tests that recursionless size balanced tree multiple attach and detach test.
24     /// </para>
25     /// <para></para>
26     /// </summary>
27     [Fact]
28     public static void RecursionlessSizeBalancedTreeMultipleAttachAndDetachTest()
29     {
30         var recursionlessSizeBalancedTree = new RecursionlessSizeBalancedTree<uint>(10000);
31         recursionlessSizeBalancedTree.TestMultipleCreationsAndDeletions(recursionlessSizeBal
            ↪ ancedTree.Allocate, recursionlessSizeBalancedTree.Free, ref
            ↪ recursionlessSizeBalancedTree.Root, () => recursionlessSizeBalancedTree.Count,
            ↪ _n);
32     }
33
34     /// <summary>
35     /// <para>
36     /// Tests that size balanced tree multiple attach and detach test.
37     /// </para>
38     /// <para></para>
39     /// </summary>
40     [Fact]
41     public static void SizeBalancedTreeMultipleAttachAndDetachTest()
42     {
43         var sizeBalancedTree = new SizeBalancedTree<uint>(10000);
44         sizeBalancedTree.TestMultipleCreationsAndDeletions(sizeBalancedTree.Allocate,
            ↪ sizeBalancedTree.Free, ref sizeBalancedTree.Root, () => sizeBalancedTree.Count,
            ↪ _n);
45     }
46
47     /// <summary>
48     /// <para>
49     /// Tests that sized and threaded avl balanced tree multiple attach and detach test.
50     /// </para>
51     /// <para></para>
52     /// </summary>
53     [Fact]
54     public static void SizedAndThreadedAVLBalancedTreeMultipleAttachAndDetachTest()
55     {
56         var avlTree = new SizedAndThreadedAVLBalancedTree<uint>(10000);
57         avlTree.TestMultipleCreationsAndDeletions(avlTree.Allocate, avlTree.Free, ref
            ↪ avlTree.Root, () => avlTree.Count, _n);
58     }
59
60     /// <summary>
61     /// <para>
62     /// Tests that recursionless size balanced tree multiple random attach and detach test.
63     /// </para>
64     /// <para></para>
65     /// </summary>
66     [Fact]
67     public static void RecursionlessSizeBalancedTreeMultipleRandomAttachAndDetachTest()
68     {
69         var recursionlessSizeBalancedTree = new RecursionlessSizeBalancedTree<uint>(10000);
70         recursionlessSizeBalancedTree.TestMultipleRandomCreationsAndDeletions(ref
            ↪ recursionlessSizeBalancedTree.Root, () => recursionlessSizeBalancedTree.Count,
            ↪ _n);
71     }
72
73     /// <summary>
74     /// <para>
75     /// Tests that size balanced tree multiple random attach and detach test.
76     /// </para>
77     /// <para></para>
78     /// </summary>
79     [Fact]
80     public static void SizeBalancedTreeMultipleRandomAttachAndDetachTest()
81     {
82         var sizeBalancedTree = new SizeBalancedTree<uint>(10000);
83         sizeBalancedTree.TestMultipleRandomCreationsAndDeletions(ref sizeBalancedTree.Root,
            ↪ () => sizeBalancedTree.Count, _n);
84     }
85
86     /// <summary>
87     /// <para>
88     /// Tests that sized and threaded avl balanced tree multiple random attach and detach
            ↪ test.
89     /// </para>

```

```
90     /// <para></para>
91     /// </summary>
92     [Fact]
93     public static void SizedAndThreadedAVLBalancedTreeMultipleRandomAttachAndDetachTest()
94     {
95         var avlTree = new SizedAndThreadedAVLBalancedTree<uint>(10000);
96         avlTree.TestMultipleRandomCreationsAndDeletions(ref avlTree.Root, () =>
97             ↪ avlTree.Count, _n);
98     }
99 }
```

Index

- ./csharp/Platform.Collections.Methods.Tests/RecursionlessSizeBalancedTree.cs, 46
- ./csharp/Platform.Collections.Methods.Tests/SizeBalancedTree.cs, 51
- ./csharp/Platform.Collections.Methods.Tests/SizedAndThreadedAVLBalancedTree.cs, 56
- ./csharp/Platform.Collections.Methods.Tests/TestExtensions.cs, 62
- ./csharp/Platform.Collections.Methods.Tests/TreesTests.cs, 64
- ./csharp/Platform.Collections.Methods/GenericCollectionMethodsBase.cs, 1
- ./csharp/Platform.Collections.Methods/Lists/AbsoluteCircularDoublyLinkedListMethods.cs, 3
- ./csharp/Platform.Collections.Methods/Lists/AbsoluteDoublyLinkedListMethodsBase.cs, 5
- ./csharp/Platform.Collections.Methods/Lists/AbsoluteOpenDoublyLinkedListMethods.cs, 7
- ./csharp/Platform.Collections.Methods/Lists/DoublyLinkedListMethodsBase.cs, 9
- ./csharp/Platform.Collections.Methods/Lists/RelativeCircularDoublyLinkedListMethods.cs, 10
- ./csharp/Platform.Collections.Methods/Lists/RelativeDoublyLinkedListMethodsBase.cs, 12
- ./csharp/Platform.Collections.Methods/Lists/RelativeOpenDoublyLinkedListMethods.cs, 14
- ./csharp/Platform.Collections.Methods/Trees/RecursionlessSizeBalancedTreeMethods.cs, 17
- ./csharp/Platform.Collections.Methods/Trees/SizeBalancedTreeMethods.cs, 20
- ./csharp/Platform.Collections.Methods/Trees/SizedAndThreadedAVLBalancedTreeMethods.cs, 23
- ./csharp/Platform.Collections.Methods/Trees/SizedBinaryTreeMethodsBase.cs, 35