

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>

```

```

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 }
133 }
134 }
135 }
136 }
137 }
138 }
139 }
140 }
141 }
142 }
143 }
144 }
145 }
146 }

/// <summary>
/// <para>
/// Detaches the core using the specified root.
/// </para>
/// <para></para>
/// </summary>
/// <param name="root">
/// <para>The root.</para>
/// <para></para>
/// </param>
/// <param name="node">
/// <para>The node.</para>
/// <para></para>
/// </param>
protected override void DetachCore(ref TElement root, TElement node)
{

```

```

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 private void LeftMaintain(ref TElement root)
137 {
138     if (!EqualToZero(root))
139     {
140         var rootLeftNode = GetLeft(root);
141         if (!EqualToZero(rootLeftNode))
142         {
143             var rootRightNode = GetRight(root);
144             var rootRightNodeSize = GetSize(rootRightNode);
145             var rootLeftNodeLeftNode = GetLeft(rootLeftNode);
146             if (!EqualToZero(rootLeftNodeLeftNode) &&

```

```

147         (EqualToZero(rootRightNode) ||
148             ⇨ GreaterThan(GetSize(rootLeftNodeLeftNode), rootRightNodeSize)))
149     {
150         RightRotate(ref root);
151     }
152     else
153     {
154         var rootLeftNodeRightNode = GetRight(rootLeftNode);
155         if (!EqualToZero(rootLeftNodeRightNode) &&
156             (EqualToZero(rootRightNode) ||
157                 ⇨ GreaterThan(GetSize(rootLeftNodeRightNode), rootRightNodeSize)))
158         {
159             LeftRotate(ref GetLeftReference(root));
160             RightRotate(ref root);
161         }
162         else
163         {
164             return;
165         }
166     }
167     LeftMaintain(ref GetLeftReference(root));
168     RightMaintain(ref GetRightReference(root));
169     LeftMaintain(ref root);
170     RightMaintain(ref root);
171 }
172
173 private void RightMaintain(ref TElement root)
174 {
175     if (!EqualToZero(root))
176     {
177         var rootRightNode = GetRight(root);
178         if (!EqualToZero(rootRightNode))
179         {
180             var rootLeftNode = GetLeft(root);
181             var rootLeftNodeSize = GetSize(rootLeftNode);
182             var rootRightNodeRightNode = GetRight(rootRightNode);
183             if (!EqualToZero(rootRightNodeRightNode) &&
184                 (EqualToZero(rootLeftNode) ||
185                     ⇨ GreaterThan(GetSize(rootRightNodeRightNode), rootLeftNodeSize)))
186             {
187                 LeftRotate(ref root);
188             }
189             else
190             {
191                 var rootRightNodeLeftNode = GetLeft(rootRightNode);
192                 if (!EqualToZero(rootRightNodeLeftNode) &&
193                     (EqualToZero(rootLeftNode) ||
194                         ⇨ GreaterThan(GetSize(rootRightNodeLeftNode), rootLeftNodeSize)))
195                 {
196                     RightRotate(ref GetRightReference(root));
197                     LeftRotate(ref root);
198                 }
199             }
200             else
201             {
202                 return;
203             }
204         }
205         LeftMaintain(ref GetLeftReference(root));
206         RightMaintain(ref GetRightReference(root));
207         LeftMaintain(ref root);
208         RightMaintain(ref root);
209     }
210 }

```

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         private static readonly int _maxPath = 11 * NumericType<TElement>.BytesSize + 4;
26
27         /// <summary>
28         /// <para>
29         /// Gets the rightest using the specified current.
30         /// </para>
31         /// <para></para>
32         /// </summary>
33         /// <param name="current">
34         /// <para>The current.</para>
35         /// <para></para>
36         /// </param>
37         /// <returns>
38         /// <para>The current.</para>
39         /// <para></para>
40         /// </returns>
41         [MethodImpl(MethodImplOptions.AggressiveInlining)]
42         protected override TElement GetRightest(TElement current)
43         {
44             var currentRight = GetRightOrDefault(current);
45             while (!EqualToZero(currentRight))
46             {
47                 current = currentRight;
48                 currentRight = GetRightOrDefault(current);
49             }
50             return current;
51         }
52
53         /// <summary>
54         /// <para>
55         /// Gets the leftest using the specified current.
56         /// </para>
57         /// <para></para>
58         /// </summary>
59         /// <param name="current">
60         /// <para>The current.</para>
61         /// <para></para>
62         /// </param>
63         /// <returns>
64         /// <para>The current.</para>
65         /// <para></para>
66         /// </returns>
67         [MethodImpl(MethodImplOptions.AggressiveInlining)]
68         protected override TElement GetLeftest(TElement current)
69         {
70             var currentLeft = GetLeftOrDefault(current);
71             while (!EqualToZero(currentLeft))
72             {
73                 current = currentLeft;
74                 currentLeft = GetLeftOrDefault(current);
75             }
76             return current;
77         }
78
79         /// <summary>
80         /// <para>
81         /// Determines whether this instance contains.
82         /// </para>
83         /// <para></para>
84         /// </summary>
85         /// <param name="node">
86         /// <para>The node.</para>
87         /// <para></para>
88         /// </param>

```

```

85    /// </param>
86    /// <param name="root">
87    /// <para>The root.</para>
88    /// <para></para>
89    /// </param>
90    /// <returns>
91    /// <para>The bool</para>
92    /// <para></para>
93    /// </returns>
94    public override bool Contains(TElement node, TElement root)
95    {
96        while (!EqualToZero(root))
97        {
98            if (FirstIsToTheLeftOfSecond(node, root)) // node.Key < root.Key
99            {
100                root = GetLeftOrDefault(root);
101            }
102            else if (FirstIsToTheRightOfSecond(node, root)) // node.Key > root.Key
103            {
104                root = GetRightOrDefault(root);
105            }
106            else // node.Key == root.Key
107            {
108                return true;
109            }
110        }
111        return false;
112    }
113
114    /// <summary>
115    /// <para>
116    /// Prints the node using the specified node.
117    /// </para>
118    /// <para></para>
119    /// </summary>
120    /// <param name="node">
121    /// <para>The node.</para>
122    /// <para></para>
123    /// </param>
124    /// <param name="sb">
125    /// <para>The sb.</para>
126    /// <para></para>
127    /// </param>
128    /// <param name="level">
129    /// <para>The level.</para>
130    /// <para></para>
131    /// </param>
132    protected override void PrintNode(TElement node, StringBuilder sb, int level)
133    {
134        base.PrintNode(node, sb, level);
135        sb.Append(' ');
136        sb.Append(GetLeftIsChild(node) ? 'l' : 'L');
137        sb.Append(GetRightIsChild(node) ? 'r' : 'R');
138        sb.Append(' ');
139        sb.Append(GetBalance(node));
140    }
141
142    /// <summary>
143    /// <para>
144    /// Increments the balance using the specified node.
145    /// </para>
146    /// <para></para>
147    /// </summary>
148    /// <param name="node">
149    /// <para>The node.</para>
150    /// <para></para>
151    /// </param>
152    [MethodImpl(MethodImplOptions.AggressiveInlining)]
153    protected void IncrementBalance(TElement node) => SetBalance(node,
154        ↪ (sbyte)(GetBalance(node) + 1));
155
156    /// <summary>
157    /// <para>
158    /// Decrements the balance using the specified node.
159    /// </para>
160    /// <para></para>
161    /// </summary>
162    /// <param name="node">

```



```

162    /// <para>The node.</para>
163    /// <para></para>
164    /// </param>
165    [MethodImpl(MethodImplOptions.AggressiveInlining)]
166    protected void DecrementBalance(TElement node) => SetBalance(node,
    ↪    (sbyte)(GetBalance(node) - 1));

167
168    /// <summary>
169    /// <para>
170    /// Gets the left or default using the specified node.
171    /// </para>
172    /// <para></para>
173    /// </summary>
174    /// <param name="node">
175    /// <para>The node.</para>
176    /// <para></para>
177    /// </param>
178    /// <returns>
179    /// <para>The element</para>
180    /// <para></para>
181    /// </returns>
182    [MethodImpl(MethodImplOptions.AggressiveInlining)]
183    protected override TElement GetLeftOrDefault(TElement node) => GetLeftIsChild(node) ?
    ↪    GetLeft(node) : default;

184
185    /// <summary>
186    /// <para>
187    /// Gets the right or default using the specified node.
188    /// </para>
189    /// <para></para>
190    /// </summary>
191    /// <param name="node">
192    /// <para>The node.</para>
193    /// <para></para>
194    /// </param>
195    /// <returns>
196    /// <para>The element</para>
197    /// <para></para>
198    /// </returns>
199    [MethodImpl(MethodImplOptions.AggressiveInlining)]
200    protected override TElement GetRightOrDefault(TElement node) => GetRightIsChild(node) ?
    ↪    GetRight(node) : default;

201
202    /// <summary>
203    /// <para>
204    /// Determines whether this instance get left is child.
205    /// </para>
206    /// <para></para>
207    /// </summary>
208    /// <param name="node">
209    /// <para>The node.</para>
210    /// <para></para>
211    /// </param>
212    /// <returns>
213    /// <para>The bool</para>
214    /// <para></para>
215    /// </returns>
216    [MethodImpl(MethodImplOptions.AggressiveInlining)]
217    protected abstract bool GetLeftIsChild(TElement node);

218
219    /// <summary>
220    /// <para>
221    /// Sets the left is child using the specified node.
222    /// </para>
223    /// <para></para>
224    /// </summary>
225    /// <param name="node">
226    /// <para>The node.</para>
227    /// <para></para>
228    /// </param>
229    /// <param name="value">
230    /// <para>The value.</para>
231    /// <para></para>
232    /// </param>
233    [MethodImpl(MethodImplOptions.AggressiveInlining)]
234    protected abstract void SetLeftIsChild(TElement node, bool value);

235
236    /// <summary>

```

```

237     /// <para>
238     /// Determines whether this instance get right is child.
239     /// </para>
240     /// <para></para>
241     /// </summary>
242     /// <param name="node">
243     /// <para>The node.</para>
244     /// <para></para>
245     /// </param>
246     /// <returns>
247     /// <para>The bool</para>
248     /// <para></para>
249     /// </returns>
250     [MethodImpl(MethodImplOptions.AggressiveInlining)]
251     protected abstract bool GetRightIsChild(TElement node);
252
253     /// <summary>
254     /// <para>
255     /// Sets the right is child using the specified node.
256     /// </para>
257     /// <para></para>
258     /// </summary>
259     /// <param name="node">
260     /// <para>The node.</para>
261     /// <para></para>
262     /// </param>
263     /// <param name="value">
264     /// <para>The value.</para>
265     /// <para></para>
266     /// </param>
267     [MethodImpl(MethodImplOptions.AggressiveInlining)]
268     protected abstract void SetRightIsChild(TElement node, bool value);
269
270     /// <summary>
271     /// <para>
272     /// Gets the balance using the specified node.
273     /// </para>
274     /// <para></para>
275     /// </summary>
276     /// <param name="node">
277     /// <para>The node.</para>
278     /// <para></para>
279     /// </param>
280     /// <returns>
281     /// <para>The sbyte</para>
282     /// <para></para>
283     /// </returns>
284     [MethodImpl(MethodImplOptions.AggressiveInlining)]
285     protected abstract sbyte GetBalance(TElement node);
286
287     /// <summary>
288     /// <para>
289     /// Sets the balance using the specified node.
290     /// </para>
291     /// <para></para>
292     /// </summary>
293     /// <param name="node">
294     /// <para>The node.</para>
295     /// <para></para>
296     /// </param>
297     /// <param name="value">
298     /// <para>The value.</para>
299     /// <para></para>
300     /// </param>
301     [MethodImpl(MethodImplOptions.AggressiveInlining)]
302     protected abstract void SetBalance(TElement node, sbyte value);
303
304     /// <summary>
305     /// <para>
306     /// Attaches the core using the specified root.
307     /// </para>
308     /// <para></para>
309     /// </summary>
310     /// <param name="root">
311     /// <para>The root.</para>
312     /// <para></para>
313     /// </param>
314     /// <param name="node">

```

```

315     /// <para>The node.</para>
316     /// <para></para>
317     /// </param>
318     /// <exception cref="InvalidOperationException">
319     /// <para>Node with the same key already attached to a tree.</para>
320     /// <para></para>
321     /// </exception>
322     protected override void AttachCore(ref TElement root, TElement node)
323     {
324         unchecked
325         {
326             // TODO: Check what is faster to use simple array or array from array pool
327             // TODO: Try to use stackalloc as an optimization (requires code generation,
328             // ↳ because of generics)
329             #if USEARRAYPOOL
330                 var path = ArrayPool.Allocate<TElement>(MaxPath);
331                 var pathPosition = 0;
332                 path[pathPosition++] = default;
333             #else
334                 var path = new TElement[_maxPath];
335                 var pathPosition = 1;
336             #endif
337             var currentNode = root;
338             while (true)
339             {
340                 if (FirstIsToTheLeftOfSecond(node, currentNode))
341                 {
342                     if (GetLeftIsChild(currentNode))
343                     {
344                         IncrementSize(currentNode);
345                         path[pathPosition++] = currentNode;
346                         currentNode = GetLeft(currentNode);
347                     }
348                     else
349                     {
350                         // Threads
351                         SetLeft(node, GetLeft(currentNode));
352                         SetRight(node, currentNode);
353                         SetLeft(currentNode, node);
354                         SetLeftIsChild(currentNode, true);
355                         DecrementBalance(currentNode);
356                         SetSize(node, One);
357                         FixSize(currentNode); // Should be incremented already
358                         break;
359                     }
360                 }
361                 else if (FirstIsToTheRightOfSecond(node, currentNode))
362                 {
363                     if (GetRightIsChild(currentNode))
364                     {
365                         IncrementSize(currentNode);
366                         path[pathPosition++] = currentNode;
367                         currentNode = GetRight(currentNode);
368                     }
369                     else
370                     {
371                         // Threads
372                         SetRight(node, GetRight(currentNode));
373                         SetLeft(node, currentNode);
374                         SetRight(currentNode, node);
375                         SetRightIsChild(currentNode, true);
376                         IncrementBalance(currentNode);
377                         SetSize(node, One);
378                         FixSize(currentNode); // Should be incremented already
379                         break;
380                     }
381                 }
382                 else
383                 {
384                     throw new InvalidOperationException("Node with the same key already
385                     ↳ attached to a tree.");
386                 }
387             }
388             // Restore balance. This is the goodness of a non-recursive
389             // implementation, when we are done with balancing we 'break'
390             // the loop and we are done.
391             while (true)
392             {

```

```

391     var parent = path[--pathPosition];
392     var isLeftNode = !AreEqual(parent, default) && AreEqual(currentNode,
    ↪ GetLeft(parent));
393     var currentNodeBalance = GetBalance(currentNode);
394     if (currentNodeBalance < -1 || currentNodeBalance > 1)
395     {
396         currentNode = Balance(currentNode);
397         if (AreEqual(parent, default))
398         {
399             root = currentNode;
400         }
401         else if (isLeftNode)
402         {
403             SetLeft(parent, currentNode);
404             FixSize(parent);
405         }
406         else
407         {
408             SetRight(parent, currentNode);
409             FixSize(parent);
410         }
411     }
412     currentNodeBalance = GetBalance(currentNode);
413     if (currentNodeBalance == 0 || AreEqual(parent, default))
414     {
415         break;
416     }
417     if (isLeftNode)
418     {
419         DecrementBalance(parent);
420     }
421     else
422     {
423         IncrementBalance(parent);
424     }
425     currentNode = parent;
426 }
427 #if USEARRAYPOOL
428     ArrayPool.Free(path);
429 #endif
430 }
431 }
432
433 private TElement Balance(TElement node)
434 {
435     unchecked
436     {
437         var rootBalance = GetBalance(node);
438         if (rootBalance < -1)
439         {
440             var left = GetLeft(node);
441             if (GetBalance(left) > 0)
442             {
443                 SetLeft(node, LeftRotateWithBalance(left));
444                 FixSize(node);
445             }
446             node = RightRotateWithBalance(node);
447         }
448         else if (rootBalance > 1)
449         {
450             var right = GetRight(node);
451             if (GetBalance(right) < 0)
452             {
453                 SetRight(node, RightRotateWithBalance(right));
454                 FixSize(node);
455             }
456             node = LeftRotateWithBalance(node);
457         }
458         return node;
459     }
460 }
461
462 /// <summary>
463 /// <para>
464 /// Lefts the rotate with balance using the specified node.
465 /// </para>
466 /// <para></para>
467 /// </summary>

```

```

468  /// <param name="node">
469  /// <para>The node.</para>
470  /// <para></para>
471  /// </param>
472  /// <returns>
473  /// <para>The element</para>
474  /// <para></para>
475  /// </returns>
476  protected TElement LeftRotateWithBalance(TElement node)
477  {
478      unchecked
479      {
480          var right = GetRight(node);
481          if (GetLeftIsChild(right))
482          {
483              SetRight(node, GetLeft(right));
484          }
485          else
486          {
487              SetRightIsChild(node, false);
488              SetLeftIsChild(right, true);
489          }
490          SetLeft(right, node);
491          // Fix size
492          SetSize(right, GetSize(node));
493          FixSize(node);
494          // Fix balance
495          var rootBalance = GetBalance(node);
496          var rightBalance = GetBalance(right);
497          if (rightBalance <= 0)
498          {
499              if (rootBalance >= 1)
500              {
501                  SetBalance(right, (sbyte)(rightBalance - 1));
502              }
503              else
504              {
505                  SetBalance(right, (sbyte)(rootBalance + rightBalance - 2));
506              }
507              SetBalance(node, (sbyte)(rootBalance - 1));
508          }
509          else
510          {
511              if (rootBalance <= rightBalance)
512              {
513                  SetBalance(right, (sbyte)(rootBalance - 2));
514              }
515              else
516              {
517                  SetBalance(right, (sbyte)(rightBalance - 1));
518              }
519              SetBalance(node, (sbyte)(rootBalance - rightBalance - 1));
520          }
521          return right;
522      }
523  }
524
525  /// <summary>
526  /// <para>
527  /// Rights the rotate with balance using the specified node.
528  /// </para>
529  /// <para></para>
530  /// </summary>
531  /// <param name="node">
532  /// <para>The node.</para>
533  /// <para></para>
534  /// </param>
535  /// <returns>
536  /// <para>The element</para>
537  /// <para></para>
538  /// </returns>
539  protected TElement RightRotateWithBalance(TElement node)
540  {
541      unchecked
542      {
543          var left = GetLeft(node);
544          if (GetRightIsChild(left))
545          {

```

```

546         SetLeft(node, GetRight(left));
547     }
548     else
549     {
550         SetLeftIsChild(node, false);
551         SetRightIsChild(left, true);
552     }
553     SetRight(left, node);
554     // Fix size
555     SetSize(left, GetSize(node));
556     FixSize(node);
557     // Fix balance
558     var rootBalance = GetBalance(node);
559     var leftBalance = GetBalance(left);
560     if (leftBalance <= 0)
561     {
562         if (leftBalance > rootBalance)
563         {
564             SetBalance(left, (sbyte)(leftBalance + 1));
565         }
566         else
567         {
568             SetBalance(left, (sbyte)(rootBalance + 2));
569         }
570         SetBalance(node, (sbyte)(rootBalance - leftBalance + 1));
571     }
572     else
573     {
574         if (rootBalance <= -1)
575         {
576             SetBalance(left, (sbyte)(leftBalance + 1));
577         }
578         else
579         {
580             SetBalance(left, (sbyte)(rootBalance + leftBalance + 2));
581         }
582         SetBalance(node, (sbyte)(rootBalance + 1));
583     }
584     return left;
585 }
586
587
588 /// <summary>
589 /// <para>
590 /// Gets the next using the specified node.
591 /// </para>
592 /// <para></para>
593 /// </summary>
594 /// <param name="node">
595 /// <para>The node.</para>
596 /// <para></para>
597 /// </param>
598 /// <returns>
599 /// <para>The current.</para>
600 /// <para></para>
601 /// </returns>
602 [MethodImpl(MethodImplOptions.AggressiveInlining)]
603 protected override TElement GetNext(TElement node)
604 {
605     var current = GetRight(node);
606     if (GetRightIsChild(node))
607     {
608         return GetLefttest(current);
609     }
610     return current;
611 }
612
613 /// <summary>
614 /// <para>
615 /// Gets the previous using the specified node.
616 /// </para>
617 /// <para></para>
618 /// </summary>
619 /// <param name="node">
620 /// <para>The node.</para>
621 /// <para></para>
622 /// </param>
623 /// </returns>

```

```

624    /// <para>The current.</para>
625    /// <para></para>
626    /// </returns>
627    [MethodImpl(MethodImplOptions.AggressiveInlining)]
628    protected override TElement GetPrevious(TElement node)
629    {
630        var current = GetLeft(node);
631        if (GetLeftIsChild(node))
632        {
633            return GetRightest(current);
634        }
635        return current;
636    }
637
638    /// <summary>
639    /// <para>
640    /// Detaches the core using the specified root.
641    /// </para>
642    /// <para></para>
643    /// </summary>
644    /// <param name="root">
645    /// <para>The root.</para>
646    /// <para></para>
647    /// </param>
648    /// <param name="node">
649    /// <para>The node.</para>
650    /// <para></para>
651    /// </param>
652    /// <exception cref="InvalidOperationException">
653    /// <para>Cannot find a node.</para>
654    /// <para></para>
655    /// </exception>
656    /// <exception cref="InvalidOperationException">
657    /// <para>Cannot find a node.</para>
658    /// <para></para>
659    /// </exception>
660    protected override void DetachCore(ref TElement root, TElement node)
661    {
662        unchecked
663        {
664            #if USEARRAYPOOL
665                var path = ArrayPool.Allocate<TElement>(MaxPath);
666                var pathPosition = 0;
667                path[pathPosition++] = default;
668            #else
669                var path = new TElement[_maxPath];
670                var pathPosition = 1;
671            #endif
672            var currentNode = root;
673            while (true)
674            {
675                if (FirstIsToTheLeftOfSecond(node, currentNode))
676                {
677                    if (!GetLeftIsChild(currentNode))
678                    {
679                        throw new InvalidOperationException("Cannot find a node.");
680                    }
681                    DecrementSize(currentNode);
682                    path[pathPosition++] = currentNode;
683                    currentNode = GetLeft(currentNode);
684                }
685                else if (FirstIsToTheRightOfSecond(node, currentNode))
686                {
687                    if (!GetRightIsChild(currentNode))
688                    {
689                        throw new InvalidOperationException("Cannot find a node.");
690                    }
691                    DecrementSize(currentNode);
692                    path[pathPosition++] = currentNode;
693                    currentNode = GetRight(currentNode);
694                }
695                else
696                {
697                    break;
698                }
699            }
700            var parent = path[--pathPosition];
701            var balanceNode = parent;

```

```

702 var isLeftNode = !AreEqual(parent, default) && AreEqual(currentNode,
    ↪ GetLeft(parent));
703 if (!GetLeftIsChild(currentNode))
704 {
705     if (!GetRightIsChild(currentNode)) // node has no children
706     {
707         if (AreEqual(parent, default))
708         {
709             root = Zero;
710         }
711         else if (isLeftNode)
712         {
713             SetLeftIsChild(parent, false);
714             SetLeft(parent, GetLeft(currentNode));
715             IncrementBalance(parent);
716         }
717         else
718         {
719             SetRightIsChild(parent, false);
720             SetRight(parent, GetRight(currentNode));
721             DecrementBalance(parent);
722         }
723     }
724     else // node has a right child
725     {
726         var successor = GetNext(currentNode);
727         SetLeft(successor, GetLeft(currentNode));
728         var right = GetRight(currentNode);
729         if (AreEqual(parent, default))
730         {
731             root = right;
732         }
733         else if (isLeftNode)
734         {
735             SetLeft(parent, right);
736             IncrementBalance(parent);
737         }
738         else
739         {
740             SetRight(parent, right);
741             DecrementBalance(parent);
742         }
743     }
744 }
745 else // node has a left child
746 {
747     if (!GetRightIsChild(currentNode))
748     {
749         var predecessor = GetPrevious(currentNode);
750         SetRight(predecessor, GetRight(currentNode));
751         var leftValue = GetLeft(currentNode);
752         if (AreEqual(parent, default))
753         {
754             root = leftValue;
755         }
756         else if (isLeftNode)
757         {
758             SetLeft(parent, leftValue);
759             IncrementBalance(parent);
760         }
761         else
762         {
763             SetRight(parent, leftValue);
764             DecrementBalance(parent);
765         }
766     }
767     else // node has a both children (left and right)
768     {
769         var predecessor = GetLeft(currentNode);
770         var successor = GetRight(currentNode);
771         var successorParent = currentNode;
772         int previousPathPosition = ++pathPosition;
773         // find the immediately next node (and its parent)
774         while (GetLeftIsChild(successor))
775         {
776             path[++pathPosition] = successorParent = successor;
777             successor = GetLeft(successor);
778             if (!AreEqual(successorParent, currentNode))

```



```

779         {
780             DecrementSize(successorParent);
781         }
782     }
783     path[previousPathPosition] = successor;
784     balanceNode = path[pathPosition];
785     // remove 'successor' from the tree
786     if (!AreEqual(successorParent, currentNode))
787     {
788         if (!GetRightIsChild(successor))
789         {
790             SetLeftIsChild(successorParent, false);
791         }
792         else
793         {
794             SetLeft(successorParent, GetRight(successor));
795         }
796         IncrementBalance(successorParent);
797         SetRightIsChild(successor, true);
798         SetRight(successor, GetRight(currentNode));
799     }
800     else
801     {
802         DecrementBalance(currentNode);
803     }
804     // set the predecessor's successor link to point to the right place
805     while (GetRightIsChild(predecessor))
806     {
807         predecessor = GetRight(predecessor);
808     }
809     SetRight(predecessor, successor);
810     // prepare 'successor' to replace 'node'
811     var left = GetLeft(currentNode);
812     SetLeftIsChild(successor, true);
813     SetLeft(successor, left);
814     SetBalance(successor, GetBalance(currentNode));
815     FixSize(successor);
816     if (AreEqual(parent, default))
817     {
818         root = successor;
819     }
820     else if (isLeftNode)
821     {
822         SetLeft(parent, successor);
823     }
824     else
825     {
826         SetRight(parent, successor);
827     }
828 }
829 }
830 // restore balance
831 if (!AreEqual(balanceNode, default))
832 {
833     while (true)
834     {
835         var balanceParent = path[--pathPosition];
836         isLeftNode = !AreEqual(balanceParent, default) && AreEqual(balanceNode,
837             ↪ GetLeft(balanceParent));
838         var currentNodeBalance = GetBalance(balanceNode);
839         if (currentNodeBalance < -1 || currentNodeBalance > 1)
840         {
841             balanceNode = Balance(balanceNode);
842             if (AreEqual(balanceParent, default))
843             {
844                 root = balanceNode;
845             }
846             else if (isLeftNode)
847             {
848                 SetLeft(balanceParent, balanceNode);
849             }
850             else
851             {
852                 SetRight(balanceParent, balanceNode);
853             }
854         }
855         currentNodeBalance = GetBalance(balanceNode);
856         if (currentNodeBalance != 0 || AreEqual(balanceParent, default))

```

```

856         {
857             break;
858         }
859         if (isLeftNode)
860         {
861             IncrementBalance(balanceParent);
862         }
863         else
864         {
865             DecrementBalance(balanceParent);
866         }
867         balanceNode = balanceParent;
868     }
869 }
870 ClearNode(node);
871 #if USEARRAYPOOL
872     ArrayPool.Free(path);
873 #endif
874 }
875 }
876
877 /// <summary>
878 /// <para>
879 /// Clears the node using the specified node.
880 /// </para>
881 /// <para></para>
882 /// </summary>
883 /// <param name="node">
884 /// <para>The node.</para>
885 /// <para></para>
886 /// </param>
887 [MethodImpl(MethodImplOptions.AggressiveInlining)]
888 protected override void ClearNode(TElement node)
889 {
890     SetLeft(node, Zero);
891     SetRight(node, Zero);
892     SetSize(node, Zero);
893     SetLeftIsChild(node, false);
894     SetRightIsChild(node, false);
895     SetBalance(node, 0);
896 }
897 }
898 }

```

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>

```

```

34     /// <para></para>
35     /// </returns>
36     [MethodImpl(MethodImplOptions.AggressiveInlining)]
37     protected abstract ref TElement GetLeftReference(TElement node);
38
39     /// <summary>
40     /// <para>
41     /// Gets the right reference using the specified node.
42     /// </para>
43     /// <para></para>
44     /// </summary>
45     /// <param name="node">
46     /// <para>The node.</para>
47     /// <para></para>
48     /// </param>
49     /// <returns>
50     /// <para>The ref element</para>
51     /// <para></para>
52     /// </returns>
53     [MethodImpl(MethodImplOptions.AggressiveInlining)]
54     protected abstract ref TElement GetRightReference(TElement node);
55
56     /// <summary>
57     /// <para>
58     /// Gets the left using the specified node.
59     /// </para>
60     /// <para></para>
61     /// </summary>
62     /// <param name="node">
63     /// <para>The node.</para>
64     /// <para></para>
65     /// </param>
66     /// <returns>
67     /// <para>The element</para>
68     /// <para></para>
69     /// </returns>
70     [MethodImpl(MethodImplOptions.AggressiveInlining)]
71     protected abstract TElement GetLeft(TElement node);
72
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 FirstIsToTheLeftOfSecond(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 GetRighttest(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 lefttest 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 GetLefttest(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
        ↳ дереве.");
641     }
642     #endif
643     DetachCore(ref root, node);
644     #if ENABLE_TREE_AUTO_DEBUG_AND_VALIDATION
645     Debug.WriteLine("--AfterDetach--");
646     Debug.WriteLine(PrintNodes(root));
647     Debug.WriteLine("-----");
648     ValidateSizes(root);
649     var sizeAfter = GetSize(root);
650     if (!AreEqual(Arithmetic.Decrement(sizeBefore), sizeAfter))
651     {
652         throw new InvalidOperationException("Tree was broken after detach.");

```

```

653     }
654 #endif
655 }
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:
706   ↳ {size}.</para>
707 /// <para></para>
708 /// </exception>
709 public void ValidateSizes(TElement node)
710 {
711     if (AreEqual(node, default))
712     {
713         return;
714     }
715     var size = GetSize(node);
716     var leftSize = GetLeftSize(node);
717     var rightSize = GetRightSize(node);
718     var expectedSize = Arithmetic.Increment(Arithmetic.Add(leftSize, rightSize));
719     if (!AreEqual(size, expectedSize))
720     {
721         throw new InvalidOperationException($"Size of {node} is not valid. Expected
722   ↳ size: {expectedSize}, actual size: {size}.");
723     }
724     ValidateSizes(GetLeft(node));
725     ValidateSizes(GetRight(node));
726 }
727
728 /// <summary>
729 /// <para>
730 /// 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> :
18         ↪ RecursionlessSizeBalancedTreeMethods<TElement>
19     {
20         private struct TreeElement
21         {
22             /// <summary>
23             /// <para>
24             /// The size.
25             /// </para>
26             /// <para></para>
27             /// </summary>
28             public TElement Size;
29             /// <summary>
30             /// <para>
31             /// The left.
32             /// </para>
33             /// <para></para>
34             /// </summary>
35             public TElement Left;
36             /// <summary>
37             /// <para>
38             /// The right.
39             /// </para>
40             /// <para></para>
41             /// </summary>
42             public TElement Right;
43         }
44
45         private readonly TreeElement[] _elements;
46         private TElement _allocated;
47
48         /// <summary>
49         /// <para>
50         /// The root.
51         /// </para>
52         /// <para></para>
53         /// </summary>
54         public TElement Root;
55
56         /// <summary>
57         /// <para>

```

```

57     /// Gets the count value.
58     /// </para>
59     /// <para></para>
60     /// </summary>
61     public TElement Count => GetSizeOrZero(Root);
62
63     /// <summary>
64     /// <para>
65     /// Initializes a new <see cref="RecursionlessSizeBalancedTree"/> instance.
66     /// </para>
67     /// <para></para>
68     /// </summary>
69     /// <param name="capacity">
70     /// <para>A capacity.</para>
71     /// <para></para>
72     /// </param>
73     public RecursionlessSizeBalancedTree(int capacity) => (_elements, _allocated) = (new
    ↪ TreeElement[capacity], One);
74
75     /// <summary>
76     /// <para>
77     /// Allocates this instance.
78     /// </para>
79     /// <para></para>
80     /// </summary>
81     /// <exception cref="InvalidOperationException">
82     /// <para>Allocated tree element is not empty.</para>
83     /// <para></para>
84     /// </exception>
85     /// <returns>
86     /// <para>The element</para>
87     /// <para></para>
88     /// </returns>
89     public TElement Allocate()
90     {
91         var newNode = _allocated;
92         if (IsEmpty(newNode))
93         {
94             _allocated = Arithmetic.Increment(_allocated);
95             return newNode;
96         }
97         else
98         {
99             throw new InvalidOperationException("Allocated tree element is not empty.");
100         }
101     }
102
103     /// <summary>
104     /// <para>
105     /// Frees the node.
106     /// </para>
107     /// <para></para>
108     /// </summary>
109     /// <param name="node">
110     /// <para>The node.</para>
111     /// <para></para>
112     /// </param>
113     public void Free(TElement node)
114     {
115         while (!EqualityComparer.Equals(_allocated, One) && IsEmpty(node))
116         {
117             var lastNode = Arithmetic.Decrement(_allocated);
118             if (EqualityComparer.Equals(lastNode, node))
119             {
120                 _allocated = lastNode;
121                 node = Arithmetic.Decrement(node);
122             }
123             else
124             {
125                 return;
126             }
127         }
128     }
129
130     /// <summary>
131     /// <para>
132     /// Determines whether this instance is empty.
133     /// </para>

```

```

134    /// <para></para>
135    /// </summary>
136    /// <param name="node">
137    /// <para>The node.</para>
138    /// <para></para>
139    /// </param>
140    /// <returns>
141    /// <para>The bool</para>
142    /// <para></para>
143    /// </returns>
144    public bool IsEmpty(TElement node) =>
145        ↪ EqualityComparer<TreeElement>.Default.Equals(GetElement(node), default);
146
147    /// <summary>
148    /// <para>
149    /// Determines whether this instance first is to the left of second.
150    /// </para>
151    /// <para></para>
152    /// </summary>
153    /// <param name="first">
154    /// <para>The first.</para>
155    /// <para></para>
156    /// </param>
157    /// <param name="second">
158    /// <para>The second.</para>
159    /// <para></para>
160    /// </param>
161    /// <returns>
162    /// <para>The bool</para>
163    /// <para></para>
164    /// </returns>
165    protected override bool FirstIsToTheLeftOfSecond(TElement first, TElement second) =>
166        ↪ Comparer.Compare(first, second) < 0;
167
168    /// <summary>
169    /// <para>
170    /// Determines whether this instance first is to the right of second.
171    /// </para>
172    /// <para></para>
173    /// </summary>
174    /// <param name="first">
175    /// <para>The first.</para>
176    /// <para></para>
177    /// </param>
178    /// <param name="second">
179    /// <para>The second.</para>
180    /// <para></para>
181    /// </param>
182    /// <returns>
183    /// <para>The bool</para>
184    /// <para></para>
185    /// </returns>
186    protected override bool FirstIsToTheRightOfSecond(TElement first, TElement second) =>
187        ↪ Comparer.Compare(first, second) > 0;
188
189    /// <summary>
190    /// <para>
191    /// Gets the left reference using the specified node.
192    /// </para>
193    /// <para></para>
194    /// </summary>
195    /// <param name="node">
196    /// <para>The node.</para>
197    /// <para></para>
198    /// </param>
199    /// <returns>
200    /// <para>The ref element</para>
201    /// <para></para>
202    /// </returns>
203    protected override ref TElement GetLeftReference(TElement node) => ref
204        ↪ GetElement(node).Left;
205
206    /// <summary>
207    /// <para>
208    /// Gets the left using the specified node.
209    /// </para>
210    /// <para></para>
211    /// </summary>

```



```

208    /// <param name="node">
209    /// <para>The node.</para>
210    /// <para></para>
211    /// </param>
212    /// <returns>
213    /// <para>The element</para>
214    /// <para></para>
215    /// </returns>
216    protected override TElement GetLeft(TElement node) => GetElement(node).Left;
217
218    /// <summary>
219    /// <para>
220    /// Gets the right reference using the specified node.
221    /// </para>
222    /// <para></para>
223    /// </summary>
224    /// <param name="node">
225    /// <para>The node.</para>
226    /// <para></para>
227    /// </param>
228    /// <returns>
229    /// <para>The ref element</para>
230    /// <para></para>
231    /// </returns>
232    protected override ref TElement GetRightReference(TElement node) => ref
    ↪ GetElement(node).Right;
233
234    /// <summary>
235    /// <para>
236    /// Gets the right 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    /// <returns>
245    /// <para>The element</para>
246    /// <para></para>
247    /// </returns>
248    protected override TElement GetRight(TElement node) => GetElement(node).Right;
249
250    /// <summary>
251    /// <para>
252    /// Gets the size using the specified node.
253    /// </para>
254    /// <para></para>
255    /// </summary>
256    /// <param name="node">
257    /// <para>The node.</para>
258    /// <para></para>
259    /// </param>
260    /// <returns>
261    /// <para>The element</para>
262    /// <para></para>
263    /// </returns>
264    protected override TElement GetSize(TElement node) => GetElement(node).Size;
265
266    /// <summary>
267    /// <para>
268    /// Prints the node value using the specified node.
269    /// </para>
270    /// <para></para>
271    /// </summary>
272    /// <param name="node">
273    /// <para>The node.</para>
274    /// <para></para>
275    /// </param>
276    /// <param name="sb">
277    /// <para>The sb.</para>
278    /// <para></para>
279    /// </param>
280    protected override void PrintNodeValue(TElement node, StringBuilder sb) =>
    ↪ sb.Append(node);
281
282    /// <summary>
283    /// <para>

```

```

284     /// Sets the left using the specified node.
285     /// </para>
286     /// <para></para>
287     /// </summary>
288     /// <param name="node">
289     /// <para>The node.</para>
290     /// <para></para>
291     /// </param>
292     /// <param name="left">
293     /// <para>The left.</para>
294     /// <para></para>
295     /// </param>
296     protected override void SetLeft(TElement node, TElement left) => GetElement(node).Left =
        ↳ left;

297
298     /// <summary>
299     /// <para>
300     /// Sets the right using the specified node.
301     /// </para>
302     /// <para></para>
303     /// </summary>
304     /// <param name="node">
305     /// <para>The node.</para>
306     /// <para></para>
307     /// </param>
308     /// <param name="right">
309     /// <para>The right.</para>
310     /// <para></para>
311     /// </param>
312     protected override void SetRight(TElement node, TElement right) =>
        ↳ GetElement(node).Right = right;

313
314     /// <summary>
315     /// <para>
316     /// Sets the size using the specified node.
317     /// </para>
318     /// <para></para>
319     /// </summary>
320     /// <param name="node">
321     /// <para>The node.</para>
322     /// <para></para>
323     /// </param>
324     /// <param name="size">
325     /// <para>The size.</para>
326     /// <para></para>
327     /// </param>
328     protected override void SetSize(TElement node, TElement size) => GetElement(node).Size =
        ↳ size;

329
330     private ref TreeElement GetElement(TElement node) => ref
        ↳ _elements[UncheckedConverter<TElement, long>.Default.Convert(node)];

331 }
332 }

```

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         private struct TreeElement
20         {
21             /// <summary>
22             /// <para>
23             /// The size.
24             /// </para>

```

```

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

```

```

103     /// <summary>
104     /// <para>
105     /// Frees the node.
106     /// </para>
107     /// <para></para>
108     /// </summary>
109     /// <param name="node">
110     /// <para>The node.</para>
111     /// <para></para>
112     /// </param>
113     public void Free(TElement node)
114     {
115         while (!EqualityComparer.Equals(_allocated, One) && IsEmpty(node))
116         {
117             var lastNode = Arithmetic.Decrement(_allocated);
118             if (EqualityComparer.Equals(lastNode, node))
119             {
120                 _allocated = lastNode;
121                 node = Arithmetic.Decrement(node);
122             }
123             else
124             {
125                 return;
126             }
127         }
128     }
129
130     /// <summary>
131     /// <para>
132     /// Determines whether this instance is empty.
133     /// </para>
134     /// <para></para>
135     /// </summary>
136     /// <param name="node">
137     /// <para>The node.</para>
138     /// <para></para>
139     /// </param>
140     /// <returns>
141     /// <para>The bool</para>
142     /// <para></para>
143     /// </returns>
144     public bool IsEmpty(TElement node) =>
145     ↪ EqualityComparer<TreeElement>.Default.Equals(GetElement(node), default);
146
147     /// <summary>
148     /// <para>
149     /// Determines whether this instance first is to the left of second.
150     /// </para>
151     /// <para></para>
152     /// </summary>
153     /// <param name="first">
154     /// <para>The first.</para>
155     /// <para></para>
156     /// </param>
157     /// <param name="second">
158     /// <para>The second.</para>
159     /// <para></para>
160     /// </param>
161     /// <returns>
162     /// <para>The bool</para>
163     /// <para></para>
164     /// </returns>
165     protected override bool FirstIsToLeftOfSecond(TElement first, TElement second) =>
166     ↪ Comparer.Compare(first, second) < 0;
167
168     /// <summary>
169     /// <para>
170     /// Determines whether this instance first is to the right of second.
171     /// </para>
172     /// <para></para>
173     /// </summary>
174     /// <param name="first">
175     /// <para>The first.</para>
176     /// <para></para>
177     /// </param>
178     /// <param name="second">
179     /// <para>The second.</para>
180     /// <para></para>
181     /// </param>

```

```

179    /// </param>
180    /// <returns>
181    /// <para>The bool</para>
182    /// <para></para>
183    /// </returns>
184    protected override bool FirstIsToTheRightOfSecond(TElement first, TElement second) =>
185        ↪ Comparer.Compare(first, second) > 0;
186
187    /// <summary>
188    /// <para>
189    /// Gets the left reference using the specified node.
190    /// </para>
191    /// <para></para>
192    /// </summary>
193    /// <param name="node">
194    /// <para>The node.</para>
195    /// <para></para>
196    /// </param>
197    /// <returns>
198    /// <para>The ref element</para>
199    /// <para></para>
200    /// </returns>
201    protected override ref TElement GetLeftReference(TElement node) => ref
202        ↪ GetElement(node).Left;
203
204    /// <summary>
205    /// <para>
206    /// Gets the left using the specified node.
207    /// </para>
208    /// <para></para>
209    /// </summary>
210    /// <param name="node">
211    /// <para>The node.</para>
212    /// <para></para>
213    /// </param>
214    /// <returns>
215    /// <para>The element</para>
216    /// <para></para>
217    /// </returns>
218    protected override TElement GetLeft(TElement node) => GetElement(node).Left;
219
220    /// <summary>
221    /// <para>
222    /// Gets the right reference using the specified node.
223    /// </para>
224    /// <para></para>
225    /// </summary>
226    /// <param name="node">
227    /// <para>The node.</para>
228    /// <para></para>
229    /// </param>
230    /// <returns>
231    /// <para>The ref element</para>
232    /// <para></para>
233    /// </returns>
234    protected override ref TElement GetRightReference(TElement node) => ref
235        ↪ GetElement(node).Right;
236
237    /// <summary>
238    /// <para>
239    /// Gets the right using the specified node.
240    /// </para>
241    /// <para></para>
242    /// </summary>
243    /// <param name="node">
244    /// <para>The node.</para>
245    /// <para></para>
246    /// </param>
247    /// <returns>
248    /// <para>The element</para>
249    /// <para></para>
250    /// </returns>
251    protected override TElement GetRight(TElement node) => GetElement(node).Right;
252
253    /// <summary>
254    /// <para>
255    /// Gets the size using the specified node.
256    /// </para>

```

```

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

```

```

328         protected override void SetSize(TElement node, TElement size) => GetElement(node).Size =
           ↪ size;
329
330         private ref TreeElement GetElement(TElement node) => ref
           ↪ _elements[UncheckedConverter<TElement, long>.Default.Convert(node)];
331     }
332 }

```

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         private struct TreeElement
20         {
21             /// <summary>
22             /// <para>
23             /// The size.
24             /// </para>
25             /// <para></para>
26             /// </summary>
27             public TElement Size;
28             /// <summary>
29             /// <para>
30             /// The left.
31             /// </para>
32             /// <para></para>
33             /// </summary>
34             public TElement Left;
35             /// <summary>
36             /// <para>
37             /// The right.
38             /// </para>
39             /// <para></para>
40             /// </summary>
41             public TElement Right;
42             /// <summary>
43             /// <para>
44             /// The balance.
45             /// </para>
46             /// <para></para>
47             /// </summary>
48             public sbyte Balance;
49             /// <summary>
50             /// <para>
51             /// The left is child.
52             /// </para>
53             /// <para></para>
54             /// </summary>
55             public bool LeftIsChild;
56             /// <summary>
57             /// <para>
58             /// The right is child.
59             /// </para>
60             /// <para></para>
61             /// </summary>
62             public bool RightIsChild;
63         }
64
65         private readonly TreeElement[] _elements;
66         private TElement _allocated;
67
68         /// <summary>
69         /// <para>

```

```

70     /// The root.
71     /// </para>
72     /// <para></para>
73     /// </summary>
74     public TElement Root;
75
76     /// <summary>
77     /// <para>
78     /// Gets the count value.
79     /// </para>
80     /// <para></para>
81     /// </summary>
82     public TElement Count => GetSizeOrZero(Root);
83
84     /// <summary>
85     /// <para>
86     /// Initializes a new <see cref="SizedAndThreadedAVLBalancedTree"/> instance.
87     /// </para>
88     /// <para></para>
89     /// </summary>
90     /// <param name="capacity">
91     /// <para>A capacity.</para>
92     /// <para></para>
93     /// </param>
94     public SizedAndThreadedAVLBalancedTree(int capacity) => (_elements, _allocated) = (new
    ↪ TreeElement[capacity], One);
95
96     /// <summary>
97     /// <para>
98     /// Allocates this instance.
99     /// </para>
100    /// <para></para>
101    /// </summary>
102    /// <exception cref="InvalidOperationException">
103    /// <para>Allocated tree element is not empty.</para>
104    /// <para></para>
105    /// </exception>
106    /// <returns>
107    /// <para>The element</para>
108    /// <para></para>
109    /// </returns>
110    public TElement Allocate()
111    {
112        var newNode = _allocated;
113        if (IsEmpty(newNode))
114        {
115            _allocated = Arithmetic.Increment(_allocated);
116            return newNode;
117        }
118        else
119        {
120            throw new InvalidOperationException("Allocated tree element is not empty.");
121        }
122    }
123
124    /// <summary>
125    /// <para>
126    /// Frees the node.
127    /// </para>
128    /// <para></para>
129    /// </summary>
130    /// <param name="node">
131    /// <para>The node.</para>
132    /// <para></para>
133    /// </param>
134    public void Free(TElement node)
135    {
136        while (!EqualityComparer.Equals(_allocated, One) && IsEmpty(node))
137        {
138            var lastNode = Arithmetic.Decrement(_allocated);
139            if (EqualityComparer.Equals(lastNode, node))
140            {
141                _allocated = lastNode;
142                node = Arithmetic.Decrement(node);
143            }
144            else
145            {
146                return;

```



```

147     }
148 }
149 }
150
151 /// <summary>
152 /// <para>
153 /// Determines whether this instance is empty.
154 /// </para>
155 /// <para></para>
156 /// </summary>
157 /// <param name="node">
158 /// <para>The node.</para>
159 /// <para></para>
160 /// </param>
161 /// <returns>
162 /// <para>The bool</para>
163 /// <para></para>
164 /// </returns>
165 public bool IsEmpty(TElement node) =>
166     ↪ EqualityComparer<TreeElement>.Default.Equals(GetElement(node), default);
167
168 /// <summary>
169 /// <para>
170 /// Determines whether this instance first is to the left of second.
171 /// </para>
172 /// <para></para>
173 /// </summary>
174 /// <param name="first">
175 /// <para>The first.</para>
176 /// <para></para>
177 /// </param>
178 /// <param name="second">
179 /// <para>The second.</para>
180 /// <para></para>
181 /// </param>
182 /// <returns>
183 /// <para>The bool</para>
184 /// <para></para>
185 /// </returns>
186 protected override bool FirstIsToLeftOfSecond(TElement first, TElement second) =>
187     ↪ Comparer.Compare(first, second) < 0;
188
189 /// <summary>
190 /// <para>
191 /// Determines whether this instance first is to the right of second.
192 /// </para>
193 /// <para></para>
194 /// </summary>
195 /// <param name="first">
196 /// <para>The first.</para>
197 /// <para></para>
198 /// </param>
199 /// <param name="second">
200 /// <para>The second.</para>
201 /// <para></para>
202 /// </param>
203 /// <returns>
204 /// <para>The bool</para>
205 /// <para></para>
206 /// </returns>
207 protected override bool FirstIsToTheRightOfSecond(TElement first, TElement second) =>
208     ↪ Comparer.Compare(first, second) > 0;
209
210 /// <summary>
211 /// <para>
212 /// Gets the balance using the specified node.
213 /// </para>
214 /// <para></para>
215 /// </summary>
216 /// <param name="node">
217 /// <para>The node.</para>
218 /// <para></para>
219 /// </param>
220 /// <returns>
221 /// <para>The sbyte</para>
222 /// <para></para>
223 /// </returns>
224 protected override sbyte GetBalance(TElement node) => GetElement(node).Balance;

```

```

222     /// <summary>
223     /// <para>
224     /// Determines whether this instance get left is child.
225     /// </para>
226     /// <para></para>
227     /// </summary>
228     /// <param name="node">
229     /// <para>The node.</para>
230     /// <para></para>
231     /// </param>
232     /// <returns>
233     /// <para>The bool</para>
234     /// <para></para>
235     /// </returns>
236     protected override bool GetLeftIsChild(TElement node) => GetElement(node).LeftIsChild;
237
238     /// <summary>
239     /// <para>
240     /// Gets the left reference using the specified node.
241     /// </para>
242     /// <para></para>
243     /// </summary>
244     /// <param name="node">
245     /// <para>The node.</para>
246     /// <para></para>
247     /// </param>
248     /// <returns>
249     /// <para>The ref element</para>
250     /// <para></para>
251     /// </returns>
252     protected override ref TElement GetLeftReference(TElement node) => ref
253     ↪ GetElement(node).Left;
254
255     /// <summary>
256     /// <para>
257     /// Gets the left using the specified node.
258     /// </para>
259     /// <para></para>
260     /// </summary>
261     /// <param name="node">
262     /// <para>The node.</para>
263     /// <para></para>
264     /// </param>
265     /// <returns>
266     /// <para>The element</para>
267     /// <para></para>
268     /// </returns>
269     protected override TElement GetLeft(TElement node) => GetElement(node).Left;
270
271     /// <summary>
272     /// <para>
273     /// Determines whether this instance get right is child.
274     /// </para>
275     /// <para></para>
276     /// </summary>
277     /// <param name="node">
278     /// <para>The node.</para>
279     /// <para></para>
280     /// </param>
281     /// <returns>
282     /// <para>The bool</para>
283     /// <para></para>
284     /// </returns>
285     protected override bool GetRightIsChild(TElement node) => GetElement(node).RightIsChild;
286
287     /// <summary>
288     /// <para>
289     /// Gets the right reference using the specified node.
290     /// </para>
291     /// <para></para>
292     /// </summary>
293     /// <param name="node">
294     /// <para>The node.</para>
295     /// <para></para>
296     /// </param>
297     /// <returns>
298     /// <para>The ref element</para>

```

```

299     /// <para></para>
300     /// </returns>
301     protected override ref TElement GetRightReference(TElement node) => ref
        ↪ GetElement(node).Right;

302
303     /// <summary>
304     /// <para>
305     /// Gets the right using the specified node.
306     /// </para>
307     /// <para></para>
308     /// </summary>
309     /// <param name="node">
310     /// <para>The node.</para>
311     /// <para></para>
312     /// </param>
313     /// <returns>
314     /// <para>The element</para>
315     /// <para></para>
316     /// </returns>
317     protected override TElement GetRight(TElement node) => GetElement(node).Right;
318
319     /// <summary>
320     /// <para>
321     /// Gets the size using the specified node.
322     /// </para>
323     /// <para></para>
324     /// </summary>
325     /// <param name="node">
326     /// <para>The node.</para>
327     /// <para></para>
328     /// </param>
329     /// <returns>
330     /// <para>The element</para>
331     /// <para></para>
332     /// </returns>
333     protected override TElement GetSize(TElement node) => GetElement(node).Size;
334
335     /// <summary>
336     /// <para>
337     /// Prints the node value using the specified node.
338     /// </para>
339     /// <para></para>
340     /// </summary>
341     /// <param name="node">
342     /// <para>The node.</para>
343     /// <para></para>
344     /// </param>
345     /// <param name="sb">
346     /// <para>The sb.</para>
347     /// <para></para>
348     /// </param>
349     protected override void PrintNodeValue(TElement node, StringBuilder sb) =>
        ↪ sb.Append(node);

350
351     /// <summary>
352     /// <para>
353     /// Sets the balance using the specified node.
354     /// </para>
355     /// <para></para>
356     /// </summary>
357     /// <param name="node">
358     /// <para>The node.</para>
359     /// <para></para>
360     /// </param>
361     /// <param name="value">
362     /// <para>The value.</para>
363     /// <para></para>
364     /// </param>
365     protected override void SetBalance(TElement node, sbyte value) =>
        ↪ GetElement(node).Balance = value;

366
367     /// <summary>
368     /// <para>
369     /// Sets the left using the specified node.
370     /// </para>
371     /// <para></para>
372     /// </summary>
373     /// <param name="node">

```

```

374    /// <para>The node.</para>
375    /// <para></para>
376    /// </param>
377    /// <param name="left">
378    /// <para>The left.</para>
379    /// <para></para>
380    /// </param>
381    protected override void SetLeft(TElement node, TElement left) => GetElement(node).Left =
    ↪ left;
382
383    /// <summary>
384    /// <para>
385    /// Sets the left is child using the specified node.
386    /// </para>
387    /// <para></para>
388    /// </summary>
389    /// <param name="node">
390    /// <para>The node.</para>
391    /// <para></para>
392    /// </param>
393    /// <param name="value">
394    /// <para>The value.</para>
395    /// <para></para>
396    /// </param>
397    protected override void SetLeftIsChild(TElement node, bool value) =>
    ↪ GetElement(node).LeftIsChild = value;
398
399    /// <summary>
400    /// <para>
401    /// Sets the right using the specified node.
402    /// </para>
403    /// <para></para>
404    /// </summary>
405    /// <param name="node">
406    /// <para>The node.</para>
407    /// <para></para>
408    /// </param>
409    /// <param name="right">
410    /// <para>The right.</para>
411    /// <para></para>
412    /// </param>
413    protected override void SetRight(TElement node, TElement right) =>
    ↪ GetElement(node).Right = right;
414
415    /// <summary>
416    /// <para>
417    /// Sets the right is child using the specified node.
418    /// </para>
419    /// <para></para>
420    /// </summary>
421    /// <param name="node">
422    /// <para>The node.</para>
423    /// <para></para>
424    /// </param>
425    /// <param name="value">
426    /// <para>The value.</para>
427    /// <para></para>
428    /// </param>
429    protected override void SetRightIsChild(TElement node, bool value) =>
    ↪ GetElement(node).RightIsChild = value;
430
431    /// <summary>
432    /// <para>
433    /// Sets the size using the specified node.
434    /// </para>
435    /// <para></para>
436    /// </summary>
437    /// <param name="node">
438    /// <para>The node.</para>
439    /// <para></para>
440    /// </param>
441    /// <param name="size">
442    /// <para>The size.</para>
443    /// <para></para>
444    /// </param>
445    protected override void SetSize(TElement node, TElement size) => GetElement(node).Size =
    ↪ size;
446

```

```

447         private ref TreeElement GetElement(TElement node) => ref
448             ↪ _elements[UncheckedConverter<TElement, long>.Default.Convert(node)];
449     }

```

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
52             ↪ SizedBinaryTreeMethodsBase<TElement> tree, Func<TElement> allocate, Action<TElement>
53             ↪ free, ref TElement root, Func<TElement> treeCount, int maximumOperationsPerCycle)
54         {
55             for (var N = 1; N < maximumOperationsPerCycle; N++)
56             {
57                 var currentCount = 0;
58                 for (var i = 0; i < N; i++)
59                 {
60                     var node = allocate();
61                     tree.Attach(ref root, node);
62                     currentCount++;
63                     Assert.Equal(currentCount, (int)UncheckedConverter<TElement,
64                         ↪ int>.Default.Convert(treeCount()));
65                 }
66                 for (var i = 1; i <= N; i++)
67                 {
68                     TElement node = UncheckedConverter<int, TElement>.Default.Convert(i);
69                     if (tree.Contains(node, root))
70                     {
71                         tree.Detach(ref root, node);
72                         free(node);
73                     }
74                 }
75             }
76         }
77     }

```

```

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     private const int _n = 500;
14
15     /// <summary>
16     /// <para>
17     /// Tests that recursionless size balanced tree multiple attach and detach test.
18     /// </para>
19     /// <para></para>
20     /// </summary>
21     [Fact]
22     public static void RecursionlessSizeBalancedTreeMultipleAttachAndDetachTest()
23     {
24         var recursionlessSizeBalancedTree = new RecursionlessSizeBalancedTree<uint>(10000);
25         recursionlessSizeBalancedTree.TestMultipleCreationsAndDeletions(recursionlessSizeBal
            ↳ ancedTree.Allocate, recursionlessSizeBalancedTree.Free, ref
            ↳ recursionlessSizeBalancedTree.Root, () => recursionlessSizeBalancedTree.Count,
            ↳ _n);
26     }
27
28     /// <summary>
29     /// <para>
30     /// Tests that size balanced tree multiple attach and detach test.
31     /// </para>
32     /// <para></para>
33     /// </summary>
34     [Fact]
35     public static void SizeBalancedTreeMultipleAttachAndDetachTest()
36     {
37         var sizeBalancedTree = new SizeBalancedTree<uint>(10000);
38         sizeBalancedTree.TestMultipleCreationsAndDeletions(sizeBalancedTree.Allocate,
            ↳ sizeBalancedTree.Free, ref sizeBalancedTree.Root, () => sizeBalancedTree.Count,
            ↳ _n);
39     }
40
41     /// <summary>
42     /// <para>
43     /// Tests that sized and threaded avl balanced tree multiple attach and detach test.
44     /// </para>
45     /// <para></para>
46     /// </summary>
47     [Fact]
48     public static void SizedAndThreadedAVLBalancedTreeMultipleAttachAndDetachTest()
49     {
50         var avlTree = new SizedAndThreadedAVLBalancedTree<uint>(10000);
51         avlTree.TestMultipleCreationsAndDeletions(avlTree.Allocate, avlTree.Free, ref
            ↳ avlTree.Root, () => avlTree.Count, _n);
52     }
53
54     /// <summary>
55     /// <para>
56     /// Tests that recursionless size balanced tree multiple random attach and detach test.
57     /// </para>
58     /// <para></para>
59     /// </summary>
60     [Fact]
61     public static void RecursionlessSizeBalancedTreeMultipleRandomAttachAndDetachTest()
62     {
63         var recursionlessSizeBalancedTree = new RecursionlessSizeBalancedTree<uint>(10000);
64         recursionlessSizeBalancedTree.TestMultipleRandomCreationsAndDeletions(ref
            ↳ recursionlessSizeBalancedTree.Root, () => recursionlessSizeBalancedTree.Count,
            ↳ _n);
65     }
66
67     /// <summary>
68     /// <para>
69     /// Tests that size balanced tree multiple random attach and detach test.
70     /// </para>
71     /// <para></para>
72     /// </summary>
73     [Fact]
74     public static void SizeBalancedTreeMultipleRandomAttachAndDetachTest()

```

```

75     {
76         var sizeBalancedTree = new SizeBalancedTree<uint>(10000);
77         sizeBalancedTree.TestMultipleRandomCreationsAndDeletions(ref sizeBalancedTree.Root,
78             ↪ () => sizeBalancedTree.Count, _n);
79     }
80     /// <summary>
81     /// <para>
82     /// Tests that sized and threaded avl balanced tree multiple random attach and detach
83     ↪ test.
84     /// </para>
85     /// <para></para>
86     /// </summary>
87     [Fact]
88     public static void SizedAndThreadedAVLBalancedTreeMultipleRandomAttachAndDetachTest()
89     {
90         var avlTree = new SizedAndThreadedAVLBalancedTree<uint>(10000);
91         avlTree.TestMultipleRandomCreationsAndDeletions(ref avlTree.Root, () =>
92             ↪ avlTree.Count, _n);
93     }
94 }

```


Index

- ./csharp/Platform.Collections.Methods.Tests/RecursionlessSizeBalancedTree.cs, 46
- ./csharp/Platform.Collections.Methods.Tests/SizeBalancedTree.cs, 50
- ./csharp/Platform.Collections.Methods.Tests/SizedAndThreadedAVLBalancedTree.cs, 55
- ./csharp/Platform.Collections.Methods.Tests/TestExtensions.cs, 61
- ./csharp/Platform.Collections.Methods.Tests/TreesTests.cs, 62
- ./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, 22
- ./csharp/Platform.Collections.Methods/Trees/SizedBinaryTreeMethodsBase.cs, 34