

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)));

```

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 /// <summary>
132 /// <para>
133 /// Detaches the core using the specified root.
134 /// </para>
135 /// <para></para>
136 /// </summary>
137 /// <param name="root">
138 /// <para>The root.</para>
139 /// <para></para>
140 /// </param>
141 /// <param name="node">
142 /// <para>The node.</para>
143 /// <para></para>
144 /// </param>
145 protected override void DetachCore(ref TElement root, TElement node)
146 {

```

```

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 = GetRightest(left);
200                 DetachCore(ref left, replacement);
201             }
202             else
203             {
204                 replacement = GetLeftest(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         /// <summary>
54         /// <para>
55         /// Detaches the core using the specified root.
56         /// </para>
57         /// <para></para>
58         /// </summary>
59         /// <param name="root">
60         /// <para>The root.</para>
61         /// <para></para>
62         /// </param>
63         /// <param name="nodeToDetach">
64         /// <para>The node to detach.</para>
65         /// <para></para>
66         /// </param>
67         /// <exception cref="InvalidOperationException">
68         /// <para>Duplicate link found in the tree.</para>
69         /// <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)))

```

```

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

```

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
    ↳ enerated/AVLTreeList.cs">TreeLib.AVLTreeList</a>.
18  /// Which itself based on: <a
    ↳ href="https://github.com/GNOME/glib/blob/master/glib/gtree.c">GNOME/glib/gtree</a>.
19  /// </remarks>
20  public abstract class SizedAndThreadedAVLBalancedTreeMethods<TElement> :
    ↳ SizedBinaryTreeMethodsBase<TElement>
21  {
22      private static readonly int _maxPath = 11 * NumericType<TElement>.BytesSize + 4;
23
24      /// <summary>
25      /// <para>
26      /// Gets the rightest using the specified current.
27      /// </para>
28      /// <para></para>
29      /// </summary>
30      /// <param name="current">
31      /// <para>The current.</para>
32      /// <para></para>
33      /// </param>
34      /// <returns>
35      /// <para>The current.</para>
36      /// <para></para>
37      /// </returns>
38      [MethodImpl(MethodImplOptions.AggressiveInlining)]
39      protected override TElement GetRightest(TElement current)
40      {
41          var currentRight = GetRightOrDefault(current);
42          while (!EqualToZero(currentRight))
43          {
44              current = currentRight;
45              currentRight = GetRightOrDefault(current);
46          }
47          return current;
48      }
49
50      /// <summary>
51      /// <para>
52      /// Gets the leftest using the specified current.
53      /// </para>
54      /// <para></para>
55      /// </summary>
56      /// <param name="current">
57      /// <para>The current.</para>
58      /// <para></para>
59      /// </param>
60      /// <returns>
61      /// <para>The current.</para>
62      /// <para></para>
63      /// </returns>
64      [MethodImpl(MethodImplOptions.AggressiveInlining)]
65      protected override TElement GetLeftest(TElement current)
66      {
67          var currentLeft = GetLeftOrDefault(current);
68          while (!EqualToZero(currentLeft))
69          {
70              current = currentLeft;
71              currentLeft = GetLeftOrDefault(current);
72          }
73          return current;
74      }
75
76      /// <summary>
77      /// <para>
78      /// Determines whether this instance contains.
79      /// </para>
80      /// <para></para>
81      /// </summary>
82      /// <param name="node">
83      /// <para>The node.</para>
84      /// <para></para>
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">
163     /// <para>The node.</para>
164     /// <para></para>
165     /// </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             {
393                 var parent = path[--pathPosition];
394                 var isLeftNode = !AreEqual(parent, default) && AreEqual(currentNode,
395                 → 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 private TElement Balance(TElement node)
433 {
434     unchecked
435     {
436         var rootBalance = GetBalance(node);
437         if (rootBalance < -1)
438         {
439             var left = GetLeft(node);
440             if (GetBalance(left) > 0)
441             {
442                 SetLeft(node, LeftRotateWithBalance(left));
443                 FixSize(node);
444             }
445             node = RightRotateWithBalance(node);
446         }
447         else if (rootBalance > 1)
448         {
449             var right = GetRight(node);
450             if (GetBalance(right) < 0)
451             {
452                 SetRight(node, RightRotateWithBalance(right));
453                 FixSize(node);
454             }
455             node = LeftRotateWithBalance(node);
456         }
457         return node;
458     }
459 }
460
461 /// <summary>
462 /// <para>
463 /// Lefts the rotate with balance using the specified node.
464 /// </para>
465 /// <para></para>
466 /// </summary>
467 /// <param name="node">
468 /// <para>The node.</para>
469 /// <para></para>
470 /// </param>

```

```

471     /// <returns>
472     /// <para>The element</para>
473     /// <para></para>
474     /// </returns>
475     protected TElement LeftRotateWithBalance(TElement node)
476     {
477         unchecked
478         {
479             var right = GetRight(node);
480             if (GetLeftIsChild(right))
481             {
482                 SetRight(node, GetLeft(right));
483             }
484             else
485             {
486                 SetRightIsChild(node, false);
487                 SetLeftIsChild(right, true);
488             }
489             SetLeft(right, node);
490             // Fix size
491             SetSize(right, GetSize(node));
492             FixSize(node);
493             // Fix balance
494             var rootBalance = GetBalance(node);
495             var rightBalance = GetBalance(right);
496             if (rightBalance <= 0)
497             {
498                 if (rootBalance >= 1)
499                 {
500                     SetBalance(right, (sbyte)(rightBalance - 1));
501                 }
502                 else
503                 {
504                     SetBalance(right, (sbyte)(rootBalance + rightBalance - 2));
505                 }
506                 SetBalance(node, (sbyte)(rootBalance - 1));
507             }
508             else
509             {
510                 if (rootBalance <= rightBalance)
511                 {
512                     SetBalance(right, (sbyte)(rootBalance - 2));
513                 }
514                 else
515                 {
516                     SetBalance(right, (sbyte)(rightBalance - 1));
517                 }
518                 SetBalance(node, (sbyte)(rootBalance - rightBalance - 1));
519             }
520             return right;
521         }
522     }
523
524     /// <summary>
525     /// <para>
526     /// Rights the rotate with balance using the specified node.
527     /// </para>
528     /// <para></para>
529     /// </summary>
530     /// <param name="node">
531     /// <para>The node.</para>
532     /// <para></para>
533     /// </param>
534     /// <returns>
535     /// <para>The element</para>
536     /// <para></para>
537     /// </returns>
538     protected TElement RightRotateWithBalance(TElement node)
539     {
540         unchecked
541         {
542             var left = GetLeft(node);
543             if (GetRightIsChild(left))
544             {
545                 SetLeft(node, GetRight(left));
546             }
547             else
548             {

```

```

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

```

```

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

```

```

704     if (!GetRightIsChild(currentNode)) // node has no children
705     {
706         if (AreEqual(parent, default))
707         {
708             root = Zero;
709         }
710         else if (isLeftNode)
711         {
712             SetLeftIsChild(parent, false);
713             SetLeft(parent, GetLeft(currentNode));
714             IncrementBalance(parent);
715         }
716         else
717         {
718             SetRightIsChild(parent, false);
719             SetRight(parent, GetRight(currentNode));
720             DecrementBalance(parent);
721         }
722     }
723     else // node has a right child
724     {
725         var successor = GetNext(currentNode);
726         SetLeft(successor, GetLeft(currentNode));
727         var right = GetRight(currentNode);
728         if (AreEqual(parent, default))
729         {
730             root = right;
731         }
732         else if (isLeftNode)
733         {
734             SetLeft(parent, right);
735             IncrementBalance(parent);
736         }
737         else
738         {
739             SetRight(parent, right);
740             DecrementBalance(parent);
741         }
742     }
743 }
744 else // node has a left child
745 {
746     if (!GetRightIsChild(currentNode))
747     {
748         var predecessor = GetPrevious(currentNode);
749         SetRight(predecessor, GetRight(currentNode));
750         var leftValue = GetLeft(currentNode);
751         if (AreEqual(parent, default))
752         {
753             root = leftValue;
754         }
755         else if (isLeftNode)
756         {
757             SetLeft(parent, leftValue);
758             IncrementBalance(parent);
759         }
760         else
761         {
762             SetRight(parent, leftValue);
763             DecrementBalance(parent);
764         }
765     }
766     else // node has a both children (left and right)
767     {
768         var predecessor = GetLeft(currentNode);
769         var successor = GetRight(currentNode);
770         var successorParent = currentNode;
771         int previousPathPosition = ++pathPosition;
772         // find the immediately next node (and its parent)
773         while (GetLeftIsChild(successor))
774         {
775             path[++pathPosition] = successorParent = successor;
776             successor = GetLeft(successor);
777             if (!AreEqual(successorParent, currentNode))
778             {
779                 DecrementSize(successorParent);
780             }
781         }

```



```

782 path[previousPathPosition] = successor;
783 balanceNode = path[pathPosition];
784 // remove 'successor' from the tree
785 if (!AreEqual(successorParent, currentNode))
786 {
787     if (!GetRightIsChild(successor))
788     {
789         SetLeftIsChild(successorParent, false);
790     }
791     else
792     {
793         SetLeft(successorParent, GetRight(successor));
794     }
795     IncrementBalance(successorParent);
796     SetRightIsChild(successor, true);
797     SetRight(successor, GetRight(currentNode));
798 }
799 else
800 {
801     DecrementBalance(currentNode);
802 }
803 // set the predecessor's successor link to point to the right place
804 while (GetRightIsChild(predecessor))
805 {
806     predecessor = GetRight(predecessor);
807 }
808 SetRight(predecessor, successor);
809 // prepare 'successor' to replace 'node'
810 var left = GetLeft(currentNode);
811 SetLeftIsChild(successor, true);
812 SetLeft(successor, left);
813 SetBalance(successor, GetBalance(currentNode));
814 FixSize(successor);
815 if (AreEqual(parent, default))
816 {
817     root = successor;
818 }
819 else if (isLeftNode)
820 {
821     SetLeft(parent, successor);
822 }
823 else
824 {
825     SetRight(parent, successor);
826 }
827 }
828 }
829 // restore balance
830 if (!AreEqual(balanceNode, default))
831 {
832     while (true)
833     {
834         var balanceParent = path[--pathPosition];
835         isLeftNode = !AreEqual(balanceParent, default) && AreEqual(balanceNode,
            ↪ GetLeft(balanceParent));
836         var currentNodeBalance = GetBalance(balanceNode);
837         if (currentNodeBalance < -1 || currentNodeBalance > 1)
838         {
839             balanceNode = Balance(balanceNode);
840             if (AreEqual(balanceParent, default))
841             {
842                 root = balanceNode;
843             }
844             else if (isLeftNode)
845             {
846                 SetLeft(balanceParent, balanceNode);
847             }
848             else
849             {
850                 SetRight(balanceParent, balanceNode);
851             }
852         }
853         currentNodeBalance = GetBalance(balanceNode);
854         if (currentNodeBalance != 0 || AreEqual(balanceParent, default))
855         {
856             break;
857         }
858         if (isLeftNode)

```

```

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

```

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

```

1  // #define ENABLE_TREE_AUTO_DEBUG_AND_VALIDATION
2
3  using System;
4  using System.Diagnostics;
5  using System.Runtime.CompilerServices;
6  using System.Text;
7  using Platform.Numbers;
8
9  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
10
11 namespace Platform.Collections.Methods.Trees
12 {
13     /// <summary>
14     /// <para>
15     /// Represents the sized binary tree methods base.
16     /// </para>
17     /// <para></para>
18     /// </summary>
19     /// <seealso cref="GenericCollectionMethodsBase{TElement}"/>
20     public abstract class SizedBinaryTreeMethodsBase<TElement> :
21         ↳ GenericCollectionMethodsBase<TElement>
22     {
23         /// <summary>
24         /// <para>
25         /// Gets the left reference using the specified node.
26         /// </para>
27         /// <para></para>
28         /// </summary>
29         /// <param name="node">
30         /// <para>The node.</para>
31         /// <para></para>
32         /// </param>
33         /// <returns>
34         /// <para>The ref element</para>
35         /// <para></para>
36         /// </returns>
37         [MethodImpl(MethodImplOptions.AggressiveInlining)]
38         protected abstract ref TElement GetLeftReference(TElement node);

```

```

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 FirstIsToLeftOfSecond(TElement first, TElement second);
178
179     /// <summary>
180     /// <para>
181     /// Determines whether this instance first is to the right of second.
182     /// </para>
183     /// <para></para>
184     /// </summary>
185     /// <param name="first">
186     /// <para>The first.</para>
187     /// <para></para>
188     /// </param>
189     /// <param name="second">
190     /// <para>The second.</para>
191     /// <para></para>
192     /// </param>
193     /// <returns>

```

```

194     /// <para>The bool</para>
195     /// <para></para>
196     /// </returns>
197     [MethodImpl(MethodImplOptions.AggressiveInlining)]
198     protected abstract bool FirstIsToTheRightOfSecond(TElement first, TElement second);
199
200     /// <summary>
201     /// <para>
202     /// Gets the left or default using the specified node.
203     /// </para>
204     /// <para></para>
205     /// </summary>
206     /// <param name="node">
207     /// <para>The node.</para>
208     /// <para></para>
209     /// </param>
210     /// <returns>
211     /// <para>The element</para>
212     /// <para></para>
213     /// </returns>
214     [MethodImpl(MethodImplOptions.AggressiveInlining)]
215     protected virtual TElement GetLeftOrDefault(TElement node) => AreEqual(node, default) ?
    ↪ default : GetLeft(node);
216
217     /// <summary>
218     /// <para>
219     /// Gets the right or default using the specified node.
220     /// </para>
221     /// <para></para>
222     /// </summary>
223     /// <param name="node">
224     /// <para>The node.</para>
225     /// <para></para>
226     /// </param>
227     /// <returns>
228     /// <para>The element</para>
229     /// <para></para>
230     /// </returns>
231     [MethodImpl(MethodImplOptions.AggressiveInlining)]
232     protected virtual TElement GetRightOrDefault(TElement node) => AreEqual(node, default) ?
    ↪ default : GetRight(node);
233
234     /// <summary>
235     /// <para>
236     /// Increments the size using the specified node.
237     /// </para>
238     /// <para></para>
239     /// </summary>
240     /// <param name="node">
241     /// <para>The node.</para>
242     /// <para></para>
243     /// </param>
244     [MethodImpl(MethodImplOptions.AggressiveInlining)]
245     protected void IncrementSize(TElement node) => SetSize(node, Increment(GetSize(node)));
246
247     /// <summary>
248     /// <para>
249     /// Decrements the size using the specified node.
250     /// </para>
251     /// <para></para>
252     /// </summary>
253     /// <param name="node">
254     /// <para>The node.</para>
255     /// <para></para>
256     /// </param>
257     [MethodImpl(MethodImplOptions.AggressiveInlining)]
258     protected void DecrementSize(TElement node) => SetSize(node, Decrement(GetSize(node)));
259
260     /// <summary>
261     /// <para>
262     /// Gets the left size using the specified node.
263     /// </para>
264     /// <para></para>
265     /// </summary>
266     /// <param name="node">
267     /// <para>The node.</para>
268     /// <para></para>
269     /// </param>

```

```

270     /// <returns>
271     /// <para>The element</para>
272     /// <para></para>
273     /// </returns>
274     [MethodImpl(MethodImplOptions.AggressiveInlining)]
275     protected TElement GetLeftSize(TElement node) => GetSizeOrZero(GetLeftOrDefault(node));
276
277     /// <summary>
278     /// <para>
279     /// Gets the right size using the specified node.
280     /// </para>
281     /// <para></para>
282     /// </summary>
283     /// <param name="node">
284     /// <para>The node.</para>
285     /// <para></para>
286     /// </param>
287     /// <returns>
288     /// <para>The element</para>
289     /// <para></para>
290     /// </returns>
291     [MethodImpl(MethodImplOptions.AggressiveInlining)]
292     protected TElement GetRightSize(TElement node) => GetSizeOrZero(GetRightOrDefault(node));
293
294     /// <summary>
295     /// <para>
296     /// Gets the size or zero using the specified node.
297     /// </para>
298     /// <para></para>
299     /// </summary>
300     /// <param name="node">
301     /// <para>The node.</para>
302     /// <para></para>
303     /// </param>
304     /// <returns>
305     /// <para>The element</para>
306     /// <para></para>
307     /// </returns>
308     [MethodImpl(MethodImplOptions.AggressiveInlining)]
309     protected TElement GetSizeOrZero(TElement node) => EqualToZero(node) ? Zero :
    ↪ GetSize(node);
310
311     /// <summary>
312     /// <para>
313     /// Fixes the size using the specified node.
314     /// </para>
315     /// <para></para>
316     /// </summary>
317     /// <param name="node">
318     /// <para>The node.</para>
319     /// <para></para>
320     /// </param>
321     [MethodImpl(MethodImplOptions.AggressiveInlining)]
322     protected void FixSize(TElement node) => SetSize(node, Increment(Add(GetLeftSize(node),
    ↪ GetRightSize(node))));
323
324     /// <summary>
325     /// <para>
326     /// Lefts the rotate using the specified root.
327     /// </para>
328     /// <para></para>
329     /// </summary>
330     /// <param name="root">
331     /// <para>The root.</para>
332     /// <para></para>
333     /// </param>
334     [MethodImpl(MethodImplOptions.AggressiveInlining)]
335     protected void LeftRotate(ref TElement root) => root = LeftRotate(root);
336
337     /// <summary>
338     /// <para>
339     /// Lefts the rotate using the specified root.
340     /// </para>
341     /// <para></para>
342     /// </summary>
343     /// <param name="root">
344     /// <para>The root.</para>
345     /// <para></para>

```

```

346     /// </param>
347     /// <returns>
348     /// <para>The right.</para>
349     /// <para></para>
350     /// </returns>
351     [MethodImpl(MethodImplOptions.AggressiveInlining)]
352     protected TElement LeftRotate(TElement root)
353     {
354         var right = GetRight(root);
355     #if ENABLE_TREE_AUTO_DEBUG_AND_VALIDATION
356         if (EqualToZero(right))
357         {
358             throw new InvalidOperationException("Right is null.");
359         }
360     #endif
361         SetRight(root, GetLeft(right));
362         SetLeft(right, root);
363         SetSize(right, GetSize(root));
364         FixSize(root);
365         return right;
366     }
367
368     /// <summary>
369     /// <para>
370     /// Rights the rotate using the specified root.
371     /// </para>
372     /// <para></para>
373     /// </summary>
374     /// <param name="root">
375     /// <para>The root.</para>
376     /// <para></para>
377     /// </param>
378     [MethodImpl(MethodImplOptions.AggressiveInlining)]
379     protected void RightRotate(ref TElement root) => root = RightRotate(root);
380
381     /// <summary>
382     /// <para>
383     /// Rights the rotate using the specified root.
384     /// </para>
385     /// <para></para>
386     /// </summary>
387     /// <param name="root">
388     /// <para>The root.</para>
389     /// <para></para>
390     /// </param>
391     /// <returns>
392     /// <para>The left.</para>
393     /// <para></para>
394     /// </returns>
395     [MethodImpl(MethodImplOptions.AggressiveInlining)]
396     protected TElement RightRotate(TElement root)
397     {
398         var left = GetLeft(root);
399     #if ENABLE_TREE_AUTO_DEBUG_AND_VALIDATION
400         if (EqualToZero(left))
401         {
402             throw new InvalidOperationException("Left is null.");
403         }
404     #endif
405         SetLeft(root, GetRight(left));
406         SetRight(left, root);
407         SetSize(left, GetSize(root));
408         FixSize(root);
409         return left;
410     }
411
412     /// <summary>
413     /// <para>
414     /// Gets the rightest using the specified current.
415     /// </para>
416     /// <para></para>
417     /// </summary>
418     /// <param name="current">
419     /// <para>The current.</para>
420     /// <para></para>
421     /// </param>
422     /// <returns>
423     /// <para>The current.</para>

```

```

424     /// <para></para>
425     /// </returns>
426     [MethodImpl(MethodImplOptions.AggressiveInlining)]
427     protected virtual TElement GetRightest(TElement current)
428     {
429         var currentRight = GetRight(current);
430         while (!EqualToZero(currentRight))
431         {
432             current = currentRight;
433             currentRight = GetRight(current);
434         }
435         return current;
436     }
437
438     /// <summary>
439     /// <para>
440     /// Gets the leftest using the specified current.
441     /// </para>
442     /// <para></para>
443     /// </summary>
444     /// <param name="current">
445     /// <para>The current.</para>
446     /// <para></para>
447     /// </param>
448     /// <returns>
449     /// <para>The current.</para>
450     /// <para></para>
451     /// </returns>
452     [MethodImpl(MethodImplOptions.AggressiveInlining)]
453     protected virtual TElement GetLeftest(TElement current)
454     {
455         var currentLeft = GetLeft(current);
456         while (!EqualToZero(currentLeft))
457         {
458             current = currentLeft;
459             currentLeft = GetLeft(current);
460         }
461         return current;
462     }
463
464     /// <summary>
465     /// <para>
466     /// Gets the next using the specified node.
467     /// </para>
468     /// <para></para>
469     /// </summary>
470     /// <param name="node">
471     /// <para>The node.</para>
472     /// <para></para>
473     /// </param>
474     /// <returns>
475     /// <para>The element</para>
476     /// <para></para>
477     /// </returns>
478     [MethodImpl(MethodImplOptions.AggressiveInlining)]
479     protected virtual TElement GetNext(TElement node) => GetLeftest(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) => GetRightest(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     /// </summary>
662     /// <param name="root">
663     /// <para>The root.</para>
664     /// </param>
665     /// <param name="node">
666     /// <para>The node.</para>
667     /// </param>
668     protected abstract void DetachCore(ref TElement root, TElement node);
669
670     /// <summary>
671     /// <para>
672     /// Fixes the sizes using the specified node.
673     /// </para>
674     /// </summary>
675     /// <param name="node">
676     /// <para>The node.</para>
677     /// </param>
678     public void FixSizes(TElement node)
679     {
680         if (AreEqual(node, default))
681         {
682             return;
683         }
684         FixSizes(GetLeft(node));
685         FixSizes(GetRight(node));
686         FixSize(node);
687     }
688
689     /// <summary>
690     /// <para>
691     /// Validates the sizes using the specified node.
692     /// </para>
693     /// </summary>
694     /// <param name="node">
695     /// <para>The node.</para>
696     /// </param>
697     /// <exception cref="InvalidOperationException">
698     /// <para>Size of {node} is not valid. Expected size: {expectedSize}, actual size:
699     /// ↪ {size}.</para>
700     /// </exception>
701     public void ValidateSizes(TElement node)
702     {
703         if (AreEqual(node, default))
704         {
705             return;
706         }
707         var size = GetSize(node);
708         var leftSize = GetLeftSize(node);
709         var rightSize = GetRightSize(node);
710         var expectedSize = Arithmetic.Increment(Arithmetic.Add(leftSize, rightSize));
711         if (!AreEqual(size, expectedSize))
712         {
713             throw new InvalidOperationException($"Size of {node} is not valid. Expected
714             ↪ size: {expectedSize}, actual size: {size}.");
715         }
716         ValidateSizes(GetLeft(node));
717         ValidateSizes(GetRight(node));
718     }
719
720     /// <summary>
721     /// <para>
722     /// Validates the size using the specified node.
723     /// </para>
724     /// </summary>
725     /// <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     public class RecursionlessSizeBalancedTree<TElement> :
11         ↳ RecursionlessSizeBalancedTreeMethods<TElement>
12     {
13         private struct TreeElement
14         {
15             public TElement Size;
16             public TElement Left;
17             public TElement Right;
18         }
19         private readonly TreeElement[] _elements;
20         private TElement _allocated;
21
22         public TElement Root;
23
24         public TElement Count => GetSizeOrZero(Root);
25
26         public RecursionlessSizeBalancedTree(int capacity) => (_elements, _allocated) = (new
27             ↳ TreeElement[capacity], One);
28
29         public TElement Allocate()
30         {
31             var newNode = _allocated;
32             if (IsEmpty(newNode))
33             {
34                 _allocated = Arithmetic.Increment(_allocated);
35                 return newNode;
36             }
37             else
38             {
39                 throw new InvalidOperationException("Allocated tree element is not empty.");
40             }
41         }
42
43         public void Free(TElement node)
44         {
45             while (!EqualityComparer.Equals(_allocated, One) && IsEmpty(node))
46             {
47                 var lastNode = Arithmetic.Decrement(_allocated);
48                 if (EqualityComparer.Equals(lastNode, node))
49                 {
50                     _allocated = lastNode;
51                     node = Arithmetic.Decrement(node);
52                 }
53                 else
54                 {
55                     return;
56                 }
57             }
58         }
59
60         public bool IsEmpty(TElement node) =>
61             ↳ EqualityComparer<TreeElement>.Default.Equals(GetElement(node), default);

```

```

60     protected override bool FirstIsToLeftOfSecond(TElement first, TElement second) =>
61         ↪ Comparer.Compare(first, second) < 0;
62     protected override bool FirstIsToTheRightOfSecond(TElement first, TElement second) =>
63         ↪ Comparer.Compare(first, second) > 0;
64     protected override ref TElement GetLeftReference(TElement node) => ref
65         ↪ GetElement(node).Left;
66     protected override TElement GetLeft(TElement node) => GetElement(node).Left;
67     protected override ref TElement GetRightReference(TElement node) => ref
68         ↪ GetElement(node).Right;
69     protected override TElement GetRight(TElement node) => GetElement(node).Right;
70     protected override TElement GetSize(TElement node) => GetElement(node).Size;
71     protected override void PrintNodeValue(TElement node, StringBuilder sb) =>
72         ↪ sb.Append(node);
73     protected override void SetLeft(TElement node, TElement left) => GetElement(node).Left =
74         ↪ left;
75     protected override void SetRight(TElement node, TElement right) =>
76         ↪ GetElement(node).Right = right;
77     protected override void SetSize(TElement node, TElement size) => GetElement(node).Size =
78         ↪ size;
79     private ref TreeElement GetElement(TElement node) => ref
80         ↪ _elements[UncheckedConverter<TElement, long>.Default.Convert(node)];
81     }
82 }
83 }

```

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     public class SizeBalancedTree<TElement> : SizeBalancedTreeMethods<TElement>
11     {
12         private struct TreeElement
13         {
14             public TElement Size;
15             public TElement Left;
16             public TElement Right;
17         }
18         private readonly TreeElement[] _elements;
19         private TElement _allocated;
20
21         public TElement Root;
22
23         public TElement Count => GetSizeOrZero(Root);
24
25         public SizeBalancedTree(int capacity) => (_elements, _allocated) = (new
26             ↪ TreeElement[capacity], One);
27
28         public TElement Allocate()
29         {
30             var newNode = _allocated;
31             if (IsEmpty(newNode))
32             {
33                 _allocated = Arithmetic.Increment(_allocated);
34                 return newNode;
35             }
36             else
37             {
38                 throw new InvalidOperationException("Allocated tree element is not empty.");
39             }
40         }
41
42         public void Free(TElement node)
43         {
44             while (!EqualityComparer.Equals(_allocated, One) && IsEmpty(node))

```

```

45     var lastNode = Arithmetic.Decrement(_allocated);
46     if (EqualityComparer.Equals(lastNode, node))
47     {
48         _allocated = lastNode;
49         node = Arithmetic.Decrement(node);
50     }
51     else
52     {
53         return;
54     }
55 }
56 }
57
58 public bool IsEmpty(TElement node) =>
59     ↪ EqualityComparer<TreeElement>.Default.Equals(GetElement(node), default);
60
61 protected override bool FirstIsToTheLeftOfSecond(TElement first, TElement second) =>
62     ↪ Comparer.Compare(first, second) < 0;
63
64 protected override bool FirstIsToTheRightOfSecond(TElement first, TElement second) =>
65     ↪ Comparer.Compare(first, second) > 0;
66
67 protected override ref TElement GetLeftReference(TElement node) => ref
68     ↪ GetElement(node).Left;
69
70 protected override TElement GetLeft(TElement node) => GetElement(node).Left;
71
72 protected override ref TElement GetRightReference(TElement node) => ref
73     ↪ GetElement(node).Right;
74
75 protected override TElement GetRight(TElement node) => GetElement(node).Right;
76
77 protected override TElement GetSize(TElement node) => GetElement(node).Size;
78
79 protected override void PrintNodeValue(TElement node, StringBuilder sb) =>
80     ↪ sb.Append(node);
81
82 protected override void SetLeft(TElement node, TElement left) => GetElement(node).Left =
83     ↪ left;
84
85 protected override void SetRight(TElement node, TElement right) =>
86     ↪ GetElement(node).Right = right;
87
88 protected override void SetSize(TElement node, TElement size) => GetElement(node).Size =
89     ↪ size;
90
91 private ref TreeElement GetElement(TElement node) => ref
92     ↪ _elements[UncheckedConverter<TElement, long>.Default.Convert(node)];
93
94 }
95 }

```

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     public class SizedAndThreadedAVLBalancedTree<TElement> :
11         ↪ SizedAndThreadedAVLBalancedTreeMethods<TElement>
12     {
13         private struct TreeElement
14         {
15             public TElement Size;
16             public TElement Left;
17             public TElement Right;
18             public sbyte Balance;
19             public bool LeftIsChild;
20             public bool RightIsChild;
21         }
22         private readonly TreeElement[] _elements;
23         private TElement _allocated;
24
25         public TElement Root;
26
27         public TElement Count => GetSizeOrZero(Root);
28
29     }
30 }

```



```

28 public SizedAndThreadedAVLBalancedTree(int capacity) => (_elements, _allocated) = (new
    ↳ TreeElement[capacity], One);
29
30 public TElement Allocate()
31 {
32     var newNode = _allocated;
33     if (IsEmpty(newNode))
34     {
35         _allocated = Arithmetic.Increment(_allocated);
36         return newNode;
37     }
38     else
39     {
40         throw new InvalidOperationException("Allocated tree element is not empty.");
41     }
42 }
43
44 public void Free(TElement node)
45 {
46     while (!EqualityComparer.Equals(_allocated, One) && IsEmpty(node))
47     {
48         var lastNode = Arithmetic.Decrement(_allocated);
49         if (EqualityComparer.Equals(lastNode, node))
50         {
51             _allocated = lastNode;
52             node = Arithmetic.Decrement(node);
53         }
54         else
55         {
56             return;
57         }
58     }
59 }
60
61 public bool IsEmpty(TElement node) =>
    ↳ EqualityComparer<TreeElement>.Default.Equals(GetElement(node), default);
62
63 protected override bool FirstIsToLeftOfSecond(TElement first, TElement second) =>
    ↳ Comparer.Compare(first, second) < 0;
64
65 protected override bool FirstIsToTheRightOfSecond(TElement first, TElement second) =>
    ↳ Comparer.Compare(first, second) > 0;
66
67 protected override sbyte GetBalance(TElement node) => GetElement(node).Balance;
68
69 protected override bool GetLeftIsChild(TElement node) => GetElement(node).LeftIsChild;
70
71 protected override ref TElement GetLeftReference(TElement node) => ref
    ↳ GetElement(node).Left;
72
73 protected override TElement GetLeft(TElement node) => GetElement(node).Left;
74
75 protected override bool GetRightIsChild(TElement node) => GetElement(node).RightIsChild;
76
77 protected override ref TElement GetRightReference(TElement node) => ref
    ↳ GetElement(node).Right;
78
79 protected override TElement GetRight(TElement node) => GetElement(node).Right;
80
81 protected override TElement GetSize(TElement node) => GetElement(node).Size;
82
83 protected override void PrintNodeValue(TElement node, StringBuilder sb) =>
    ↳ sb.Append(node);
84
85 protected override void SetBalance(TElement node, sbyte value) =>
    ↳ GetElement(node).Balance = value;
86
87 protected override void SetLeft(TElement node, TElement left) => GetElement(node).Left =
    ↳ left;
88
89 protected override void SetLeftIsChild(TElement node, bool value) =>
    ↳ GetElement(node).LeftIsChild = value;
90
91 protected override void SetRight(TElement node, TElement right) =>
    ↳ GetElement(node).Right = right;
92
93 protected override void SetRightIsChild(TElement node, bool value) =>
    ↳ GetElement(node).RightIsChild = value;
94

```

```

95     protected override void SetSize(TElement node, TElement size) => GetElement(node).Size =
    ↪ size;
96     private ref TreeElement GetElement(TElement node) => ref
    ↪ _elements[UncheckedConverter<TElement, long>.Default.Convert(node)];
97 }
98 }

```

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      public static class TestExtensions
10     {
11         public static void TestMultipleCreationsAndDeletions<TElement>(this
    ↪ SizedBinaryTreeMethodsBase<TElement> tree, Func<TElement> allocate, Action<TElement>
    ↪ free, ref TElement root, Func<TElement> treeCount, int maximumOperationsPerCycle)
12     {
13         for (var N = 1; N < maximumOperationsPerCycle; N++)
14         {
15             var currentCount = 0;
16             for (var i = 0; i < N; i++)
17             {
18                 var node = allocate();
19                 tree.Attach(ref root, node);
20                 currentCount++;
21                 Assert.Equal(currentCount, (int)UncheckedConverter<TElement,
    ↪ int>.Default.Convert(treeCount()));
22             }
23             for (var i = 1; i <= N; i++)
24             {
25                 TElement node = UncheckedConverter<int, TElement>.Default.Convert(i);
26                 if (tree.Contains(node, root))
27                 {
28                     tree.Detach(ref root, node);
29                     free(node);
30                     currentCount--;
31                     Assert.Equal(currentCount, (int)UncheckedConverter<TElement,
    ↪ int>.Default.Convert(treeCount()));
32                 }
33             }
34         }
35     }
36
37     public static void TestMultipleRandomCreationsAndDeletions<TElement>(this
    ↪ SizedBinaryTreeMethodsBase<TElement> tree, ref TElement root, Func<TElement>
    ↪ treeCount, int maximumOperationsPerCycle)
38     {
39         var random = new System.Random(0);
40         var added = new HashSet<TElement>();
41         var currentCount = 0;
42         for (var N = 1; N < maximumOperationsPerCycle; N++)
43         {
44             for (var i = 0; i < N; i++)
45             {
46                 var node = UncheckedConverter<int, TElement>.Default.Convert(random.Next(1,
    ↪ N));
47                 if (added.Add(node))
48                 {
49                     tree.Attach(ref root, node);
50                     currentCount++;
51                     Assert.Equal(currentCount, (int)UncheckedConverter<TElement,
    ↪ int>.Default.Convert(treeCount()));
52                 }
53             }
54             for (var i = 1; i <= N; i++)
55             {
56                 TElement node = UncheckedConverter<int,
    ↪ TElement>.Default.Convert(random.Next(1, N));
57                 if (tree.Contains(node, root))
58                 {
59                     tree.Detach(ref root, node);
60                     currentCount--;
61                     Assert.Equal(currentCount, (int)UncheckedConverter<TElement,
    ↪ int>.Default.Convert(treeCount()));

```

```

62         added.Remove(node);
63     }
64 }
65 }
66 }
67 }
68 }

```

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

```

1  using Xunit;
2
3  namespace Platform.Collections.Methods.Tests
4  {
5      public static class TreesTests
6      {
7          private const int _n = 500;
8
9          [Fact]
10         public static void RecursionlessSizeBalancedTreeMultipleAttachAndDetachTest()
11         {
12             var recursionlessSizeBalancedTree = new RecursionlessSizeBalancedTree<uint>(10000);
13             recursionlessSizeBalancedTree.TestMultipleCreationsAndDeletions(recursionlessSizeBal
14                 ↪ ancedTree.Allocate, recursionlessSizeBalancedTree.Free, ref
15                 ↪ recursionlessSizeBalancedTree.Root, () => recursionlessSizeBalancedTree.Count,
16                 ↪ _n);
17         }
18
19         [Fact]
20         public static void SizeBalancedTreeMultipleAttachAndDetachTest()
21         {
22             var sizeBalancedTree = new SizeBalancedTree<uint>(10000);
23             sizeBalancedTree.TestMultipleCreationsAndDeletions(sizeBalancedTree.Allocate,
24                 ↪ sizeBalancedTree.Free, ref sizeBalancedTree.Root, () => sizeBalancedTree.Count,
25                 ↪ _n);
26         }
27
28         [Fact]
29         public static void SizedAndThreadedAVLBalancedTreeMultipleAttachAndDetachTest()
30         {
31             var avlTree = new SizedAndThreadedAVLBalancedTree<uint>(10000);
32             avlTree.TestMultipleCreationsAndDeletions(avlTree.Allocate, avlTree.Free, ref
33                 ↪ avlTree.Root, () => avlTree.Count, _n);
34         }
35
36         [Fact]
37         public static void RecursionlessSizeBalancedTreeMultipleRandomAttachAndDetachTest()
38         {
39             var recursionlessSizeBalancedTree = new RecursionlessSizeBalancedTree<uint>(10000);
40             recursionlessSizeBalancedTree.TestMultipleRandomCreationsAndDeletions(ref
41                 ↪ recursionlessSizeBalancedTree.Root, () => recursionlessSizeBalancedTree.Count,
42                 ↪ _n);
43         }
44
45         [Fact]
46         public static void SizeBalancedTreeMultipleRandomAttachAndDetachTest()
47         {
48             var sizeBalancedTree = new SizeBalancedTree<uint>(10000);
49             sizeBalancedTree.TestMultipleRandomCreationsAndDeletions(ref sizeBalancedTree.Root,
50                 ↪ () => sizeBalancedTree.Count, _n);
51         }
52
53         [Fact]
54         public static void SizedAndThreadedAVLBalancedTreeMultipleRandomAttachAndDetachTest()
55         {
56             var avlTree = new SizedAndThreadedAVLBalancedTree<uint>(10000);
57             avlTree.TestMultipleRandomCreationsAndDeletions(ref avlTree.Root, () =>
58                 ↪ avlTree.Count, _n);
59         }
60     }
61 }

```

Index

- ./csharp/Platform.Collections.Methods.Tests/RecursionlessSizeBalancedTree.cs, 46
- ./csharp/Platform.Collections.Methods.Tests/SizeBalancedTree.cs, 47
- ./csharp/Platform.Collections.Methods.Tests/SizedAndThreadedAVLBalancedTree.cs, 48
- ./csharp/Platform.Collections.Methods.Tests/TestExtensions.cs, 50
- ./csharp/Platform.Collections.Methods.Tests/TreesTests.cs, 51
- ./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