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

```
using System.Collections.Generic;
using System.Runtime.CompilerServices;
using Platform.Numbers;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Collections.Methods
{
    /// <summary>
    /// <para>Represents a base implementation of methods for a collection of elements of type TElement.</para>
    /// <para>Представляет базовую реализацию методов коллекции элементов типа TElement.</para>
    /// </summary>
    /// <typeparam name="TElement"><para>Source type of conversion.</para><para>Исходный тип конверсии.</para></typeparam>
    public abstract class GenericCollectionMethodsBase<TElement>
    {
        /// <summary>
        /// <para>Returns a null constant of type <see cref="TElement" />.</para>
        /// <para>Возвращает нулевую константу типа <see cref="TElement" />.</para>
        /// </summary>
        /// <returns><para>A null constant of type <see cref="TElement" />.</para><para>Нулевую константу типа <see cref="TElement" />.</para></returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected virtual TElement GetZero() => default;

        /// <summary>
        /// <para>Determines whether the specified value is equal to zero type <see cref="TElement" />.</para>
        /// <para>Определяет равно ли нулю указанное значение типа <see cref="TElement" />.</para>
        /// </summary>
        /// <returns><para></para>Is the specified value equal to zero type <see cref="TElement" /><para>Равно ли нулю указанное значение типа <see cref="TElement" /></para></returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected virtual bool EqualToZero(TElement value) => EqualityComparer.Equals(value, Zero);

        /// <summary>
        /// <para>Presents the Range in readable format.</para>
        /// <para>Представляет диапазон в удобном для чтения формате.</para>
        /// </summary>
        /// <returns><para>String representation of the Range.</para><para>Строковое представление диапазона.</para></returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected virtual bool AreEqual(TElement first, TElement second) => EqualityComparer.Equals(first, second);

        /// <summary>
        /// <para>Presents the Range in readable format.</para>
        /// <para>Представляет диапазон в удобном для чтения формате.</para>
        /// </summary>
        /// <returns><para>String representation of the Range.</para><para>Строковое представление диапазона.</para></returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected virtual bool GreaterThanZero(TElement value) => Comparer.Compare(value, Zero) > 0;

        /// <summary>
        /// <para>Presents the Range in readable format.</para>
        /// <para>Представляет диапазон в удобном для чтения формате.</para>
        /// </summary>
        /// <returns><para>String representation of the Range.</para><para>Строковое представление диапазона.</para></returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected virtual bool GreaterThan(TElement first, TElement second) => Comparer.Compare(first, second) > 0;

        /// <summary>
        /// <para>Presents the Range in readable format.</para>
        /// <para>Представляет диапазон в удобном для чтения формате.</para>
        /// </summary>
        /// <returns><para>String representation of the Range.</para><para>Строковое 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>
```

```
        /// <para>
        /// Attaches the before using the specified base element.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="baseElement">
        /// <para>The base element.</para>
        /// <para></para>
        /// </param>
        /// <param name="newElement">
        /// <para>The new element.</para>
        /// <para></para>
        /// </param>
        public void AttachBefore(TElement baseElement, TElement newElement)
        {
            var baseElementPrevious = GetPrevious(baseElement);
            SetPrevious(newElement, baseElementPrevious);
            SetNext(newElement, baseElement);
            if (AreEqual(baseElement, GetFirst()))
            {
                SetFirst(newElement);
            }
            SetNext(baseElementPrevious, newElement);
            SetPrevious(baseElement, newElement);
            IncrementSize();
        }

        /// <summary>
        /// <para>
        /// Attaches the after using the specified base element.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="baseElement">
        /// <para>The base element.</para>
        /// <para></para>
        /// </param>
        /// <param name="newElement">
        /// <para>The new element.</para>
        /// <para></para>
        /// </param>
        public void AttachAfter(TElement baseElement, TElement newElement)
        {
            var baseElementNext = GetNext(baseElement);
            SetPrevious(newElement, baseElement);
            SetNext(newElement, baseElementNext);
            if (AreEqual(baseElement, GetLast()))
            {
                SetLast(newElement);
            }
            SetPrevious(baseElementNext, newElement);
            SetNext(baseElement, newElement);
            IncrementSize();
        }

        /// <summary>
        /// <para>
        /// Attaches the as first using the specified element.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="element">
        /// <para>The element.</para>
        /// <para></para>
        /// </param>
        public void AttachAsFirst(TElement element)
        {
            var first = GetFirst();
            if (EqualToZero(first))
            {
                SetFirst(element);
                SetLast(element);
                SetPrevious(element, element);
                SetNext(element, element);
                IncrementSize();
            }
            else
            {
```

```
                AttachBefore(first, element);
            }
        }

        /// <summary>
        /// <para>
        /// Attaches the as last using the specified element.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="element">
        /// <para>The element.</para>
        /// <para></para>
        /// </param>
        public void AttachAsLast(TElement element)
        {
            var last = GetLast();
            if (EqualToZero(last))
            {
                AttachAsFirst(element);
            }
            else
            {
                AttachAfter(last, element);
            }
        }

        /// <summary>
        /// <para>
        /// Detaches the element.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="element">
        /// <para>The element.</para>
        /// <para></para>
        /// </param>
        public void Detach(TElement element)
        {
            var elementPrevious = GetPrevious(element);
            var elementNext = GetNext(element);
            if (AreEqual(elementNext, element))
            {
                SetFirst(Zero);
                SetLast(Zero);
            }
            else
            {
                SetNext(elementPrevious, elementNext);
                SetPrevious(elementNext, elementPrevious);
                if (AreEqual(element, GetFirst()))
                {
                    SetFirst(elementNext);
                }
                if (AreEqual(element, GetLast()))
                {
                    SetLast(elementPrevious);
                }
            }
            SetPrevious(element, Zero);
            SetNext(element, Zero);
            DecrementSize();
        }
    }
}
```

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

```
using System.Runtime.CompilerServices;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Collections.Methods.Lists
{
    /// <summary>
    /// <para>
    /// Represents the absolute doubly linked list methods base.
    /// </para>
    /// <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>
```

```
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected abstract void SetSize(TElement size);

        /// <summary>
        /// <para>
        /// Increments the size.
        /// </para>
        /// <para></para>
        /// </summary>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected void IncrementSize() => SetSize(Increment(GetSize()));

        /// <summary>
        /// <para>
        /// Decrements the size.
        /// </para>
        /// <para></para>
        /// </summary>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected void DecrementSize() => SetSize(Decrement(GetSize()));
    }
}
```

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

```
#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Collections.Methods.Lists
{
    /// <summary>
    /// <para>
    /// Represents the absolute open doubly linked list methods.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <seealso cref="AbsoluteDoublyLinkedListMethodsBase{TElement}"/>
    public abstract class AbsoluteOpenDoublyLinkedListMethods<TElement> :
    ↪ AbsoluteDoublyLinkedListMethodsBase<TElement>
    {
        /// <summary>
        /// <para>
        /// Attaches the before using the specified base element.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="baseElement">
        /// <para>The base element.</para>
        /// <para></para>
        /// </param>
        /// <param name="newElement">
        /// <para>The new element.</para>
        /// <para></para>
        /// </param>
        public void AttachBefore(TElement baseElement, TElement newElement)
        {
            var baseElementPrevious = GetPrevious(baseElement);
            SetPrevious(newElement, baseElementPrevious);
            SetNext(newElement, baseElement);
            if (EqualToZero(baseElementPrevious))
            {
                SetFirst(newElement);
            }
            else
            {
                SetNext(baseElementPrevious, newElement);
            }
            SetPrevious(baseElement, newElement);
            IncrementSize();
        }

        /// <summary>
        /// <para>
        /// Attaches the after using the specified base element.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="baseElement">
```

```
        /// <para>The base element.</para>
        /// <para></para>
        /// </param>
        /// <param name="newElement">
        /// <para>The new element.</para>
        /// <para></para>
        /// </param>
        public void AttachAfter(TElement baseElement, TElement newElement)
        {
            var baseElementNext = GetNext(baseElement);
            SetPrevious(newElement, baseElement);
            SetNext(newElement, baseElementNext);
            if (EqualToZero(baseElementNext))
            {
                SetLast(newElement);
            }
            else
            {
                SetPrevious(baseElementNext, newElement);
            }
            SetNext(baseElement, newElement);
            IncrementSize();
        }

        /// <summary>
        /// <para>
        /// Attaches the as first using the specified element.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="element">
        /// <para>The element.</para>
        /// <para></para>
        /// </param>
        public void AttachAsFirst(TElement element)
        {
            var first = GetFirst();
            if (EqualToZero(first))
            {
                SetFirst(element);
                SetLast(element);
                SetPrevious(element, Zero);
                SetNext(element, Zero);
                IncrementSize();
            }
            else
            {
                AttachBefore(first, element);
            }
        }

        /// <summary>
        /// <para>
        /// Attaches the as last using the specified element.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="element">
        /// <para>The element.</para>
        /// <para></para>
        /// </param>
        public void AttachAsLast(TElement element)
        {
            var last = GetLast();
            if (EqualToZero(last))
            {
                AttachAsFirst(element);
            }
            else
            {
                AttachAfter(last, element);
            }
        }

        /// <summary>
        /// <para>
        /// Detaches the element.
        /// </para>
```

```
        /// <para></para>
        /// </summary>
        /// <param name="element">
        /// <para>The element.</para>
        /// <para></para>
        /// </param>
        public void Detach(TElement element)
        {
            var elementPrevious = GetPrevious(element);
            var elementNext = GetNext(element);
            if (EqualToZero(elementPrevious))
            {
                SetFirst(elementNext);
            }
            else
            {
                SetNext(elementPrevious, elementNext);
            }
            if (EqualToZero(elementNext))
            {
                SetLast(elementPrevious);
            }
            else
            {
                SetPrevious(elementNext, elementPrevious);
            }
            SetPrevious(element, Zero);
            SetNext(element, Zero);
            DecrementSize();
        }
    }
}
```

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

```
using System.Runtime.CompilerServices;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Collections.Methods.Lists
{
    /// <remarks>
    /// Based on <a href="https://en.wikipedia.org/wiki/Doubly_linked_list">doubly linked
    ↪  list</a> implementation.
    /// </remarks>
    public abstract class DoublyLinkedListMethodsBase<TElement> :
    ↪  GenericCollectionMethodsBase<TElement>
    {
        /// <summary>
        /// <para>
        /// Gets the previous using the specified element.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="element">
        /// <para>The element.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The element</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected abstract TElement GetPrevious(TElement element);

        /// <summary>
        /// <para>
        /// Gets the next using the specified element.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="element">
        /// <para>The element.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The element</para>
        /// <para></para>
        /// </returns>
```

```
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected abstract TElement GetNext(TElement element);

        /// <summary>
        /// <para>
        /// Sets the previous using the specified element.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="element">
        /// <para>The element.</para>
        /// <para></para>
        /// </param>
        /// <param name="previous">
        /// <para>The previous.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected abstract void SetPrevious(TElement element, TElement previous);

        /// <summary>
        /// <para>
        /// Sets the next using the specified element.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="element">
        /// <para>The element.</para>
        /// <para></para>
        /// </param>
        /// <param name="next">
        /// <para>The next.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected abstract void SetNext(TElement element, TElement next);
    }
}
```

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

```
#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Collections.Methods.Lists
{
    /// <summary>
    /// <para>
    /// Represents the relative circular doubly linked list methods.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <seealso cref="RelativeDoublyLinkedListMethodsBase{TElement}"/>
    public abstract class RelativeCircularDoublyLinkedListMethods<TElement> :
    ↪ RelativeDoublyLinkedListMethodsBase<TElement>
    {
        /// <summary>
        /// <para>
        /// Attaches the before using the specified head element.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="headElement">
        /// <para>The head element.</para>
        /// <para></para>
        /// </param>
        /// <param name="baseElement">
        /// <para>The base element.</para>
        /// <para></para>
        /// </param>
        /// <param name="newElement">
        /// <para>The new element.</para>
        /// <para></para>
        /// </param>
        public void AttachBefore(TElement headElement, TElement baseElement, TElement newElement)
        {
            var baseElementPrevious = GetPrevious(baseElement);
            SetPrevious(newElement, baseElementPrevious);
            SetNext(newElement, baseElement);
            if (AreEqual(baseElement, GetFirst(headElement)))
```

```
            {
                SetFirst(headElement, newElement);
            }
            SetNext(baseElementPrevious, newElement);
            SetPrevious(baseElement, newElement);
            IncrementSize(headElement);
        }

        /// <summary>
        /// <para>
        /// Attaches the after using the specified head element.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="headElement">
        /// <para>The head element.</para>
        /// <para></para>
        /// </param>
        /// <param name="baseElement">
        /// <para>The base element.</para>
        /// <para></para>
        /// </param>
        /// <param name="newElement">
        /// <para>The new element.</para>
        /// <para></para>
        /// </param>
        public void AttachAfter(TElement headElement, TElement baseElement, TElement newElement)
        {
            var baseElementNext = GetNext(baseElement);
            SetPrevious(newElement, baseElement);
            SetNext(newElement, baseElementNext);
            if (AreEqual(baseElement, GetLast(headElement)))
            {
                SetLast(headElement, newElement);
            }
            SetPrevious(baseElementNext, newElement);
            SetNext(baseElement, newElement);
            IncrementSize(headElement);
        }

        /// <summary>
        /// <para>
        /// Attaches the as first using the specified head element.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="headElement">
        /// <para>The head element.</para>
        /// <para></para>
        /// </param>
        /// <param name="element">
        /// <para>The element.</para>
        /// <para></para>
        /// </param>
        public void AttachAsFirst(TElement headElement, TElement element)
        {
            var first = GetFirst(headElement);
            if (EqualToZero(first))
            {
                SetFirst(headElement, element);
                SetLast(headElement, element);
                SetPrevious(element, element);
                SetNext(element, element);
                IncrementSize(headElement);
            }
            else
            {
                AttachBefore(headElement, first, element);
            }
        }

        /// <summary>
        /// <para>
        /// Attaches the as last using the specified head element.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="headElement">
```

```
        /// <para>The head element.</para>
        /// <para></para>
        /// </param>
        /// <param name="element">
        /// <para>The element.</para>
        /// <para></para>
        /// </param>
        public void AttachAsLast(TElement headElement, TElement element)
        {
            var last = GetLast(headElement);
            if (EqualToZero(last))
            {
                AttachAsFirst(headElement, element);
            }
            else
            {
                AttachAfter(headElement, last, element);
            }
        }

        /// <summary>
        /// <para>
        /// Detaches the head element.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="headElement">
        /// <para>The head element.</para>
        /// <para></para>
        /// </param>
        /// <param name="element">
        /// <para>The element.</para>
        /// <para></para>
        /// </param>
        public void Detach(TElement headElement, TElement element)
        {
            var elementPrevious = GetPrevious(element);
            var elementNext = GetNext(element);
            if (AreEqual(elementNext, element))
            {
                SetFirst(headElement, Zero);
                SetLast(headElement, Zero);
            }
            else
            {
                SetNext(elementPrevious, elementNext);
                SetPrevious(elementNext, elementPrevious);
                if (AreEqual(element, GetFirst(headElement)))
                {
                    SetFirst(headElement, elementNext);
                }
                if (AreEqual(element, GetLast(headElement)))
                {
                    SetLast(headElement, elementPrevious);
                }
            }
            SetPrevious(element, Zero);
            SetNext(element, Zero);
            DecrementSize(headElement);
        }
    }
}
```

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

```
using System.Runtime.CompilerServices;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Collections.Methods.Lists
{
    /// <summary>
    /// <para>
    /// Represents the relative doubly linked list methods base.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <seealso cref="DoublyLinkedListMethodsBase{TElement}"/>
    public abstract class RelativeDoublyLinkedListMethodsBase<TElement> :
    ↪  DoublyLinkedListMethodsBase<TElement>
```

```
    {
        /// <summary>
        /// <para>
        /// Gets the first using the specified head element.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="headElement">
        /// <para>The head element.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The element</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected abstract TElement GetFirst(TElement headElement);

        /// <summary>
        /// <para>
        /// Gets the last using the specified head element.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="headElement">
        /// <para>The head element.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The element</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected abstract TElement GetLast(TElement headElement);

        /// <summary>
        /// <para>
        /// Gets the size using the specified head element.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="headElement">
        /// <para>The head element.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The element</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected abstract TElement GetSize(TElement headElement);

        /// <summary>
        /// <para>
        /// Sets the first using the specified head element.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="headElement">
        /// <para>The head element.</para>
        /// <para></para>
        /// </param>
        /// <param name="element">
        /// <para>The element.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected abstract void SetFirst(TElement headElement, TElement element);

        /// <summary>
        /// <para>
        /// Sets the last using the specified head element.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="headElement">
        /// <para>The head element.</para>
        /// <para></para>
```

```
        /// </param>
        /// <param name="element">
        /// <para>The element.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected abstract void SetLast(TElement headElement, TElement element);

        /// <summary>
        /// <para>
        /// Sets the size using the specified head element.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="headElement">
        /// <para>The head element.</para>
        /// <para></para>
        /// </param>
        /// <param name="size">
        /// <para>The size.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected abstract void SetSize(TElement headElement, TElement size);

        /// <summary>
        /// <para>
        /// Increments the size using the specified head element.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="headElement">
        /// <para>The head element.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected void IncrementSize(TElement headElement) => SetSize(headElement,
        ↪  Increment(GetSize(headElement)));

        /// <summary>
        /// <para>
        /// Decrements the size using the specified head element.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="headElement">
        /// <para>The head element.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected void DecrementSize(TElement headElement) => SetSize(headElement,
        ↪  Decrement(GetSize(headElement)));
    }
}
```

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

```
#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Collections.Methods.Lists
{
    /// <summary>
    /// <para>
    /// Represents the relative open doubly linked list methods.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <seealso cref="RelativeDoublyLinkedListMethodsBase{TElement}"/>
    public abstract class RelativeOpenDoublyLinkedListMethods<TElement> :
    ↪  RelativeDoublyLinkedListMethodsBase<TElement>
    {
        /// <summary>
        /// <para>
        /// Attaches the before using the specified head element.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="headElement">
        /// <para>The head element.</para>
```

```
        /// <para></para>
        /// </param>
        /// <param name="baseElement">
        /// <para>The base element.</para>
        /// <para></para>
        /// </param>
        /// <param name="newElement">
        /// <para>The new element.</para>
        /// <para></para>
        /// </param>
        public void AttachBefore(TElement headElement, TElement baseElement, TElement newElement)
        {
            var baseElementPrevious = GetPrevious(baseElement);
            SetPrevious(newElement, baseElementPrevious);
            SetNext(newElement, baseElement);
            if (EqualToZero(baseElementPrevious))
            {
                SetFirst(headElement, newElement);
            }
            else
            {
                SetNext(baseElementPrevious, newElement);
            }
            SetPrevious(baseElement, newElement);
            IncrementSize(headElement);
        }

        /// <summary>
        /// <para>
        /// Attaches the after using the specified head element.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="headElement">
        /// <para>The head element.</para>
        /// <para></para>
        /// </param>
        /// <param name="baseElement">
        /// <para>The base element.</para>
        /// <para></para>
        /// </param>
        /// <param name="newElement">
        /// <para>The new element.</para>
        /// <para></para>
        /// </param>
        public void AttachAfter(TElement headElement, TElement baseElement, TElement newElement)
        {
            var baseElementNext = GetNext(baseElement);
            SetPrevious(newElement, baseElement);
            SetNext(newElement, baseElementNext);
            if (EqualToZero(baseElementNext))
            {
                SetLast(headElement, newElement);
            }
            else
            {
                SetPrevious(baseElementNext, newElement);
            }
            SetNext(baseElement, newElement);
            IncrementSize(headElement);
        }

        /// <summary>
        /// <para>
        /// Attaches the as first using the specified head element.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="headElement">
        /// <para>The head element.</para>
        /// <para></para>
        /// </param>
        /// <param name="element">
        /// <para>The element.</para>
        /// <para></para>
        /// </param>
        public void AttachAsFirst(TElement headElement, TElement element)
        {
```

```
            var first = GetFirst(headElement);
            if (EqualToZero(first))
            {
                SetFirst(headElement, element);
                SetLast(headElement, element);
                SetPrevious(element, Zero);
                SetNext(element, Zero);
                IncrementSize(headElement);
            }
            else
            {
                AttachBefore(headElement, first, element);
            }
        }

        /// <summary>
        /// <para>
        /// Attaches the as last using the specified head element.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="headElement">
        /// <para>The head element.</para>
        /// <para></para>
        /// </param>
        /// <param name="element">
        /// <para>The element.</para>
        /// <para></para>
        /// </param>
        public void AttachAsLast(TElement headElement, TElement element)
        {
            var last = GetLast(headElement);
            if (EqualToZero(last))
            {
                AttachAsFirst(headElement, element);
            }
            else
            {
                AttachAfter(headElement, last, element);
            }
        }

        /// <summary>
        /// <para>
        /// Detaches the head element.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="headElement">
        /// <para>The head element.</para>
        /// <para></para>
        /// </param>
        /// <param name="element">
        /// <para>The element.</para>
        /// <para></para>
        /// </param>
        public void Detach(TElement headElement, TElement element)
        {
            var elementPrevious = GetPrevious(element);
            var elementNext = GetNext(element);
            if (EqualToZero(elementPrevious))
            {
                SetFirst(headElement, elementNext);
            }
            else
            {
                SetNext(elementPrevious, elementNext);
            }
            if (EqualToZero(elementNext))
            {
                SetLast(headElement, elementPrevious);
            }
            else
            {
                SetPrevious(elementNext, elementPrevious);
            }
            SetPrevious(element, Zero);
            SetNext(element, Zero);
```

```
            DecrementSize(headElement);
        }
    }
}
```

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

```
#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Collections.Methods.Trees
{
    /// <summary>
    /// <para>
    /// Represents the recursionless size balanced tree methods.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <seealso cref="SizedBinaryTreeMethodsBase{TElement}"/>
    public abstract class RecursionlessSizeBalancedTreeMethods<TElement> :
    ↪ SizedBinaryTreeMethodsBase<TElement>
    {
        /// <summary>
        /// <para>
        /// Attaches the core using the specified root.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="root">
        /// <para>The root.</para>
        /// <para></para>
        /// </param>
        /// <param name="node">
        /// <para>The node.</para>
        /// <para></para>
        /// </param>
        protected override void AttachCore(ref TElement root, TElement node)
        {
            while (true)
            {
                ref var left = ref GetLeftReference(root);
                var leftSize = GetSizeOrZero(left);
                ref var right = ref GetRightReference(root);
                var rightSize = GetSizeOrZero(right);
                if (FirstIsToTheLeftOfSecond(node, root)) // node.Key less than root.Key
                {
                    if (EqualToZero(left))
                    {
                        IncrementSize(root);
                        SetSize(node, One);
                        left = node;
                        return;
                    }
                    if (FirstIsToTheLeftOfSecond(node, left)) // node.Key less than left.Key
                    {
                        if (GreaterThan(Increment(leftSize), rightSize))
                        {
                            RightRotate(ref root);
                        }
                        else
                        {
                            IncrementSize(root);
                            root = ref left;
                        }
                    }
                    else  // node.Key greater than left.Key
                    {
                        var leftRightSize = GetSizeOrZero(GetRight(left));
                        if (GreaterThan(Increment(leftRightSize), rightSize))
                        {
                            if (EqualToZero(leftRightSize) && EqualToZero(rightSize))
                            {
                                SetLeft(node, left);
                                SetRight(node, root);
                                SetSize(node, Add(leftSize, Two)); // Two (2) - node the size of
                                ↪  root and a node itself
                                SetLeft(root, Zero);
                                SetSize(root, One);
                                root = node;
                                return;
```

```
                        }
                        LeftRotate(ref left);
                        RightRotate(ref root);
                    }
                    else
                    {
                        IncrementSize(root);
                        root = ref left;
                    }
                }
            }
            else // node.Key greater than root.Key
            {
                if (EqualToZero(right))
                {
                    IncrementSize(root);
                    SetSize(node, One);
                    right = node;
                    return;
                }
                if (FirstIsToTheRightOfSecond(node, right)) // node.Key greater than
                ↪ right.Key
                {
                    if (GreaterThan(Increment(rightSize), leftSize))
                    {
                        LeftRotate(ref root);
                    }
                    else
                    {
                        IncrementSize(root);
                        root = ref right;
                    }
                }
                else // node.Key less than right.Key
                {
                    var rightLeftSize = GetSizeOrZero(GetLeft(right));
                    if (GreaterThan(Increment(rightLeftSize), leftSize))
                    {
                        if (EqualToZero(rightLeftSize) && EqualToZero(leftSize))
                        {
                            SetLeft(node, root);
                            SetRight(node, right);
                            SetSize(node, Add(rightSize, Two)); // Two (2) - node the size
                            ↪  of root and a node itself
                            SetRight(root, Zero);
                            SetSize(root, One);
                            root = node;
                            return;
                        }
                        RightRotate(ref right);
                        LeftRotate(ref root);
                    }
                    else
                    {
                        IncrementSize(root);
                        root = ref right;
                    }
                }
            }
        }
    }

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

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

```
                    return;
                }
            }
        }
    }
}
```

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

```
using System;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Collections.Methods.Trees
{
    /// <summary>
    /// <para>
    /// Represents the size balanced tree methods.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <seealso cref="SizedBinaryTreeMethodsBase{TElement}"/>
    public abstract class SizeBalancedTreeMethods<TElement> :
    ↪ SizedBinaryTreeMethodsBase<TElement>
    {
        /// <summary>
        /// <para>
        /// Attaches the core using the specified root.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="root">
        /// <para>The root.</para>
        /// <para></para>
        /// </param>
        /// <param name="node">
        /// <para>The node.</para>
        /// <para></para>
        /// </param>
        protected override void AttachCore(ref TElement root, TElement node)
        {
            if (EqualToZero(root))
            {
                root = node;
                IncrementSize(root);
            }
            else
            {
                IncrementSize(root);
                if (FirstIsToTheLeftOfSecond(node, root))
                {
                    AttachCore(ref GetLeftReference(root), node);
                    LeftMaintain(ref root);
                }
                else
                {
                    AttachCore(ref GetRightReference(root), node);
                    RightMaintain(ref root);
                }
            }
        }

        /// <summary>
        /// <para>
        /// Detaches the core using the specified root.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="root">
        /// <para>The root.</para>
        /// <para></para>
        /// </param>
        /// <param name="nodeToDetach">
        /// <para>The node to detach.</para>
        /// <para></para>
        /// </param>
        /// <exception cref="InvalidOperationException">
        /// <para>Duplicate link found in the tree.</para>
        /// <para></para>
```

```
        /// </exception>
        protected override void DetachCore(ref TElement root, TElement nodeToDetach)
        {
            ref var currentNode = ref root;
            ref var parent = ref root;
            var replacementNode = Zero;
            while (!AreEqual(currentNode, nodeToDetach))
            {
                DecrementSize(currentNode);
                if (FirstIsToTheLeftOfSecond(nodeToDetach, currentNode))
                {
                    parent = ref currentNode;
                    currentNode = ref GetLeftReference(currentNode);
                }
                else if (FirstIsToTheRightOfSecond(nodeToDetach, currentNode))
                {
                    parent = ref currentNode;
                    currentNode = ref GetRightReference(currentNode);
                }
                else
                {
                    throw new InvalidOperationException("Duplicate link found in the tree.");
                }
            }
            var nodeToDetachLeft = GetLeft(nodeToDetach);
            var node = GetRight(nodeToDetach);
            if (!EqualToZero(nodeToDetachLeft) && !EqualToZero(node))
            {
                var leftestNode = GetLeftest(node);
                DetachCore(ref GetRightReference(nodeToDetach), leftestNode);
                SetLeft(leftestNode, nodeToDetachLeft);
                node = GetRight(nodeToDetach);
                if (!EqualToZero(node))
                {
                    SetRight(leftestNode, node);
                    SetSize(leftestNode, Increment(Add(GetSize(nodeToDetachLeft),
                    ↪  GetSize(node))));
                }
                else
                {
                    SetSize(leftestNode, Increment(GetSize(nodeToDetachLeft)));
                }
                replacementNode = leftestNode;
            }
            else if (!EqualToZero(nodeToDetachLeft))
            {
                replacementNode = nodeToDetachLeft;
            }
            else if (!EqualToZero(node))
            {
                replacementNode = node;
            }
            if (AreEqual(root, nodeToDetach))
            {
                root = replacementNode;
            }
            else if (AreEqual(GetLeft(parent), nodeToDetach))
            {
                SetLeft(parent, replacementNode);
            }
            else if (AreEqual(GetRight(parent), nodeToDetach))
            {
                SetRight(parent, replacementNode);
            }
            ClearNode(nodeToDetach);
        }

        /// <summary>
        /// <para>
        /// Lefts the maintain using the specified root.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="root">
        /// <para>The root.</para>
        /// <para></para>
        /// </param>
        private void LeftMaintain(ref TElement root)
```

```
        {
            if (!EqualToZero(root))
            {
                var rootLeftNode = GetLeft(root);
                if (!EqualToZero(rootLeftNode))
                {
                    var rootRightNode = GetRight(root);
                    var rootRightNodeSize = GetSize(rootRightNode);
                    var rootLeftNodeLeftNode = GetLeft(rootLeftNode);
                    if (!EqualToZero(rootLeftNodeLeftNode) &&
                        (EqualToZero(rootRightNode) ||
                        ↪  GreaterThan(GetSize(rootLeftNodeLeftNode), rootRightNodeSize)))
                    {
                        RightRotate(ref root);
                    }
                    else
                    {
                        var rootLeftNodeRightNode = GetRight(rootLeftNode);
                        if (!EqualToZero(rootLeftNodeRightNode) &&
                            (EqualToZero(rootRightNode) ||
                            ↪  GreaterThan(GetSize(rootLeftNodeRightNode), rootRightNodeSize)))
                        {
                            LeftRotate(ref GetLeftReference(root));
                            RightRotate(ref root);
                        }
                        else
                        {
                            return;
                        }
                    }
                    LeftMaintain(ref GetLeftReference(root));
                    RightMaintain(ref GetRightReference(root));
                    LeftMaintain(ref root);
                    RightMaintain(ref root);
                }
            }
        }

        /// <summary>
        /// <para>
        /// Rights the maintain using the specified root.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="root">
        /// <para>The root.</para>
        /// <para></para>
        /// </param>
        private void RightMaintain(ref TElement root)
        {
            if (!EqualToZero(root))
            {
                var rootRightNode = GetRight(root);
                if (!EqualToZero(rootRightNode))
                {
                    var rootLeftNode = GetLeft(root);
                    var rootLeftNodeSize = GetSize(rootLeftNode);
                    var rootRightNodeRightNode = GetRight(rootRightNode);
                    if (!EqualToZero(rootRightNodeRightNode) &&
                        (EqualToZero(rootLeftNode) ||
                        ↪  GreaterThan(GetSize(rootRightNodeRightNode), rootLeftNodeSize)))
                    {
                        LeftRotate(ref root);
                    }
                    else
                    {
                        var rootRightNodeLeftNode = GetLeft(rootRightNode);
                        if (!EqualToZero(rootRightNodeLeftNode) &&
                            (EqualToZero(rootLeftNode) ||
                            ↪  GreaterThan(GetSize(rootRightNodeLeftNode), rootLeftNodeSize)))
                        {
                            RightRotate(ref GetRightReference(root));
                            LeftRotate(ref root);
                        }
                        else
                        {
                            return;
                        }
```

```
                    }
                    LeftMaintain(ref GetLeftReference(root));
                    RightMaintain(ref GetRightReference(root));
                    LeftMaintain(ref root);
                    RightMaintain(ref root);
                }
            }
        }
    }
}
```

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

```
using System;
using System.Runtime.CompilerServices;
using System.Text;
#if USEARRAYPOOL
using Platform.Collections;
#endif
using Platform.Reflection;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Collections.Methods.Trees
{
    /// <summary>
    /// Combination of Size, Height (AVL), and threads.
    /// </summary>
    /// <remarks>
    /// Based on: <a href="https://github.com/programmatom/TreeLib/blob/master/TreeLib/TreeLib/Generated/AVLTreeList.cs">TreeLib.AVLTreeList</a>.
    /// Which itself based on: <a href="https://github.com/GNOME/glib/blob/master/glib/gtree.c">GNOME/glib/gtree</a>.
    /// </remarks>
    public abstract class SizedAndThreadedAVLBalancedTreeMethods<TElement> : SizedBinaryTreeMethodsBase<TElement>
    {
        /// <summary>
        /// <para>
        /// The bytes size.
        /// </para>
        /// <para></para>
        /// </summary>
        private static readonly int _maxPath = 11 * NumericType<TElement>.BytesSize + 4;

        /// <summary>
        /// <para>
        /// Gets the rightest using the specified current.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="current">
        /// <para>The current.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The current.</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override TElement GetRightest(TElement current)
        {
            var currentRight = GetRightOrDefault(current);
            while (!EqualToZero(currentRight))
            {
                current = currentRight;
                currentRight = GetRightOrDefault(current);
            }
            return current;
        }

        /// <summary>
        /// <para>
        /// Gets the leftest using the specified current.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="current">
        /// <para>The current.</para>
        /// <para></para>
```

```
        /// </param>
        /// <returns>
        /// <para>The current.</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override TElement GetLeftest(TElement current)
        {
            var currentLeft = GetLeftOrDefault(current);
            while (!EqualToZero(currentLeft))
            {
                current = currentLeft;
                currentLeft = GetLeftOrDefault(current);
            }
            return current;
        }

        /// <summary>
        /// <para>
        /// Determines whether this instance contains.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="node">
        /// <para>The node.</para>
        /// <para></para>
        /// </param>
        /// <param name="root">
        /// <para>The root.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        public override bool Contains(TElement node, TElement root)
        {
            while (!EqualToZero(root))
            {
                if (FirstIsToTheLeftOfSecond(node, root)) // node.Key < root.Key
                {
                    root = GetLeftOrDefault(root);
                }
                else if (FirstIsToTheRightOfSecond(node, root)) // node.Key > root.Key
                {
                    root = GetRightOrDefault(root);
                }
                else // node.Key == root.Key
                {
                    return true;
                }
            }
            return false;
        }

        /// <summary>
        /// <para>
        /// Prints the node using the specified node.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="node">
        /// <para>The node.</para>
        /// <para></para>
        /// </param>
        /// <param name="sb">
        /// <para>The sb.</para>
        /// <para></para>
        /// </param>
        /// <param name="level">
        /// <para>The level.</para>
        /// <para></para>
        /// </param>
        protected override void PrintNode(TElement node, StringBuilder sb, int level)
        {
            base.PrintNode(node, sb, level);
            sb.Append(' ');
            sb.Append(GetLeftIsChild(node) ? 'l' : 'L');
```

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

```
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected abstract bool GetLeftIsChild(TElement node);

        /// <summary>
        /// <para>
        /// Sets the left is child using the specified node.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="node">
        /// <para>The node.</para>
        /// <para></para>
        /// </param>
        /// <param name="value">
        /// <para>The value.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected abstract void SetLeftIsChild(TElement node, bool value);

        /// <summary>
        /// <para>
        /// Determines whether this instance get right is child.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="node">
        /// <para>The node.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected abstract bool GetRightIsChild(TElement node);

        /// <summary>
        /// <para>
        /// Sets the right is child using the specified node.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="node">
        /// <para>The node.</para>
        /// <para></para>
        /// </param>
        /// <param name="value">
        /// <para>The value.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected abstract void SetRightIsChild(TElement node, bool value);

        /// <summary>
        /// <para>
        /// Gets the balance using the specified node.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="node">
        /// <para>The node.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The sbyte</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected abstract sbyte GetBalance(TElement node);

        /// <summary>
        /// <para>
```

```
295         /// Sets the balance using the specified node.
296         /// </para>
297         /// <para></para>
298         /// </summary>
299         /// <param name="node">
300         /// <para>The node.</para>
301         /// <para></para>
302         /// </param>
303         /// <param name="value">
304         /// <para>The value.</para>
305         /// <para></para>
306         /// </param>
307         [MethodImpl(MethodImplOptions.AggressiveInlining)]
308         protected abstract void SetBalance(TElement node, sbyte value);
309 
310         /// <summary>
311         /// <para>
312         /// Attaches the core using the specified root.
313         /// </para>
314         /// <para></para>
315         /// </summary>
316         /// <param name="root">
317         /// <para>The root.</para>
318         /// <para></para>
319         /// </param>
320         /// <param name="node">
321         /// <para>The node.</para>
322         /// <para></para>
323         /// </param>
324         /// <exception cref="InvalidOperationException">
325         /// <para>Node with the same key already attached to a tree.</para>
326         /// <para></para>
327         /// </exception>
328         protected override void AttachCore(ref TElement root, TElement node)
329         {
330             unchecked
331             {
332                 // TODO: Check what is faster to use simple array or array from array pool
333                 // TODO: Try to use stackalloc as an optimization (requires code generation,
                    ↪  because of generics)
334 #if USEARRAYPOOL
335                 var path = ArrayPool.Allocate<TElement>(MaxPath);
336                 var pathPosition = 0;
337                 path[pathPosition++] = default;
338 #else
339                 var path = new TElement[_maxPath];
340                 var pathPosition = 1;
341 #endif
342                 var currentNode = root;
343                 while (true)
344                 {
345                     if (FirstIsToTheLeftOfSecond(node, currentNode))
346                     {
347                         if (GetLeftIsChild(currentNode))
348                         {
349                             IncrementSize(currentNode);
350                             path[pathPosition++] = currentNode;
351                             currentNode = GetLeft(currentNode);
352                         }
353                         else
354                         {
355                             // Threads
356                             SetLeft(node, GetLeft(currentNode));
357                             SetRight(node, currentNode);
358                             SetLeft(currentNode, node);
359                             SetLeftIsChild(currentNode, true);
360                             DecrementBalance(currentNode);
361                             SetSize(node, One);
362                             FixSize(currentNode); // Should be incremented already
363                             break;
364                         }
365                     }
366                     else if (FirstIsToTheRightOfSecond(node, currentNode))
367                     {
368                         if (GetRightIsChild(currentNode))
369                         {
370                             IncrementSize(currentNode);
371                             path[pathPosition++] = currentNode;
```

```
                            currentNode = GetRight(currentNode);
                        }
                        else
                        {
                            // Threads
                            SetRight(node, GetRight(currentNode));
                            SetLeft(node, currentNode);
                            SetRight(currentNode, node);
                            SetRightIsChild(currentNode, true);
                            IncrementBalance(currentNode);
                            SetSize(node, One);
                            FixSize(currentNode); // Should be incremented already
                            break;
                        }
                    }
                    else
                    {
                        throw new InvalidOperationException("Node with the same key already
                        ↪  attached to a tree.");
                    }
                }
                // Restore balance. This is the goodness of a non-recursive
                // implementation, when we are done with balancing we 'break'
                // the loop and we are done.
                while (true)
                {
                    var parent = path[--pathPosition];
                    var isLeftNode = !AreEqual(parent, default) && AreEqual(currentNode,
                    ↪  GetLeft(parent));
                    var currentNodeBalance = GetBalance(currentNode);
                    if (currentNodeBalance < -1 || currentNodeBalance > 1)
                    {
                        currentNode = Balance(currentNode);
                        if (AreEqual(parent, default))
                        {
                            root = currentNode;
                        }
                        else if (isLeftNode)
                        {
                            SetLeft(parent, currentNode);
                            FixSize(parent);
                        }
                        else
                        {
                            SetRight(parent, currentNode);
                            FixSize(parent);
                        }
                    }
                    currentNodeBalance = GetBalance(currentNode);
                    if (currentNodeBalance == 0 || AreEqual(parent, default))
                    {
                        break;
                    }
                    if (isLeftNode)
                    {
                        DecrementBalance(parent);
                    }
                    else
                    {
                        IncrementBalance(parent);
                    }
                    currentNode = parent;
                }
#if USEARRAYPOOL
                ArrayPool.Free(path);
#endif
            }
        }

        /// <summary>
        /// <para>
        /// Balances the node.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="node">
        /// <para>The node.</para>
        /// <para></para>
```

```
        /// </param>
        /// <returns>
        /// <para>The element</para>
        /// <para></para>
        /// </returns>
        private TElement Balance(TElement node)
        {
            unchecked
            {
                var rootBalance = GetBalance(node);
                if (rootBalance < -1)
                {
                    var left = GetLeft(node);
                    if (GetBalance(left) > 0)
                    {
                        SetLeft(node, LeftRotateWithBalance(left));
                        FixSize(node);
                    }
                    node = RightRotateWithBalance(node);
                }
                else if (rootBalance > 1)
                {
                    var right = GetRight(node);
                    if (GetBalance(right) < 0)
                    {
                        SetRight(node, RightRotateWithBalance(right));
                        FixSize(node);
                    }
                    node = LeftRotateWithBalance(node);
                }
                return node;
            }
        }

        /// <summary>
        /// <para>
        /// Lefts the rotate with balance using the specified node.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="node">
        /// <para>The node.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The element</para>
        /// <para></para>
        /// </returns>
        protected TElement LeftRotateWithBalance(TElement node)
        {
            unchecked
            {
                var right = GetRight(node);
                if (GetLeftIsChild(right))
                {
                    SetRight(node, GetLeft(right));
                }
                else
                {
                    SetRightIsChild(node, false);
                    SetLeftIsChild(right, true);
                }
                SetLeft(right, node);
                // Fix size
                SetSize(right, GetSize(node));
                FixSize(node);
                // Fix balance
                var rootBalance = GetBalance(node);
                var rightBalance = GetBalance(right);
                if (rightBalance <= 0)
                {
                    if (rootBalance >= 1)
                    {
                        SetBalance(right, (sbyte)(rightBalance - 1));
                    }
                    else
                    {
                        SetBalance(right, (sbyte)(rootBalance + rightBalance - 2));
```

```
                    }
                    SetBalance(node, (sbyte)(rootBalance - 1));
                }
                else
                {
                    if (rootBalance <= rightBalance)
                    {
                        SetBalance(right, (sbyte)(rootBalance - 2));
                    }
                    else
                    {
                        SetBalance(right, (sbyte)(rightBalance - 1));
                    }
                    SetBalance(node, (sbyte)(rootBalance - rightBalance - 1));
                }
                return right;
            }
        }

        /// <summary>
        /// <para>
        /// Rights the rotate with balance using the specified node.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="node">
        /// <para>The node.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The element</para>
        /// <para></para>
        /// </returns>
        protected TElement RightRotateWithBalance(TElement node)
        {
            unchecked
            {
                var left = GetLeft(node);
                if (GetRightIsChild(left))
                {
                    SetLeft(node, GetRight(left));
                }
                else
                {
                    SetLeftIsChild(node, false);
                    SetRightIsChild(left, true);
                }
                SetRight(left, node);
                // Fix size
                SetSize(left, GetSize(node));
                FixSize(node);
                // Fix balance
                var rootBalance = GetBalance(node);
                var leftBalance = GetBalance(left);
                if (leftBalance <= 0)
                {
                    if (leftBalance > rootBalance)
                    {
                        SetBalance(left, (sbyte)(leftBalance + 1));
                    }
                    else
                    {
                        SetBalance(left, (sbyte)(rootBalance + 2));
                    }
                    SetBalance(node, (sbyte)(rootBalance - leftBalance + 1));
                }
                else
                {
                    if (rootBalance <= -1)
                    {
                        SetBalance(left, (sbyte)(leftBalance + 1));
                    }
                    else
                    {
                        SetBalance(left, (sbyte)(rootBalance + leftBalance + 2));
                    }
                    SetBalance(node, (sbyte)(rootBalance + 1));
                }
```

```
                return left;
            }
        }

        /// <summary>
        /// <para>
        /// Gets the next using the specified node.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="node">
        /// <para>The node.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The current.</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override TElement GetNext(TElement node)
        {
            var current = GetRight(node);
            if (GetRightIsChild(node))
            {
                return GetLeftest(current);
            }
            return current;
        }

        /// <summary>
        /// <para>
        /// Gets the previous using the specified node.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="node">
        /// <para>The node.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The current.</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override TElement GetPrevious(TElement node)
        {
            var current = GetLeft(node);
            if (GetLeftIsChild(node))
            {
                return GetRightest(current);
            }
            return current;
        }

        /// <summary>
        /// <para>
        /// Detaches the core using the specified root.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="root">
        /// <para>The root.</para>
        /// <para></para>
        /// </param>
        /// <param name="node">
        /// <para>The node.</para>
        /// <para></para>
        /// </param>
        /// <exception cref="InvalidOperationException">
        /// <para>Cannot find a node.</para>
        /// <para></para>
        /// </exception>
        /// <exception cref="InvalidOperationException">
        /// <para>Cannot find a node.</para>
        /// <para></para>
        /// </exception>
        protected override void DetachCore(ref TElement root, TElement node)
        {
```

```
            unchecked
            {
#if USEARRAYPOOL
                var path = ArrayPool.Allocate<TElement>(MaxPath);
                var pathPosition = 0;
                path[pathPosition++] = default;
#else
                var path = new TElement[_maxPath];
                var pathPosition = 1;
#endif
                var currentNode = root;
                while (true)
                {
                    if (FirstIsToTheLeftOfSecond(node, currentNode))
                    {
                        if (!GetLeftIsChild(currentNode))
                        {
                            throw new InvalidOperationException("Cannot find a node.");
                        }
                        DecrementSize(currentNode);
                        path[pathPosition++] = currentNode;
                        currentNode = GetLeft(currentNode);
                    }
                    else if (FirstIsToTheRightOfSecond(node, currentNode))
                    {
                        if (!GetRightIsChild(currentNode))
                        {
                            throw new InvalidOperationException("Cannot find a node.");
                        }
                        DecrementSize(currentNode);
                        path[pathPosition++] = currentNode;
                        currentNode = GetRight(currentNode);
                    }
                    else
                    {
                        break;
                    }
                }
                var parent = path[--pathPosition];
                var balanceNode = parent;
                var isLeftNode = !AreEqual(parent, default) && AreEqual(currentNode,
                ↪  GetLeft(parent));
                if (!GetLeftIsChild(currentNode))
                {
                    if (!GetRightIsChild(currentNode)) // node has no children
                    {
                        if (AreEqual(parent, default))
                        {
                            root = Zero;
                        }
                        else if (isLeftNode)
                        {
                            SetLeftIsChild(parent, false);
                            SetLeft(parent, GetLeft(currentNode));
                            IncrementBalance(parent);
                        }
                        else
                        {
                            SetRightIsChild(parent, false);
                            SetRight(parent, GetRight(currentNode));
                            DecrementBalance(parent);
                        }
                    }
                    else // node has a right child
                    {
                        var successor = GetNext(currentNode);
                        SetLeft(successor, GetLeft(currentNode));
                        var right = GetRight(currentNode);
                        if (AreEqual(parent, default))
                        {
                            root = right;
                        }
                        else if (isLeftNode)
                        {
                            SetLeft(parent, right);
                            IncrementBalance(parent);
                        }
                        else
                        {
```

```
                        SetRight(parent, right);
                        DecrementBalance(parent);
                    }
                }
            }
            else // node has a left child
            {
                if (!GetRightIsChild(currentNode))
                {
                    var predecessor = GetPrevious(currentNode);
                    SetRight(predecessor, GetRight(currentNode));
                    var leftValue = GetLeft(currentNode);
                    if (AreEqual(parent, default))
                    {
                        root = leftValue;
                    }
                    else if (isLeftNode)
                    {
                        SetLeft(parent, leftValue);
                        IncrementBalance(parent);
                    }
                    else
                    {
                        SetRight(parent, leftValue);
                        DecrementBalance(parent);
                    }
                }
                else // node has a both children (left and right)
                {
                    var predecessor = GetLeft(currentNode);
                    var successor = GetRight(currentNode);
                    var successorParent = currentNode;
                    int previousPathPosition = ++pathPosition;
                    // find the immediately next node (and its parent)
                    while (GetLeftIsChild(successor))
                    {
                        path[++pathPosition] = successorParent = successor;
                        successor = GetLeft(successor);
                        if (!AreEqual(successorParent, currentNode))
                        {
                            DecrementSize(successorParent);
                        }
                    }
                    path[previousPathPosition] = successor;
                    balanceNode = path[pathPosition];
                    // remove 'successor' from the tree
                    if (!AreEqual(successorParent, currentNode))
                    {
                        if (!GetRightIsChild(successor))
                        {
                            SetLeftIsChild(successorParent, false);
                        }
                        else
                        {
                            SetLeft(successorParent, GetRight(successor));
                        }
                        IncrementBalance(successorParent);
                        SetRightIsChild(successor, true);
                        SetRight(successor, GetRight(currentNode));
                    }
                    else
                    {
                        DecrementBalance(currentNode);
                    }
                    // set the predecessor's successor link to point to the right place
                    while (GetRightIsChild(predecessor))
                    {
                        predecessor = GetRight(predecessor);
                    }
                    SetRight(predecessor, successor);
                    // prepare 'successor' to replace 'node'
                    var left = GetLeft(currentNode);
                    SetLeftIsChild(successor, true);
                    SetLeft(successor, left);
                    SetBalance(successor, GetBalance(currentNode));
                    FixSize(successor);
                    if (AreEqual(parent, default))
                    {
```

```
                            root = successor;
                        }
                        else if (isLeftNode)
                        {
                            SetLeft(parent, successor);
                        }
                        else
                        {
                            SetRight(parent, successor);
                        }
                    }
                }
                // restore balance
                if (!AreEqual(balanceNode, default))
                {
                    while (true)
                    {
                        var balanceParent = path[--pathPosition];
                        isLeftNode = !AreEqual(balanceParent, default) && AreEqual(balanceNode,
                        ↪  GetLeft(balanceParent));
                        var currentNodeBalance = GetBalance(balanceNode);
                        if (currentNodeBalance < -1 || currentNodeBalance > 1)
                        {
                            balanceNode = Balance(balanceNode);
                            if (AreEqual(balanceParent, default))
                            {
                                root = balanceNode;
                            }
                            else if (isLeftNode)
                            {
                                SetLeft(balanceParent, balanceNode);
                            }
                            else
                            {
                                SetRight(balanceParent, balanceNode);
                            }
                        }
                        currentNodeBalance = GetBalance(balanceNode);
                        if (currentNodeBalance != 0 || AreEqual(balanceParent, default))
                        {
                            break;
                        }
                        if (isLeftNode)
                        {
                            IncrementBalance(balanceParent);
                        }
                        else
                        {
                            DecrementBalance(balanceParent);
                        }
                        balanceNode = balanceParent;
                    }
                }
                ClearNode(node);
#if USEARRAYPOOL
                ArrayPool.Free(path);
#endif
            }
        }

        /// <summary>
        /// <para>
        /// Clears the node using the specified node.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="node">
        /// <para>The node.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void ClearNode(TElement node)
        {
            SetLeft(node, Zero);
            SetRight(node, Zero);
            SetSize(node, Zero);
            SetLeftIsChild(node, false);
            SetRightIsChild(node, false);
```

```
            SetBalance(node, 0);
        }
    }
}
```

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

```
//#define ENABLE_TREE_AUTO_DEBUG_AND_VALIDATION

using System;
using System.Diagnostics;
using System.Runtime.CompilerServices;
using System.Text;
using Platform.Numbers;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Collections.Methods.Trees
{
    /// <summary>
    /// <para>
    /// Represents the sized binary tree methods base.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <seealso cref="GenericCollectionMethodsBase{TElement}"/>
    public abstract class SizedBinaryTreeMethodsBase<TElement> :
    ↪ GenericCollectionMethodsBase<TElement>
    {
        /// <summary>
        /// <para>
        /// Gets the left reference using the specified node.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="node">
        /// <para>The node.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The ref element</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected abstract ref TElement GetLeftReference(TElement node);

        /// <summary>
        /// <para>
        /// Gets the right reference using the specified node.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="node">
        /// <para>The node.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The ref element</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected abstract ref TElement GetRightReference(TElement node);

        /// <summary>
        /// <para>
        /// Gets the left using the specified node.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="node">
        /// <para>The node.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The element</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected abstract TElement GetLeft(TElement node);

```

```
        /// <summary>
        /// <para>
        /// Gets the right using the specified node.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="node">
        /// <para>The node.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The element</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected abstract TElement GetRight(TElement node);

        /// <summary>
        /// <para>
        /// Gets the size using the specified node.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="node">
        /// <para>The node.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The element</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected abstract TElement GetSize(TElement node);

        /// <summary>
        /// <para>
        /// Sets the left using the specified node.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="node">
        /// <para>The node.</para>
        /// <para></para>
        /// </param>
        /// <param name="left">
        /// <para>The left.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected abstract void SetLeft(TElement node, TElement left);

        /// <summary>
        /// <para>
        /// Sets the right using the specified node.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="node">
        /// <para>The node.</para>
        /// <para></para>
        /// </param>
        /// <param name="right">
        /// <para>The right.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected abstract void SetRight(TElement node, TElement right);

        /// <summary>
        /// <para>
        /// Sets the size using the specified node.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="node">
        /// <para>The node.</para>
        /// <para></para>
        /// </param>
```

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

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

```
305         /// <para>The element</para>
306         /// <para></para>
307         /// </returns>
308         [MethodImpl(MethodImplOptions.AggressiveInlining)]
309         protected TElement GetSizeOrZero(TElement node) => EqualToZero(node) ? Zero :
            ↪ GetSize(node);
310
311         /// <summary>
312         /// <para>
313         /// Fixes the size using the specified node.
314         /// </para>
315         /// <para></para>
316         /// </summary>
317         /// <param name="node">
318         /// <para>The node.</para>
319         /// <para></para>
320         /// </param>
321         [MethodImpl(MethodImplOptions.AggressiveInlining)]
322         protected void FixSize(TElement node) => SetSize(node, Increment(Add(GetLeftSize(node),
            ↪ GetRightSize(node))));
323
324         /// <summary>
325         /// <para>
326         /// Lefts the rotate using the specified root.
327         /// </para>
328         /// <para></para>
329         /// </summary>
330         /// <param name="root">
331         /// <para>The root.</para>
332         /// <para></para>
333         /// </param>
334         [MethodImpl(MethodImplOptions.AggressiveInlining)]
335         protected void LeftRotate(ref TElement root) => root = LeftRotate(root);
336
337         /// <summary>
338         /// <para>
339         /// Lefts the rotate using the specified root.
340         /// </para>
341         /// <para></para>
342         /// </summary>
343         /// <param name="root">
344         /// <para>The root.</para>
345         /// <para></para>
346         /// </param>
347         /// <returns>
348         /// <para>The right.</para>
349         /// <para></para>
350         /// </returns>
351         [MethodImpl(MethodImplOptions.AggressiveInlining)]
352         protected TElement LeftRotate(TElement root)
353         {
354             var right = GetRight(root);
355 #if ENABLE_TREE_AUTO_DEBUG_AND_VALIDATION
356             if (EqualToZero(right))
357             {
358                 throw new InvalidOperationException("Right is null.");
359             }
360 #endif
361             SetRight(root, GetLeft(right));
362             SetLeft(right, root);
363             SetSize(right, GetSize(root));
364             FixSize(root);
365             return right;
366         }
367
368         /// <summary>
369         /// <para>
370         /// Rights the rotate using the specified root.
371         /// </para>
372         /// <para></para>
373         /// </summary>
374         /// <param name="root">
375         /// <para>The root.</para>
376         /// <para></para>
377         /// </param>
378         [MethodImpl(MethodImplOptions.AggressiveInlining)]
379         protected void RightRotate(ref TElement root) => root = RightRotate(root);
380
```

```
        /// <summary>
        /// <para>
        /// Rights the rotate using the specified root.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="root">
        /// <para>The root.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The left.</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected TElement RightRotate(TElement root)
        {
            var left = GetLeft(root);
#if ENABLE_TREE_AUTO_DEBUG_AND_VALIDATION
            if (EqualToZero(left))
            {
                throw new InvalidOperationException("Left is null.");
            }
#endif
            SetLeft(root, GetRight(left));
            SetRight(left, root);
            SetSize(left, GetSize(root));
            FixSize(root);
            return left;
        }

        /// <summary>
        /// <para>
        /// Gets the rightest using the specified current.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="current">
        /// <para>The current.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The current.</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected virtual TElement GetRightest(TElement current)
        {
            var currentRight = GetRight(current);
            while (!EqualToZero(currentRight))
            {
                current = currentRight;
                currentRight = GetRight(current);
            }
            return current;
        }

        /// <summary>
        /// <para>
        /// Gets the leftest using the specified current.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="current">
        /// <para>The current.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The current.</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected virtual TElement GetLeftest(TElement current)
        {
            var currentLeft = GetLeft(current);
            while (!EqualToZero(currentLeft))
            {
                current = currentLeft;
```

```
            currentLeft = GetLeft(current);
        }
        return current;
    }

    /// <summary>
    /// <para>
    /// Gets the next using the specified node.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <param name="node">
    /// <para>The node.</para>
    /// <para></para>
    /// </param>
    /// <returns>
    /// <para>The element</para>
    /// <para></para>
    /// </returns>
    [MethodImpl(MethodImplOptions.AggressiveInlining)]
    protected virtual TElement GetNext(TElement node) => GetLeftest(GetRight(node));

    /// <summary>
    /// <para>
    /// Gets the previous using the specified node.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <param name="node">
    /// <para>The node.</para>
    /// <para></para>
    /// </param>
    /// <returns>
    /// <para>The element</para>
    /// <para></para>
    /// </returns>
    [MethodImpl(MethodImplOptions.AggressiveInlining)]
    protected virtual TElement GetPrevious(TElement node) => GetRightest(GetLeft(node));

    /// <summary>
    /// <para>
    /// Determines whether this instance contains.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <param name="node">
    /// <para>The node.</para>
    /// <para></para>
    /// </param>
    /// <param name="root">
    /// <para>The root.</para>
    /// <para></para>
    /// </param>
    /// <returns>
    /// <para>The bool</para>
    /// <para></para>
    /// </returns>
    [MethodImpl(MethodImplOptions.AggressiveInlining)]
    public virtual bool Contains(TElement node, TElement root)
    {
        while (!EqualToZero(root))
        {
            if (FirstIsToTheLeftOfSecond(node, root)) // node.Key < root.Key
            {
                root = GetLeft(root);
            }
            else if (FirstIsToTheRightOfSecond(node, root)) // node.Key > root.Key
            {
                root = GetRight(root);
            }
            else // node.Key == root.Key
            {
                return true;
            }
        }
        return false;
    }

```

```
        /// <summary>
        /// <para>
        /// Clears the node using the specified node.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="node">
        /// <para>The node.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected virtual void ClearNode(TElement node)
        {
            SetLeft(node, Zero);
            SetRight(node, Zero);
            SetSize(node, Zero);
        }

        /// <summary>
        /// <para>
        /// Attaches the root.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="root">
        /// <para>The root.</para>
        /// <para></para>
        /// </param>
        /// <param name="node">
        /// <para>The node.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public void Attach(ref TElement root, TElement node)
        {
#if ENABLE_TREE_AUTO_DEBUG_AND_VALIDATION
            ValidateSizes(root);
            Debug.WriteLine("--BeforeAttach--");
            Debug.WriteLine(PrintNodes(root));
            Debug.WriteLine("----------------");
            var sizeBefore = GetSize(root);
#endif
            if (EqualToZero(root))
            {
                SetSize(node, One);
                root = node;
                return;
            }
            AttachCore(ref root, node);
#if ENABLE_TREE_AUTO_DEBUG_AND_VALIDATION
            Debug.WriteLine("--AfterAttach--");
            Debug.WriteLine(PrintNodes(root));
            Debug.WriteLine("----------------");
            ValidateSizes(root);
            var sizeAfter = GetSize(root);
            if (!AreEqual(Arithmetic.Increment(sizeBefore), sizeAfter))
            {
                throw new InvalidOperationException("Tree was broken after attach.");
            }
#endif
        }

        /// <summary>
        /// <para>
        /// Attaches the core using the specified root.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="root">
        /// <para>The root.</para>
        /// <para></para>
        /// </param>
        /// <param name="node">
        /// <para>The node.</para>
        /// <para></para>
        /// </param>
        protected abstract void AttachCore(ref TElement root, TElement node);

```

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

```
        }

        /// <summary>
        /// <para>
        /// Validates the sizes using the specified node.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="node">
        /// <para>The node.</para>
        /// <para></para>
        /// </param>
        /// <exception cref="InvalidOperationException">
        /// <para>Size of {node} is not valid. Expected size: {expectedSize}, actual size:
        ↪  {size}.</para>
        /// <para></para>
        /// </exception>
        public void ValidateSizes(TElement node)
        {
            if (AreEqual(node, default))
            {
                return;
            }
            var size = GetSize(node);
            var leftSize = GetLeftSize(node);
            var rightSize = GetRightSize(node);
            var expectedSize = Arithmetic.Increment(Arithmetic.Add(leftSize, rightSize));
            if (!AreEqual(size, expectedSize))
            {
                throw new InvalidOperationException($"Size of {node} is not valid. Expected
                ↪  size: {expectedSize}, actual size: {size}.");
            }
            ValidateSizes(GetLeft(node));
            ValidateSizes(GetRight(node));
        }

        /// <summary>
        /// <para>
        /// Validates the size using the specified node.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="node">
        /// <para>The node.</para>
        /// <para></para>
        /// </param>
        /// <exception cref="InvalidOperationException">
        /// <para>Size of {node} is not valid. Expected size: {expectedSize}, actual size:
        ↪  {size}.</para>
        /// <para></para>
        /// </exception>
        public void ValidateSize(TElement node)
        {
            var size = GetSize(node);
            var leftSize = GetLeftSize(node);
            var rightSize = GetRightSize(node);
            var expectedSize = Arithmetic.Increment(Arithmetic.Add(leftSize, rightSize));
            if (!AreEqual(size, expectedSize))
            {
                throw new InvalidOperationException($"Size of {node} is not valid. Expected
                ↪  size: {expectedSize}, actual size: {size}.");
            }
        }

        /// <summary>
        /// <para>
        /// Prints the nodes using the specified node.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="node">
        /// <para>The node.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The string</para>
        /// <para></para>
```

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

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

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

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

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

```
        /// <summary>
        /// <para>
        /// Allocates this instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <exception cref="InvalidOperationException">
        /// <para>Allocated tree element is not empty.</para>
        /// <para></para>
        /// </exception>
        /// <returns>
        /// <para>The element</para>
        /// <para></para>
        /// </returns>
        public TElement Allocate()
        {
            var newNode = _allocated;
            if (IsEmpty(newNode))
            {
                _allocated = Arithmetic.Increment(_allocated);
                return newNode;
            }
            else
            {
                throw new InvalidOperationException("Allocated tree element is not empty.");
            }
        }

        /// <summary>
        /// <para>
        /// Frees the node.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="node">
        /// <para>The node.</para>
        /// <para></para>
        /// </param>
        public void Free(TElement node)
        {
            while (!EqualityComparer.Equals(_allocated, One) && IsEmpty(node))
            {
                var lastNode = Arithmetic.Decrement(_allocated);
                if (EqualityComparer.Equals(lastNode, node))
                {
                    _allocated = lastNode;
                    node = Arithmetic.Decrement(node);
                }
                else
                {
                    return;
                }
            }
        }

        /// <summary>
        /// <para>
        /// Determines whether this instance is empty.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="node">
        /// <para>The node.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        public bool IsEmpty(TElement node) =>
        ↪  EqualityComparer<TreeElement>.Default.Equals(GetElement(node), default);

        /// <summary>
        /// <para>
        /// Determines whether this instance first is to the left of second.
        /// </para>
        /// <para></para>
        /// </summary>
```

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

```
        /// </param>
        /// <returns>
        /// <para>The ref element</para>
        /// <para></para>
        /// </returns>
        protected override ref TElement GetRightReference(TElement node) => ref
        ↪  GetElement(node).Right;

        /// <summary>
        /// <para>
        /// Gets the right using the specified node.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="node">
        /// <para>The node.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The element</para>
        /// <para></para>
        /// </returns>
        protected override TElement GetRight(TElement node) => GetElement(node).Right;

        /// <summary>
        /// <para>
        /// Gets the size using the specified node.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="node">
        /// <para>The node.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The element</para>
        /// <para></para>
        /// </returns>
        protected override TElement GetSize(TElement node) => GetElement(node).Size;

        /// <summary>
        /// <para>
        /// Prints the node value using the specified node.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="node">
        /// <para>The node.</para>
        /// <para></para>
        /// </param>
        /// <param name="sb">
        /// <para>The sb.</para>
        /// <para></para>
        /// </param>
        protected override void PrintNodeValue(TElement node, StringBuilder sb) =>
        ↪  sb.Append(node);

        /// <summary>
        /// <para>
        /// Sets the left using the specified node.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="node">
        /// <para>The node.</para>
        /// <para></para>
        /// </param>
        /// <param name="left">
        /// <para>The left.</para>
        /// <para></para>
        /// </param>
        protected override void SetLeft(TElement node, TElement left) => GetElement(node).Left =
        ↪  left;

        /// <summary>
        /// <para>
        /// Sets the right using the specified node.
        /// </para>
```

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

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

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

```
            /// The size.
            /// </para>
            /// <para></para>
            /// </summary>
            public TElement Size;
            /// <summary>
            /// <para>
            /// The left.
            /// </para>
            /// <para></para>
            /// </summary>
            public TElement Left;
            /// <summary>
            /// <para>
            /// The right.
            /// </para>
            /// <para></para>
            /// </summary>
            public TElement Right;
        }

        /// <summary>
        /// <para>
        /// The elements.
        /// </para>
        /// <para></para>
        /// </summary>
        private readonly TreeElement[] _elements;
        /// <summary>
        /// <para>
        /// The allocated.
        /// </para>
        /// <para></para>
        /// </summary>
        private TElement _allocated;

        /// <summary>
        /// <para>
        /// The root.
        /// </para>
        /// <para></para>
        /// </summary>
        public TElement Root;

        /// <summary>
        /// <para>
        /// Gets the count value.
        /// </para>
        /// <para></para>
        /// </summary>
        public TElement Count => GetSizeOrZero(Root);

        /// <summary>
        /// <para>
        /// Initializes a new <see cref="SizeBalancedTree"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="capacity">
        /// <para>A capacity.</para>
        /// <para></para>
        /// </param>
        public SizeBalancedTree(int capacity) => (_elements, _allocated) = (new
        ↪  TreeElement[capacity], One);

        /// <summary>
        /// <para>
        /// Allocates this instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <exception cref="InvalidOperationException">
        /// <para>Allocated tree element is not empty.</para>
        /// <para></para>
        /// </exception>
        /// <returns>
        /// <para>The element</para>
        /// <para></para>
```

```
        /// </returns>
        public TElement Allocate()
        {
            var newNode = _allocated;
            if (IsEmpty(newNode))
            {
                _allocated = Arithmetic.Increment(_allocated);
                return newNode;
            }
            else
            {
                throw new InvalidOperationException("Allocated tree element is not empty.");
            }
        }

        /// <summary>
        /// <para>
        /// Frees the node.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="node">
        /// <para>The node.</para>
        /// <para></para>
        /// </param>
        public void Free(TElement node)
        {
            while (!EqualityComparer.Equals(_allocated, One) && IsEmpty(node))
            {
                var lastNode = Arithmetic.Decrement(_allocated);
                if (EqualityComparer.Equals(lastNode, node))
                {
                    _allocated = lastNode;
                    node = Arithmetic.Decrement(node);
                }
                else
                {
                    return;
                }
            }
        }

        /// <summary>
        /// <para>
        /// Determines whether this instance is empty.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="node">
        /// <para>The node.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        public bool IsEmpty(TElement node) =>
        ↪  EqualityComparer<TreeElement>.Default.Equals(GetElement(node), default);

        /// <summary>
        /// <para>
        /// Determines whether this instance first is to the left of second.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="first">
        /// <para>The first.</para>
        /// <para></para>
        /// </param>
        /// <param name="second">
        /// <para>The second.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
```

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

```
        /// <para></para>
        /// </summary>
        /// <param name="node">
        /// <para>The node.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The element</para>
        /// <para></para>
        /// </returns>
        protected override TElement GetRight(TElement node) => GetElement(node).Right;

        /// <summary>
        /// <para>
        /// Gets the size using the specified node.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="node">
        /// <para>The node.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The element</para>
        /// <para></para>
        /// </returns>
        protected override TElement GetSize(TElement node) => GetElement(node).Size;

        /// <summary>
        /// <para>
        /// Prints the node value using the specified node.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="node">
        /// <para>The node.</para>
        /// <para></para>
        /// </param>
        /// <param name="sb">
        /// <para>The sb.</para>
        /// <para></para>
        /// </param>
        protected override void PrintNodeValue(TElement node, StringBuilder sb) =>
        ↪  sb.Append(node);

        /// <summary>
        /// <para>
        /// Sets the left using the specified node.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="node">
        /// <para>The node.</para>
        /// <para></para>
        /// </param>
        /// <param name="left">
        /// <para>The left.</para>
        /// <para></para>
        /// </param>
        protected override void SetLeft(TElement node, TElement left) => GetElement(node).Left =
        ↪  left;

        /// <summary>
        /// <para>
        /// Sets the right using the specified node.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="node">
        /// <para>The node.</para>
        /// <para></para>
        /// </param>
        /// <param name="right">
        /// <para>The right.</para>
        /// <para></para>
        /// </param>
        protected override void SetRight(TElement node, TElement right) =>
        ↪  GetElement(node).Right = right;
```

```

        /// <summary>
        /// <para>
        /// Sets the size using the specified node.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="node">
        /// <para>The node.</para>
        /// <para></para>
        /// </param>
        /// <param name="size">
        /// <para>The size.</para>
        /// <para></para>
        /// </param>
        protected override void SetSize(TElement node, TElement size) => GetElement(node).Size =
        ↪  size;

        /// <summary>
        /// <para>
        /// Gets the element using the specified node.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="node">
        /// <para>The node.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The ref tree element</para>
        /// <para></para>
        /// </returns>
        private ref TreeElement GetElement(TElement node) => ref
        ↪  _elements[UncheckedConverter<TElement, long>.Default.Convert(node)];
    }
}
```

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

```
using System;
using System.Collections.Generic;
using System.Text;
using Platform.Numbers;
using Platform.Collections.Methods.Trees;
using Platform.Converters;

namespace Platform.Collections.Methods.Tests
{
    /// <summary>
    /// <para>
    /// Represents the sized and threaded avl balanced tree.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <seealso cref="SizedAndThreadedAVLBalancedTreeMethods{TElement}"/>
    public class SizedAndThreadedAVLBalancedTree<TElement> :
    ↪  SizedAndThreadedAVLBalancedTreeMethods<TElement>
    {
        /// <summary>
        /// <para>
        /// The tree element.
        /// </para>
        /// <para></para>
        /// </summary>
        private struct TreeElement
        {
            /// <summary>
            /// <para>
            /// The size.
            /// </para>
            /// <para></para>
            /// </summary>
            public TElement Size;
            /// <summary>
            /// <para>
            /// The left.
            /// </para>
            /// <para></para>
            /// </summary>
            public TElement Left;
```

```
            /// <summary>
            /// <para>
            /// The right.
            /// </para>
            /// <para></para>
            /// </summary>
            public TElement Right;
            /// <summary>
            /// <para>
            /// The balance.
            /// </para>
            /// <para></para>
            /// </summary>
            public sbyte Balance;
            /// <summary>
            /// <para>
            /// The left is child.
            /// </para>
            /// <para></para>
            /// </summary>
            public bool LeftIsChild;
            /// <summary>
            /// <para>
            /// The right is child.
            /// </para>
            /// <para></para>
            /// </summary>
            public bool RightIsChild;
        }

        /// <summary>
        /// <para>
        /// The elements.
        /// </para>
        /// <para></para>
        /// </summary>
        private readonly TreeElement[] _elements;
        /// <summary>
        /// <para>
        /// The allocated.
        /// </para>
        /// <para></para>
        /// </summary>
        private TElement _allocated;

        /// <summary>
        /// <para>
        /// The root.
        /// </para>
        /// <para></para>
        /// </summary>
        public TElement Root;

        /// <summary>
        /// <para>
        /// Gets the count value.
        /// </para>
        /// <para></para>
        /// </summary>
        public TElement Count => GetSizeOrZero(Root);

        /// <summary>
        /// <para>
        /// Initializes a new <see cref="SizedAndThreadedAVLBalancedTree"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="capacity">
        /// <para>A capacity.</para>
        /// <para></para>
        /// </param>
        public SizedAndThreadedAVLBalancedTree(int capacity) => (_elements, _allocated) = (new
        ↪  TreeElement[capacity], One);

        /// <summary>
        /// <para>
        /// Allocates this instance.
        /// </para>
```

```
        /// <para></para>
        /// </summary>
        /// <exception cref="InvalidOperationException">
        /// <para>Allocated tree element is not empty.</para>
        /// <para></para>
        /// </exception>
        /// <returns>
        /// <para>The element</para>
        /// <para></para>
        /// </returns>
        public TElement Allocate()
        {
            var newNode = _allocated;
            if (IsEmpty(newNode))
            {
                _allocated = Arithmetic.Increment(_allocated);
                return newNode;
            }
            else
            {
                throw new InvalidOperationException("Allocated tree element is not empty.");
            }
        }

        /// <summary>
        /// <para>
        /// Frees the node.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="node">
        /// <para>The node.</para>
        /// <para></para>
        /// </param>
        public void Free(TElement node)
        {
            while (!EqualityComparer.Equals(_allocated, One) && IsEmpty(node))
            {
                var lastNode = Arithmetic.Decrement(_allocated);
                if (EqualityComparer.Equals(lastNode, node))
                {
                    _allocated = lastNode;
                    node = Arithmetic.Decrement(node);
                }
                else
                {
                    return;
                }
            }
        }

        /// <summary>
        /// <para>
        /// Determines whether this instance is empty.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="node">
        /// <para>The node.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        public bool IsEmpty(TElement node) =>
        ↪  EqualityComparer<TreeElement>.Default.Equals(GetElement(node), default);

        /// <summary>
        /// <para>
        /// Determines whether this instance first is to the left of second.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="first">
        /// <para>The first.</para>
        /// <para></para>
        /// </param>
```

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

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

```
        /// <returns>
        /// <para>The element</para>
        /// <para></para>
        /// </returns>
        protected override TElement GetSize(TElement node) => GetElement(node).Size;

        /// <summary>
        /// <para>
        /// Prints the node value using the specified node.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="node">
        /// <para>The node.</para>
        /// <para></para>
        /// </param>
        /// <param name="sb">
        /// <para>The sb.</para>
        /// <para></para>
        /// </param>
        protected override void PrintNodeValue(TElement node, StringBuilder sb) =>
        ↪  sb.Append(node);

        /// <summary>
        /// <para>
        /// Sets the balance using the specified node.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="node">
        /// <para>The node.</para>
        /// <para></para>
        /// </param>
        /// <param name="value">
        /// <para>The value.</para>
        /// <para></para>
        /// </param>
        protected override void SetBalance(TElement node, sbyte value) =>
        ↪  GetElement(node).Balance = value;

        /// <summary>
        /// <para>
        /// Sets the left using the specified node.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="node">
        /// <para>The node.</para>
        /// <para></para>
        /// </param>
        /// <param name="left">
        /// <para>The left.</para>
        /// <para></para>
        /// </param>
        protected override void SetLeft(TElement node, TElement left) => GetElement(node).Left =
        ↪  left;

        /// <summary>
        /// <para>
        /// Sets the left is child using the specified node.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="node">
        /// <para>The node.</para>
        /// <para></para>
        /// </param>
        /// <param name="value">
        /// <para>The value.</para>
        /// <para></para>
        /// </param>
        protected override void SetLeftIsChild(TElement node, bool value) =>
        ↪  GetElement(node).LeftIsChild = value;

        /// <summary>
        /// <para>
        /// Sets the right using the specified node.
        /// </para>
```

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

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

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

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

```
        /// </typeparam>
        /// <param name="tree">
        /// <para>The tree.</para>
        /// <para></para>
        /// </param>
        /// <param name="root">
        /// <para>The root.</para>
        /// <para></para>
        /// </param>
        /// <param name="treeCount">
        /// <para>The tree count.</para>
        /// <para></para>
        /// </param>
        /// <param name="maximumOperationsPerCycle">
        /// <para>The maximum operations per cycle.</para>
        /// <para></para>
        /// </param>
        public static void TestMultipleRandomCreationsAndDeletions<TElement>(this
        ↪  SizedBinaryTreeMethodsBase<TElement> tree, ref TElement root, Func<TElement>
        ↪  treeCount, int maximumOperationsPerCycle)
        {
            var random = new System.Random(0);
            var added = new HashSet<TElement>();
            var currentCount = 0;
            for (var N = 1; N < maximumOperationsPerCycle; N++)
            {
                for (var i = 0; i < N; i++)
                {
                    var node = UncheckedConverter<int, TElement>.Default.Convert(random.Next(1,
                    ↪  N));
                    if (added.Add(node))
                    {
                        tree.Attach(ref root, node);
                        currentCount++;
                        Assert.Equal(currentCount, (int)UncheckedConverter<TElement,
                        ↪  int>.Default.Convert(treeCount()));
                    }
                }
                for (var i = 1; i <= N; i++)
                {
                    TElement node = UncheckedConverter<int,
                    ↪  TElement>.Default.Convert(random.Next(1, N));
                    if (tree.Contains(node, root))
                    {
                        tree.Detach(ref root, node);
                        currentCount--;
                        Assert.Equal(currentCount, (int)UncheckedConverter<TElement,
                        ↪  int>.Default.Convert(treeCount()));
                        added.Remove(node);
                    }
                }
            }
        }
    }
}
```

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

```
using Xunit;

namespace Platform.Collections.Methods.Tests
{
    /// <summary>
    /// <para>
    /// Represents the trees tests.
    /// </para>
    /// <para></para>
    /// </summary>
    public static class TreesTests
    {
        /// <summary>
        /// <para>
        /// The .
        /// </para>
        /// <para></para>
        /// </summary>
        private const int _n = 500;

        /// <summary>
```

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

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

Index