LinksPlatform's Platform.Collections.Methods Class Library

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

```csharp
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>
```

```csharp
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected virtual bool GreaterOrEqualThanZero(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 GreaterOrEqualThan(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>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected virtual bool LessOrEqualThanZero(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 LessOrEqualThan(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>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected virtual bool LessThanZero(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 LessThan(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>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected virtual TElement Increment(TElement value) =>
        ↪ Arithmetic<TElement>.Increment(value);

        /// <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 TElement Decrement(TElement value) =>
        ↪ Arithmetic<TElement>.Decrement(value);

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

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

        /// <summary>
        /// <para>Returns minimum value of the range.</para>
        /// <para>Возвращает минимальное значение диапазона.</para>
        /// </summary>
        protected readonly TElement Zero;

        /// <summary>
        /// <para>Returns minimum value of the range.</para>
        /// <para>Возвращает минимальное значение диапазона.</para>
        /// </summary>
        protected readonly TElement One;

         /// <summary>
        /// <para>Returns minimum value of the range.</para>
        /// <para>Возвращает минимальное значение диапазона.</para>
        /// </summary>
        protected readonly TElement Two;

        /// <summary>
        /// <para>Returns minimum value of the range.</para>
        /// <para>Возвращает минимальное значение диапазона.</para>
        /// </summary>
        protected readonly EqualityComparer<TElement> EqualityComparer;

        /// <summary>
        /// <para>Returns minimum value of the range.</para>
        /// <para>Возвращает минимальное значение диапазона.</para>
        /// </summary>
        protected readonly Comparer<TElement> Comparer;

        /// <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>
        protected GenericCollectionMethodsBase()
        {
            EqualityComparer = EqualityComparer<TElement>.Default;
            Comparer = Comparer<TElement>.Default;
            Zero = GetZero(); //-V3068
            One = Increment(Zero); //-V3068
            Two = Increment(One); //-V3068
        }
    }
}
```

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

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

namespace Platform.Collections.Methods.Lists
{
    /// <summary>
    /// <para>
    /// Represents the absolute circular doubly linked list methods.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <seealso cref="AbsoluteDoublyLinkedListMethodsBase{TElement}"/>
    public abstract class AbsoluteCircularDoublyLinkedListMethods<TElement> :
    ↪  AbsoluteDoublyLinkedListMethodsBase<TElement>
    {
        /// <summary>
```

```csharp
        /// <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>
```

```csharp
        /// </summary>
        /// <seealso cref="DoublyLinkedListMethodsBase{TElement}"/>
        public abstract class AbsoluteDoublyLinkedListMethodsBase<TElement> :
        ↪  DoublyLinkedListMethodsBase<TElement>
        {
            /// <summary>
            /// <para>
            /// Gets the first.
            /// </para>
            /// <para></para>
            /// </summary>
            /// <returns>
            /// <para>The element</para>
            /// <para></para>
            /// </returns>
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            protected abstract TElement GetFirst();

            /// <summary>
            /// <para>
            /// Gets the last.
            /// </para>
            /// <para></para>
            /// </summary>
            /// <returns>
            /// <para>The element</para>
            /// <para></para>
            /// </returns>
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            protected abstract TElement GetLast();

            /// <summary>
            /// <para>
            /// Gets the size.
            /// </para>
            /// <para></para>
            /// </summary>
            /// <returns>
            /// <para>The element</para>
            /// <para></para>
            /// </returns>
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            protected abstract TElement GetSize();

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

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

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

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

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

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

```csharp
        /// <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>
```

```csharp
        /// <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

```csharp
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>
```

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

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

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

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

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

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

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

```csharp
    {
        /// <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>
```

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

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

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

```csharp
        /// <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)
        {
```

```csharp
            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);
```

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

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

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

```csharp
                    }
                    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);
```

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

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

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

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

```csharp
                        {
                            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);
                    }
                }
            }
        }
        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

```csharp
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
{
```

```csharp
        /// <summary>
        /// Combination of Size, Height (AVL), and threads.
        /// </summary>
        /// <remarks>
        /// Based on: <a href="https://github.com/programmatom/TreeLib/blob/master/TreeLib/TreeLib/G⌋
        ↪    enerated/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>
        {
            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>
```

```csharp
        /// <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');
            sb.Append(GetRightIsChild(node) ? 'r' : 'R');
            sb.Append(' ');
            sb.Append(GetBalance(node));
        }

        /// <summary>
        /// <para>
        /// Increments the balance using the specified node.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="node">
        /// <para>The node.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected void IncrementBalance(TElement node) => SetBalance(node,
        ↪  (sbyte)(GetBalance(node) + 1));

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

```
165        [MethodImpl(MethodImplOptions.AggressiveInlining)]
166        protected void DecrementBalance(TElement node) => SetBalance(node,
           ↪  (sbyte)(GetBalance(node) - 1));
167
168        /// <summary>
169        /// <para>
170        /// Gets the left or default using the specified node.
171        /// </para>
172        /// <para></para>
173        /// </summary>
174        /// <param name="node">
175        /// <para>The node.</para>
176        /// <para></para>
177        /// </param>
178        /// <returns>
179        /// <para>The element</para>
180        /// <para></para>
181        /// </returns>
182        [MethodImpl(MethodImplOptions.AggressiveInlining)]
183        protected override TElement GetLeftOrDefault(TElement node) => GetLeftIsChild(node) ?
           ↪  GetLeft(node) : default;
184
185        /// <summary>
186        /// <para>
187        /// Gets the right or default using the specified node.
188        /// </para>
189        /// <para></para>
190        /// </summary>
191        /// <param name="node">
192        /// <para>The node.</para>
193        /// <para></para>
194        /// </param>
195        /// <returns>
196        /// <para>The element</para>
197        /// <para></para>
198        /// </returns>
199        [MethodImpl(MethodImplOptions.AggressiveInlining)]
200        protected override TElement GetRightOrDefault(TElement node) => GetRightIsChild(node) ?
           ↪  GetRight(node) : default;
201
202        /// <summary>
203        /// <para>
204        /// Determines whether this instance get left is child.
205        /// </para>
206        /// <para></para>
207        /// </summary>
208        /// <param name="node">
209        /// <para>The node.</para>
210        /// <para></para>
211        /// </param>
212        /// <returns>
213        /// <para>The bool</para>
214        /// <para></para>
215        /// </returns>
216        [MethodImpl(MethodImplOptions.AggressiveInlining)]
217        protected abstract bool GetLeftIsChild(TElement node);
218
219        /// <summary>
220        /// <para>
221        /// Sets the left is child using the specified node.
222        /// </para>
223        /// <para></para>
224        /// </summary>
225        /// <param name="node">
226        /// <para>The node.</para>
227        /// <para></para>
228        /// </param>
229        /// <param name="value">
230        /// <para>The value.</para>
231        /// <para></para>
232        /// </param>
233        [MethodImpl(MethodImplOptions.AggressiveInlining)]
234        protected abstract void SetLeftIsChild(TElement node, bool value);
235
236        /// <summary>
237        /// <para>
238        /// Determines whether this instance get right is child.
239        /// </para>
```

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

```csharp
        /// <exception cref="InvalidOperationException">
        /// <para>Node with the same key already attached to a tree.</para>
        /// <para></para>
        /// </exception>
        protected override void AttachCore(ref TElement root, TElement node)
        {
            unchecked
            {
                // TODO: Check what is faster to use simple array or array from array pool
                // TODO: Try to use stackalloc as an optimization (requires code generation,
                ↪  because of generics)
#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))
                        {
                            IncrementSize(currentNode);
                            path[pathPosition++] = currentNode;
                            currentNode = GetLeft(currentNode);
                        }
                        else
                        {
                            // Threads
                            SetLeft(node, GetLeft(currentNode));
                            SetRight(node, currentNode);
                            SetLeft(currentNode, node);
                            SetLeftIsChild(currentNode, true);
                            DecrementBalance(currentNode);
                            SetSize(node, One);
                            FixSize(currentNode); // Should be incremented already
                            break;
                        }
                    }
                    else if (FirstIsToTheRightOfSecond(node, currentNode))
                    {
                        if (GetRightIsChild(currentNode))
                        {
                            IncrementSize(currentNode);
                            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));
```

```csharp
                            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
                    }
                }
        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>
```

```csharp
        /// <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
                {
```

```csharp
                    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)]
```

```csharp
        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))
                {
```

```csharp
                    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);
                            }
                        }
```

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

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

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

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

```csharp
        /// <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>
```

```csharp
        /// </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>
        /// <param name="size">
        /// <para>The size.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected abstract void SetSize(TElement node, TElement size);

        /// <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>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected abstract bool FirstIsToTheLeftOfSecond(TElement first, TElement second);

        /// <summary>
        /// <para>
        /// Determines whether this instance first is to the right 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>
```

```csharp
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected abstract bool FirstIsToTheRightOfSecond(TElement first, TElement second);

        /// <summary>
        /// <para>
        /// Gets the left or default 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 GetLeftOrDefault(TElement node) => AreEqual(node, default) ?
        ↪    default : GetLeft(node);

        /// <summary>
        /// <para>
        /// Gets the right or default 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 GetRightOrDefault(TElement node) => AreEqual(node, default) ?
        ↪    default : GetRight(node);

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

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

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

```csharp
        /// <returns>
        /// <para>The element</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected TElement GetLeftSize(TElement node) => GetSizeOrZero(GetLeftOrDefault(node));

        /// <summary>
        /// <para>
        /// Gets the right 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 TElement GetRightSize(TElement node) => GetSizeOrZero(GetRightOrDefault(node));

        /// <summary>
        /// <para>
        /// Gets the size or zero 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 TElement GetSizeOrZero(TElement node) => EqualToZero(node) ? Zero :
        ↪  GetSize(node);

        /// <summary>
        /// <para>
        /// Fixes the size using the specified node.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="node">
        /// <para>The node.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected void FixSize(TElement node) => SetSize(node, Increment(Add(GetLeftSize(node),
        ↪  GetRightSize(node))));

        /// <summary>
        /// <para>
        /// Lefts the rotate using the specified root.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="root">
        /// <para>The root.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected void LeftRotate(ref TElement root) => root = LeftRotate(root);

        /// <summary>
        /// <para>
        /// Lefts the rotate using the specified root.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="root">
        /// <para>The root.</para>
        /// <para></para>
```

```csharp
        /// </param>
        /// <returns>
        /// <para>The right.</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected TElement LeftRotate(TElement root)
        {
            var right = GetRight(root);
#if ENABLE_TREE_AUTO_DEBUG_AND_VALIDATION
            if (EqualToZero(right))
            {
                throw new InvalidOperationException("Right is null.");
            }
#endif
            SetRight(root, GetLeft(right));
            SetLeft(right, root);
            SetSize(right, GetSize(root));
            FixSize(root);
            return right;
        }

        /// <summary>
        /// <para>
        /// Rights the rotate using the specified root.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="root">
        /// <para>The root.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected void RightRotate(ref TElement root) => root = RightRotate(root);

        /// <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>
```

```csharp
        /// <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>
```

```csharp
        /// <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))
```

```csharp
                {
                    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);

        /// <summary>
        /// <para>
        /// Detaches 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 Detach(ref TElement root, TElement node)
        {
#if ENABLE_TREE_AUTO_DEBUG_AND_VALIDATION
            ValidateSizes(root);
            Debug.WriteLine("--BeforeDetach--");
            Debug.WriteLine(PrintNodes(root));
            Debug.WriteLine("----------------");
            var sizeBefore = GetSize(root);
            if (EqualToZero(root))
            {
                throw new InvalidOperationException($"Элемент с {node} не содержится в
                    дереве.");
            }
#endif
            DetachCore(ref root, node);
#if ENABLE_TREE_AUTO_DEBUG_AND_VALIDATION
            Debug.WriteLine("--AfterDetach--");
            Debug.WriteLine(PrintNodes(root));
            Debug.WriteLine("----------------");
            ValidateSizes(root);
            var sizeAfter = GetSize(root);
            if (!AreEqual(Arithmetic.Decrement(sizeBefore), sizeAfter))
            {
                throw new InvalidOperationException("Tree was broken after detach.");
            }
#endif
        }
```

```csharp
        /// <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 abstract void DetachCore(ref TElement root, TElement node);

        /// <summary>
        /// <para>
        /// Fixes the sizes using the specified node.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="node">
        /// <para>The node.</para>
        /// <para></para>
        /// </param>
        public void FixSizes(TElement node)
        {
            if (AreEqual(node, default))
            {
                return;
            }
            FixSizes(GetLeft(node));
            FixSizes(GetRight(node));
            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">
```

```csharp
        /// <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>
        /// </returns>
        public string PrintNodes(TElement node)
        {
            var sb = new StringBuilder();
            PrintNodes(node, sb);
            return sb.ToString();
        }

        /// <summary>
        /// <para>
        /// Prints the nodes 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>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public void PrintNodes(TElement node, StringBuilder sb) => PrintNodes(node, sb, 0);

        /// <summary>
        /// <para>
        /// Prints the nodes 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>
        public void PrintNodes(TElement node, StringBuilder sb, int level)
```

```csharp
        {
            if (AreEqual(node, default))
            {
                return;
            }
            PrintNodes(GetLeft(node), sb, level + 1);
            PrintNode(node, sb, level);
            sb.AppendLine();
            PrintNodes(GetRight(node), sb, level + 1);
        }

        /// <summary>
        /// <para>
        /// Prints the node 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>
        /// </returns>
        public string PrintNode(TElement node)
        {
            var sb = new StringBuilder();
            PrintNode(node, sb);
            return sb.ToString();
        }

        /// <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>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected void PrintNode(TElement node, StringBuilder sb) => PrintNode(node, sb, 0);

        /// <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 virtual void PrintNode(TElement node, StringBuilder sb, int level)
        {
            sb.Append('\t', level);
            sb.Append(node);
            PrintNodeValue(node, sb);
            sb.Append(' ');
            sb.Append('s');
            sb.Append(GetSize(node));
        }

        /// <summary>
```

```
887    /// <para>
888    /// Prints the node value using the specified node.
889    /// </para>
890    /// <para></para>
891    /// </summary>
892    /// <param name="node">
893    /// <para>The node.</para>
894    /// <para></para>
895    /// </param>
896    /// <param name="sb">
897    /// <para>The sb.</para>
898    /// <para></para>
899    /// </param>
900    protected abstract void PrintNodeValue(TElement node, StringBuilder sb);
901  }
902 }
```

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

```
1  using System;
2  using System.Collections.Generic;
3  using System.Text;
4  using Platform.Numbers;
5  using Platform.Collections.Methods.Trees;
6  using Platform.Converters;
7
8  namespace Platform.Collections.Methods.Tests
9  {
10     public class RecursionlessSizeBalancedTree<TElement> :
     ↪   RecursionlessSizeBalancedTreeMethods<TElement>
11     {
12         private struct TreeElement
13         {
14             public TElement Size;
15             public TElement Left;
16             public TElement Right;
17         }
18         private readonly TreeElement[] _elements;
19         private TElement _allocated;
20
21         public TElement Root;
22
23         public TElement Count => GetSizeOrZero(Root);
24
25         public RecursionlessSizeBalancedTree(int capacity) => (_elements, _allocated) = (new
     ↪   TreeElement[capacity], One);
26
27         public TElement Allocate()
28         {
29             var newNode = _allocated;
30             if (IsEmpty(newNode))
31             {
32                 _allocated = Arithmetic.Increment(_allocated);
33                 return newNode;
34             }
35             else
36             {
37                 throw new InvalidOperationException("Allocated tree element is not empty.");
38             }
39         }
40
41         public void Free(TElement node)
42         {
43             while (!EqualityComparer.Equals(_allocated, One) && IsEmpty(node))
44             {
45                 var lastNode = Arithmetic.Decrement(_allocated);
46                 if (EqualityComparer.Equals(lastNode, node))
47                 {
48                     _allocated = lastNode;
49                     node = Arithmetic.Decrement(node);
50                 }
51                 else
52                 {
53                     return;
54                 }
55             }
56         }
57
58         public bool IsEmpty(TElement node) =>
     ↪   EqualityComparer<TreeElement>.Default.Equals(GetElement(node), default);
59
```

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

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

```
1  using System;
2  using System.Collections.Generic;
3  using System.Text;
4  using Platform.Numbers;
5  using Platform.Collections.Methods.Trees;
6  using Platform.Converters;
7
8  namespace Platform.Collections.Methods.Tests
9  {
10     public class SizeBalancedTree<TElement> : SizeBalancedTreeMethods<TElement>
11     {
12         private struct TreeElement
13         {
14             public TElement Size;
15             public TElement Left;
16             public TElement Right;
17         }
18         private readonly TreeElement[] _elements;
19         private TElement _allocated;
20
21         public TElement Root;
22
23         public TElement Count => GetSizeOrZero(Root);
24
25         public SizeBalancedTree(int capacity) => (_elements, _allocated) = (new
           ↪   TreeElement[capacity], One);
26
27         public TElement Allocate()
28         {
29             var newNode = _allocated;
30             if (IsEmpty(newNode))
31             {
32                 _allocated = Arithmetic.Increment(_allocated);
33                 return newNode;
34             }
35             else
36             {
37                 throw new InvalidOperationException("Allocated tree element is not empty.");
38             }
39         }
40
41         public void Free(TElement node)
42         {
43             while (!EqualityComparer.Equals(_allocated, One) && IsEmpty(node))
44             {
```

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

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

```
1  using System;
2  using System.Collections.Generic;
3  using System.Text;
4  using Platform.Numbers;
5  using Platform.Collections.Methods.Trees;
6  using Platform.Converters;
7
8  namespace Platform.Collections.Methods.Tests
9  {
10     public class SizedAndThreadedAVLBalancedTree<TElement> :
    ↪   SizedAndThreadedAVLBalancedTreeMethods<TElement>
11     {
12         private struct TreeElement
13         {
14             public TElement Size;
15             public TElement Left;
16             public TElement Right;
17             public sbyte Balance;
18             public bool LeftIsChild;
19             public bool RightIsChild;
20         }
21         private readonly TreeElement[] _elements;
22         private TElement _allocated;
23
24         public TElement Root;
25
26         public TElement Count => GetSizeOrZero(Root);
27
```

```csharp
28      public SizedAndThreadedAVLBalancedTree(int capacity) => (_elements, _allocated) = (new
   ↪    TreeElement[capacity], One);

30      public TElement Allocate()
31      {
32          var newNode = _allocated;
33          if (IsEmpty(newNode))
34          {
35              _allocated = Arithmetic.Increment(_allocated);
36              return newNode;
37          }
38          else
39          {
40              throw new InvalidOperationException("Allocated tree element is not empty.");
41          }
42      }

44      public void Free(TElement node)
45      {
46          while (!EqualityComparer.Equals(_allocated, One) && IsEmpty(node))
47          {
48              var lastNode = Arithmetic.Decrement(_allocated);
49              if (EqualityComparer.Equals(lastNode, node))
50              {
51                  _allocated = lastNode;
52                  node = Arithmetic.Decrement(node);
53              }
54              else
55              {
56                  return;
57              }
58          }
59      }

61      public bool IsEmpty(TElement node) =>
   ↪    EqualityComparer<TreeElement>.Default.Equals(GetElement(node), default);

63      protected override bool FirstIsToTheLeftOfSecond(TElement first, TElement second) =>
   ↪    Comparer.Compare(first, second) < 0;

65      protected override bool FirstIsToTheRightOfSecond(TElement first, TElement second) =>
   ↪    Comparer.Compare(first, second) > 0;

67      protected override sbyte GetBalance(TElement node) => GetElement(node).Balance;

69      protected override bool GetLeftIsChild(TElement node) => GetElement(node).LeftIsChild;

71      protected override ref TElement GetLeftReference(TElement node) => ref
   ↪    GetElement(node).Left;

73      protected override TElement GetLeft(TElement node) => GetElement(node).Left;

75      protected override bool GetRightIsChild(TElement node) => GetElement(node).RightIsChild;

77      protected override ref TElement GetRightReference(TElement node) => ref
   ↪    GetElement(node).Right;

79      protected override TElement GetRight(TElement node) => GetElement(node).Right;

81      protected override TElement GetSize(TElement node) => GetElement(node).Size;

83      protected override void PrintNodeValue(TElement node, StringBuilder sb) =>
   ↪    sb.Append(node);

85      protected override void SetBalance(TElement node, sbyte value) =>
   ↪    GetElement(node).Balance = value;

87      protected override void SetLeft(TElement node, TElement left) => GetElement(node).Left =
   ↪    left;

89      protected override void SetLeftIsChild(TElement node, bool value) =>
   ↪    GetElement(node).LeftIsChild = value;

91      protected override void SetRight(TElement node, TElement right) =>
   ↪    GetElement(node).Right = right;

93      protected override void SetRightIsChild(TElement node, bool value) =>
   ↪    GetElement(node).RightIsChild = value;
```

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

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

```csharp
using System;
using System.Collections.Generic;
using Xunit;
using Platform.Collections.Methods.Trees;
using Platform.Converters;

namespace Platform.Collections.Methods.Tests
{
    public static class TestExtensions
    {
        public static void TestMultipleCreationsAndDeletions<TElement>(this
        ↪    SizedBinaryTreeMethodsBase<TElement> tree, Func<TElement> allocate, Action<TElement>
        ↪    free, ref TElement root, Func<TElement> treeCount, int maximumOperationsPerCycle)
        {
            for (var N = 1; N < maximumOperationsPerCycle; N++)
            {
                var currentCount = 0;
                for (var i = 0; i < N; i++)
                {
                    var node = allocate();
                    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(i);
                    if (tree.Contains(node, root))
                    {
                        tree.Detach(ref root, node);
                        free(node);
                        currentCount--;
                        Assert.Equal(currentCount, (int)UncheckedConverter<TElement,
                        ↪    int>.Default.Convert(treeCount()));
                    }
                }
            }
        }

        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()));
```

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

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

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

Index