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

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

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

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

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

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

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

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

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

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

```
124         /// <returns><para>String representation of the Range.</para><para>Строковое
            ↪ представление диапазона.</para></returns>
125         [MethodImpl(MethodImplOptions.AggressiveInlining)]
126         protected virtual TElement Add(TElement first, TElement second) =>
            ↪ Arithmetic<TElement>.Add(first, second);
127
128         /// <summary>
129         /// <para>Presents the Range in readable format.</para>
130         /// <para>Представляет диапазон в удобном для чтения формате.</para>
131         /// </summary>
132         /// <returns><para>String representation of the Range.</para><para>Строковое
            ↪ представление диапазона.</para></returns>
133         [MethodImpl(MethodImplOptions.AggressiveInlining)]
134         protected virtual TElement Subtract(TElement first, TElement second) =>
            ↪ Arithmetic<TElement>.Subtract(first, second);
135
136         /// <summary>
137         /// <para>Returns minimum value of the range.</para>
138         /// <para>Возвращает минимальное значение диапазона.</para>
139         /// </summary>
140         protected readonly TElement Zero;
141
142         /// <summary>
143         /// <para>Returns minimum value of the range.</para>
144         /// <para>Возвращает минимальное значение диапазона.</para>
145         /// </summary>
146         protected readonly TElement One;
147
148          /// <summary>
149         /// <para>Returns minimum value of the range.</para>
150         /// <para>Возвращает минимальное значение диапазона.</para>
151         /// </summary>
152         protected readonly TElement Two;
153
154         /// <summary>
155         /// <para>Returns minimum value of the range.</para>
156         /// <para>Возвращает минимальное значение диапазона.</para>
157         /// </summary>
158         protected readonly EqualityComparer<TElement> EqualityComparer;
159
160         /// <summary>
161         /// <para>Returns minimum value of the range.</para>
162         /// <para>Возвращает минимальное значение диапазона.</para>
163         /// </summary>
164         protected readonly Comparer<TElement> Comparer;
165
166         /// <summary>
167         /// <para>Presents the Range in readable format.</para>
168         /// <para>Представляет диапазон в удобном для чтения формате.</para>
169         /// </summary>
170         /// <returns><para>String representation of the Range.</para><para>Строковое
            ↪ представление диапазона.</para></returns>
171         protected GenericCollectionMethodsBase()
172         {
173             EqualityComparer = EqualityComparer<TElement>.Default;
174             Comparer = Comparer<TElement>.Default;
175             Zero = GetZero(); //-V3068
176             One = Increment(Zero); //-V3068
177             Two = Increment(One); //-V3068
178         }
179     }
180 }
```

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

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

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

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

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

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

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

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

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

```
using System.Runtime.CompilerServices;

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
using System.Runtime.CompilerServices;

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
using System.Runtime.CompilerServices;

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
using System;

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

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

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

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

        private void LeftMaintain(ref TElement root)
        {
            if (!EqualToZero(root))
            {
                var rootLeftNode = GetLeft(root);
                if (!EqualToZero(rootLeftNode))
                {
                    var rootRightNode = GetRight(root);
                    var rootRightNodeSize = GetSize(rootRightNode);
                    var rootLeftNodeLeftNode = GetLeft(rootLeftNode);
                    if (!EqualToZero(rootLeftNodeLeftNode) &&
```

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

        private void RightMaintain(ref TElement root)
        {
            if (!EqualToZero(root))
            {
                var rootRightNode = GetRight(root);
                if (!EqualToZero(rootRightNode))
                {
                    var rootLeftNode = GetLeft(root);
                    var rootLeftNodeSize = GetSize(rootLeftNode);
                    var rootRightNodeRightNode = GetRight(rootRightNode);
                    if (!EqualToZero(rootRightNodeRightNode) &&
                        (EqualToZero(rootLeftNode) ||
                        ↪ GreaterThan(GetSize(rootRightNodeRightNode), rootLeftNodeSize)))
                    {
                        LeftRotate(ref root);
                    }
                    else
                    {
                        var rootRightNodeLeftNode = GetLeft(rootRightNode);
                        if (!EqualToZero(rootRightNodeLeftNode) &&
                            (EqualToZero(rootLeftNode) ||
                            ↪ GreaterThan(GetSize(rootRightNodeLeftNode), rootLeftNodeSize)))
                        {
                            RightRotate(ref GetRightReference(root));
                            LeftRotate(ref root);
                        }
                        else
                        {
                            return;
                        }
                    }
                    LeftMaintain(ref GetLeftReference(root));
                    RightMaintain(ref GetRightReference(root));
                    LeftMaintain(ref root);
                    RightMaintain(ref root);
                }
            }
        }
    }
}
```

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

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

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

```
10
11  namespace Platform.Collections.Methods.Trees
12  {
13      /// <summary>
14      /// Combination of Size, Height (AVL), and threads.
15      /// </summary>
16      /// <remarks>
17      /// Based on: <a href="https://github.com/programmatom/TreeLib/blob/master/TreeLib/TreeLib/G
        ↪  enerated/AVLTreeList.cs">TreeLib.AVLTreeList</a>.
18      /// Which itself based on: <a
        ↪  href="https://github.com/GNOME/glib/blob/master/glib/gtree.c">GNOME/glib/gtree</a>.
19      /// </remarks>
20      public abstract class SizedAndThreadedAVLBalancedTreeMethods<TElement> :
        ↪  SizedBinaryTreeMethodsBase<TElement>
21      {
22          private static readonly int _maxPath = 11 * NumericType<TElement>.BytesSize + 4;
23
24          /// <summary>
25          /// <para>
26          /// Gets the rightest using the specified current.
27          /// </para>
28          /// <para></para>
29          /// </summary>
30          /// <param name="current">
31          /// <para>The current.</para>
32          /// <para></para>
33          /// </param>
34          /// <returns>
35          /// <para>The current.</para>
36          /// <para></para>
37          /// </returns>
38          [MethodImpl(MethodImplOptions.AggressiveInlining)]
39          protected override TElement GetRightest(TElement current)
40          {
41              var currentRight = GetRightOrDefault(current);
42              while (!EqualToZero(currentRight))
43              {
44                  current = currentRight;
45                  currentRight = GetRightOrDefault(current);
46              }
47              return current;
48          }
49
50          /// <summary>
51          /// <para>
52          /// Gets the leftest using the specified current.
53          /// </para>
54          /// <para></para>
55          /// </summary>
56          /// <param name="current">
57          /// <para>The current.</para>
58          /// <para></para>
59          /// </param>
60          /// <returns>
61          /// <para>The current.</para>
62          /// <para></para>
63          /// </returns>
64          [MethodImpl(MethodImplOptions.AggressiveInlining)]
65          protected override TElement GetLeftest(TElement current)
66          {
67              var currentLeft = GetLeftOrDefault(current);
68              while (!EqualToZero(currentLeft))
69              {
70                  current = currentLeft;
71                  currentLeft = GetLeftOrDefault(current);
72              }
73              return current;
74          }
75
76          /// <summary>
77          /// <para>
78          /// Determines whether this instance contains.
79          /// </para>
80          /// <para></para>
81          /// </summary>
82          /// <param name="node">
83          /// <para>The node.</para>
84          /// <para></para>
```

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

        /// <summary>
        /// <para>
        /// Prints the node using the specified node.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="node">
        /// <para>The node.</para>
        /// <para></para>
        /// </param>
        /// <param name="sb">
        /// <para>The sb.</para>
        /// <para></para>
        /// </param>
        /// <param name="level">
        /// <para>The level.</para>
        /// <para></para>
        /// </param>
        protected override void PrintNode(TElement node, StringBuilder sb, int level)
        {
            base.PrintNode(node, sb, level);
            sb.Append(' ');
            sb.Append(GetLeftIsChild(node) ? 'l' : 'L');
            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>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected void DecrementBalance(TElement node) => SetBalance(node,
        ↪  (sbyte)(GetBalance(node) - 1));

        /// <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 override TElement GetLeftOrDefault(TElement node) => GetLeftIsChild(node) ?
        ↪  GetLeft(node) : default;

        /// <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 override TElement GetRightOrDefault(TElement node) => GetRightIsChild(node) ?
        ↪  GetRight(node) : default;

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

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

        /// <summary>
```

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

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

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

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

        /// <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">
```

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

```
                    var parent = path[--pathPosition];
                    var isLeftNode = !AreEqual(parent, default) && AreEqual(currentNode,
                    ↪  GetLeft(parent));
                    var currentNodeBalance = GetBalance(currentNode);
                    if (currentNodeBalance < -1 || currentNodeBalance > 1)
                    {
                        currentNode = Balance(currentNode);
                        if (AreEqual(parent, default))
                        {
                            root = currentNode;
                        }
                        else if (isLeftNode)
                        {
                            SetLeft(parent, currentNode);
                            FixSize(parent);
                        }
                        else
                        {
                            SetRight(parent, currentNode);
                            FixSize(parent);
                        }
                    }
                    currentNodeBalance = GetBalance(currentNode);
                    if (currentNodeBalance == 0 || AreEqual(parent, default))
                    {
                        break;
                    }
                    if (isLeftNode)
                    {
                        DecrementBalance(parent);
                    }
                    else
                    {
                        IncrementBalance(parent);
                    }
                    currentNode = parent;
                }
#if USEARRAYPOOL
                ArrayPool.Free(path);
#endif
            }
        }

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

        /// <summary>
        /// <para>
        /// Lefts the rotate with balance using the specified node.
        /// </para>
        /// <para></para>
        /// </summary>
```

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

        /// <summary>
        /// <para>
        /// Rights the rotate with balance using the specified node.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="node">
        /// <para>The node.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The element</para>
        /// <para></para>
        /// </returns>
        protected TElement RightRotateWithBalance(TElement node)
        {
            unchecked
            {
                var left = GetLeft(node);
                if (GetRightIsChild(left))
                {
```

```
                    SetLeft(node, GetRight(left));
                }
                else
                {
                    SetLeftIsChild(node, false);
                    SetRightIsChild(left, true);
                }
                SetRight(left, node);
                // Fix size
                SetSize(left, GetSize(node));
                FixSize(node);
                // Fix balance
                var rootBalance = GetBalance(node);
                var leftBalance = GetBalance(left);
                if (leftBalance <= 0)
                {
                    if (leftBalance > rootBalance)
                    {
                        SetBalance(left, (sbyte)(leftBalance + 1));
                    }
                    else
                    {
                        SetBalance(left, (sbyte)(rootBalance + 2));
                    }
                    SetBalance(node, (sbyte)(rootBalance - leftBalance + 1));
                }
                else
                {
                    if (rootBalance <= -1)
                    {
                        SetBalance(left, (sbyte)(leftBalance + 1));
                    }
                    else
                    {
                        SetBalance(left, (sbyte)(rootBalance + leftBalance + 2));
                    }
                    SetBalance(node, (sbyte)(rootBalance + 1));
                }
                return left;
            }
        }

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

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

```
        /// <para>The current.</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override TElement GetPrevious(TElement node)
        {
            var current = GetLeft(node);
            if (GetLeftIsChild(node))
            {
                return GetRightest(current);
            }
            return current;
        }

        /// <summary>
        /// <para>
        /// Detaches the core using the specified root.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="root">
        /// <para>The root.</para>
        /// <para></para>
        /// </param>
        /// <param name="node">
        /// <para>The node.</para>
        /// <para></para>
        /// </param>
        /// <exception cref="InvalidOperationException">
        /// <para>Cannot find a node.</para>
        /// <para></para>
        /// </exception>
        /// <exception cref="InvalidOperationException">
        /// <para>Cannot find a node.</para>
        /// <para></para>
        /// </exception>
        protected override void DetachCore(ref TElement root, TElement node)
        {
            unchecked
            {
#if USEARRAYPOOL
                var path = ArrayPool.Allocate<TElement>(MaxPath);
                var pathPosition = 0;
                path[pathPosition++] = default;
#else
                var path = new TElement[_maxPath];
                var pathPosition = 1;
#endif
                var currentNode = root;
                while (true)
                {
                    if (FirstIsToTheLeftOfSecond(node, currentNode))
                    {
                        if (!GetLeftIsChild(currentNode))
                        {
                            throw new InvalidOperationException("Cannot find a node.");
                        }
                        DecrementSize(currentNode);
                        path[pathPosition++] = currentNode;
                        currentNode = GetLeft(currentNode);
                    }
                    else if (FirstIsToTheRightOfSecond(node, currentNode))
                    {
                        if (!GetRightIsChild(currentNode))
                        {
                            throw new InvalidOperationException("Cannot find a node.");
                        }
                        DecrementSize(currentNode);
                        path[pathPosition++] = currentNode;
                        currentNode = GetRight(currentNode);
                    }
                    else
                    {
                        break;
                    }
                }
                var parent = path[--pathPosition];
                var balanceNode = parent;
```

```
                var isLeftNode = !AreEqual(parent, default) && AreEqual(currentNode,
                ↪  GetLeft(parent));
                if (!GetLeftIsChild(currentNode))
                {
                    if (!GetRightIsChild(currentNode)) // node has no children
                    {
                        if (AreEqual(parent, default))
                        {
                            root = Zero;
                        }
                        else if (isLeftNode)
                        {
                            SetLeftIsChild(parent, false);
                            SetLeft(parent, GetLeft(currentNode));
                            IncrementBalance(parent);
                        }
                        else
                        {
                            SetRightIsChild(parent, false);
                            SetRight(parent, GetRight(currentNode));
                            DecrementBalance(parent);
                        }
                    }
                    else // node has a right child
                    {
                        var successor = GetNext(currentNode);
                        SetLeft(successor, GetLeft(currentNode));
                        var right = GetRight(currentNode);
                        if (AreEqual(parent, default))
                        {
                            root = right;
                        }
                        else if (isLeftNode)
                        {
                            SetLeft(parent, right);
                            IncrementBalance(parent);
                        }
                        else
                        {
                            SetRight(parent, right);
                            DecrementBalance(parent);
                        }
                    }
                }
                else // node has a left child
                {
                    if (!GetRightIsChild(currentNode))
                    {
                        var predecessor = GetPrevious(currentNode);
                        SetRight(predecessor, GetRight(currentNode));
                        var leftValue = GetLeft(currentNode);
                        if (AreEqual(parent, default))
                        {
                            root = leftValue;
                        }
                        else if (isLeftNode)
                        {
                            SetLeft(parent, leftValue);
                            IncrementBalance(parent);
                        }
                        else
                        {
                            SetRight(parent, leftValue);
                            DecrementBalance(parent);
                        }
                    }
                    else // node has a both children (left and right)
                    {
                        var predecessor = GetLeft(currentNode);
                        var successor = GetRight(currentNode);
                        var successorParent = currentNode;
                        int previousPathPosition = ++pathPosition;
                        // find the immediately next node (and its parent)
                        while (GetLeftIsChild(successor))
                        {
                            path[++pathPosition] = successorParent = successor;
                            successor = GetLeft(successor);
                            if (!AreEqual(successorParent, currentNode))
```

```
                    {
                        DecrementSize(successorParent);
                    }
                }
                path[previousPathPosition] = successor;
                balanceNode = path[pathPosition];
                // remove 'successor' from the tree
                if (!AreEqual(successorParent, currentNode))
                {
                    if (!GetRightIsChild(successor))
                    {
                        SetLeftIsChild(successorParent, false);
                    }
                    else
                    {
                        SetLeft(successorParent, GetRight(successor));
                    }
                    IncrementBalance(successorParent);
                    SetRightIsChild(successor, true);
                    SetRight(successor, GetRight(currentNode));
                }
                else
                {
                    DecrementBalance(currentNode);
                }
                // set the predecessor's successor link to point to the right place
                while (GetRightIsChild(predecessor))
                {
                    predecessor = GetRight(predecessor);
                }
                SetRight(predecessor, successor);
                // prepare 'successor' to replace 'node'
                var left = GetLeft(currentNode);
                SetLeftIsChild(successor, true);
                SetLeft(successor, left);
                SetBalance(successor, GetBalance(currentNode));
                FixSize(successor);
                if (AreEqual(parent, default))
                {
                    root = successor;
                }
                else if (isLeftNode)
                {
                    SetLeft(parent, successor);
                }
                else
                {
                    SetRight(parent, successor);
                }
            }
        }
        // restore balance
        if (!AreEqual(balanceNode, default))
        {
            while (true)
            {
                var balanceParent = path[--pathPosition];
                isLeftNode = !AreEqual(balanceParent, default) && AreEqual(balanceNode,
                ↪  GetLeft(balanceParent));
                var currentNodeBalance = GetBalance(balanceNode);
                if (currentNodeBalance < -1 || currentNodeBalance > 1)
                {
                    balanceNode = Balance(balanceNode);
                    if (AreEqual(balanceParent, default))
                    {
                        root = balanceNode;
                    }
                    else if (isLeftNode)
                    {
                        SetLeft(balanceParent, balanceNode);
                    }
                    else
                    {
                        SetRight(balanceParent, balanceNode);
                    }
                }
                currentNodeBalance = GetBalance(balanceNode);
                if (currentNodeBalance != 0 || AreEqual(balanceParent, default))
```

```
                        {
                            break;
                        }
                        if (isLeftNode)
                        {
                            IncrementBalance(balanceParent);
                        }
                        else
                        {
                            DecrementBalance(balanceParent);
                        }
                        balanceNode = balanceParent;
                    }
                }
                ClearNode(node);
#if USEARRAYPOOL
                ArrayPool.Free(path);
#endif
            }
        }

        /// <summary>
        /// <para>
        /// Clears the node using the specified node.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="node">
        /// <para>The node.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void ClearNode(TElement node)
        {
            SetLeft(node, Zero);
            SetRight(node, Zero);
            SetSize(node, Zero);
            SetLeftIsChild(node, false);
            SetRightIsChild(node, false);
            SetBalance(node, 0);
        }
    }
}
```

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

```
//#define ENABLE_TREE_AUTO_DEBUG_AND_VALIDATION

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

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

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

```
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected abstract ref TElement GetLeftReference(TElement node);

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

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

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

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

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

```
112         /// </summary>
113         /// <param name="node">
114         /// <para>The node.</para>
115         /// <para></para>
116         /// </param>
117         /// <param name="left">
118         /// <para>The left.</para>
119         /// <para></para>
120         /// </param>
121         [MethodImpl(MethodImplOptions.AggressiveInlining)]
122         protected abstract void SetLeft(TElement node, TElement left);
123 
124         /// <summary>
125         /// <para>
126         /// Sets the right using the specified node.
127         /// </para>
128         /// <para></para>
129         /// </summary>
130         /// <param name="node">
131         /// <para>The node.</para>
132         /// <para></para>
133         /// </param>
134         /// <param name="right">
135         /// <para>The right.</para>
136         /// <para></para>
137         /// </param>
138         [MethodImpl(MethodImplOptions.AggressiveInlining)]
139         protected abstract void SetRight(TElement node, TElement right);
140 
141         /// <summary>
142         /// <para>
143         /// Sets the size using the specified node.
144         /// </para>
145         /// <para></para>
146         /// </summary>
147         /// <param name="node">
148         /// <para>The node.</para>
149         /// <para></para>
150         /// </param>
151         /// <param name="size">
152         /// <para>The size.</para>
153         /// <para></para>
154         /// </param>
155         [MethodImpl(MethodImplOptions.AggressiveInlining)]
156         protected abstract void SetSize(TElement node, TElement size);
157 
158         /// <summary>
159         /// <para>
160         /// Determines whether this instance first is to the left of second.
161         /// </para>
162         /// <para></para>
163         /// </summary>
164         /// <param name="first">
165         /// <para>The first.</para>
166         /// <para></para>
167         /// </param>
168         /// <param name="second">
169         /// <para>The second.</para>
170         /// <para></para>
171         /// </param>
172         /// <returns>
173         /// <para>The bool</para>
174         /// <para></para>
175         /// </returns>
176         [MethodImpl(MethodImplOptions.AggressiveInlining)]
177         protected abstract bool FirstIsToTheLeftOfSecond(TElement first, TElement second);
178 
179         /// <summary>
180         /// <para>
181         /// Determines whether this instance first is to the right of second.
182         /// </para>
183         /// <para></para>
184         /// </summary>
185         /// <param name="first">
186         /// <para>The first.</para>
187         /// <para></para>
188         /// </param>
189         /// <param name="second">
```

```
        /// <para>The second.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <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>
        /// <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>
        /// </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>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected virtual TElement GetRightest(TElement current)
        {
            var currentRight = GetRight(current);
            while (!EqualToZero(currentRight))
            {
                current = currentRight;
                currentRight = GetRight(current);
            }
            return current;
        }

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

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

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

```

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

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

        /// <summary>
        /// <para>
        /// Attaches the root.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="root">
        /// <para>The root.</para>
        /// <para></para>
        /// </param>
        /// <param name="node">
        /// <para>The node.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public void Attach(ref TElement root, TElement node)
        {
#if ENABLE_TREE_AUTO_DEBUG_AND_VALIDATION
            ValidateSizes(root);
            Debug.WriteLine("--BeforeAttach--");
            Debug.WriteLine(PrintNodes(root));
```

```
576             Debug.WriteLine("----------------");
577             var sizeBefore = GetSize(root);
578 #endif
579             if (EqualToZero(root))
580             {
581                 SetSize(node, One);
582                 root = node;
583                 return;
584             }
585             AttachCore(ref root, node);
586 #if ENABLE_TREE_AUTO_DEBUG_AND_VALIDATION
587             Debug.WriteLine("--AfterAttach--");
588             Debug.WriteLine(PrintNodes(root));
589             Debug.WriteLine("----------------");
590             ValidateSizes(root);
591             var sizeAfter = GetSize(root);
592             if (!AreEqual(Arithmetic.Increment(sizeBefore), sizeAfter))
593             {
594                 throw new InvalidOperationException("Tree was broken after attach.");
595             }
596 #endif
597         }
598 
599         /// <summary>
600         /// <para>
601         /// Attaches the core using the specified root.
602         /// </para>
603         /// <para></para>
604         /// </summary>
605         /// <param name="root">
606         /// <para>The root.</para>
607         /// <para></para>
608         /// </param>
609         /// <param name="node">
610         /// <para>The node.</para>
611         /// <para></para>
612         /// </param>
613         protected abstract void AttachCore(ref TElement root, TElement node);
614 
615         /// <summary>
616         /// <para>
617         /// Detaches the root.
618         /// </para>
619         /// <para></para>
620         /// </summary>
621         /// <param name="root">
622         /// <para>The root.</para>
623         /// <para></para>
624         /// </param>
625         /// <param name="node">
626         /// <para>The node.</para>
627         /// <para></para>
628         /// </param>
629         [MethodImpl(MethodImplOptions.AggressiveInlining)]
630         public void Detach(ref TElement root, TElement node)
631         {
632 #if ENABLE_TREE_AUTO_DEBUG_AND_VALIDATION
633             ValidateSizes(root);
634             Debug.WriteLine("--BeforeDetach--");
635             Debug.WriteLine(PrintNodes(root));
636             Debug.WriteLine("----------------");
637             var sizeBefore = GetSize(root);
638             if (EqualToZero(root))
639             {
640                 throw new InvalidOperationException($"Элемент с {node} не содержится в
                    ↪ дереве.");
641             }
642 #endif
643             DetachCore(ref root, node);
644 #if ENABLE_TREE_AUTO_DEBUG_AND_VALIDATION
645             Debug.WriteLine("--AfterDetach--");
646             Debug.WriteLine(PrintNodes(root));
647             Debug.WriteLine("----------------");
648             ValidateSizes(root);
649             var sizeAfter = GetSize(root);
650             if (!AreEqual(Arithmetic.Decrement(sizeBefore), sizeAfter))
651             {
652                 throw new InvalidOperationException("Tree was broken after detach.");
```

```
            }
#endif
        }

        /// <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">
        /// <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)
        {
            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>
        /// <para>
        /// Prints the node value using the specified node.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="node">
        /// <para>The node.</para>
        /// <para></para>
        /// </param>
        /// <param name="sb">
        /// <para>The sb.</para>
        /// <para></para>
        /// </param>
        protected abstract void PrintNodeValue(TElement node, StringBuilder sb);
    }
}
```

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

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

namespace Platform.Collections.Methods.Tests
{
    /// <summary>
    /// <para>
    /// Represents the recursionless size balanced tree.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <seealso cref="RecursionlessSizeBalancedTreeMethods{TElement}"/>
    public class RecursionlessSizeBalancedTree<TElement> :
    ↪ RecursionlessSizeBalancedTreeMethods<TElement>
    {
        private struct TreeElement
        {
            /// <summary>
            /// <para>
            /// The size.
            /// </para>
            /// <para></para>
            /// </summary>
            public TElement Size;
            /// <summary>
            /// <para>
            /// The left.
            /// </para>
            /// <para></para>
            /// </summary>
            public TElement Left;
            /// <summary>
            /// <para>
            /// The right.
            /// </para>
            /// <para></para>
            /// </summary>
            public TElement Right;
        }

        private readonly TreeElement[] _elements;
        private TElement _allocated;

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

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

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

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

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

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

        /// <summary>
        /// <para>
        /// Determines whether this instance is empty.
        /// </para>
```

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

        /// <summary>
        /// <para>
        /// Determines whether this instance first is to the left of second.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="first">
        /// <para>The first.</para>
        /// <para></para>
        /// </param>
        /// <param name="second">
        /// <para>The second.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        protected override bool FirstIsToTheLeftOfSecond(TElement first, TElement second) =>
        ↪  Comparer.Compare(first, second) < 0;

        /// <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>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        protected override bool FirstIsToTheRightOfSecond(TElement first, TElement second) =>
        ↪  Comparer.Compare(first, second) > 0;

        /// <summary>
        /// <para>
        /// Gets the left reference using the specified node.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="node">
        /// <para>The node.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The ref element</para>
        /// <para></para>
        /// </returns>
        protected override ref TElement GetLeftReference(TElement node) => ref
        ↪  GetElement(node).Left;

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

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

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

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

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

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

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

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

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

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

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

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

namespace Platform.Collections.Methods.Tests
{
    /// <summary>
    /// <para>
    /// Represents the size balanced tree.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <seealso cref="SizeBalancedTreeMethods{TElement}"/>
    public class SizeBalancedTree<TElement> : SizeBalancedTreeMethods<TElement>
    {
        private struct TreeElement
        {
            /// <summary>
            /// <para>
            /// The size.
            /// </para>
```

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

        private readonly TreeElement[] _elements;
        private TElement _allocated;

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

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

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

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

```

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

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

        /// <summary>
        /// <para>
        /// Determines whether this instance first is to the left of second.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="first">
        /// <para>The first.</para>
        /// <para></para>
        /// </param>
        /// <param name="second">
        /// <para>The second.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        protected override bool FirstIsToTheLeftOfSecond(TElement first, TElement second) =>
        ↪  Comparer.Compare(first, second) < 0;

        /// <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>
```

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

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

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

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

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

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

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

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

```
1  using System;
2  using System.Collections.Generic;
3  using System.Text;
4  using Platform.Numbers;
5  using Platform.Collections.Methods.Trees;
6  using Platform.Converters;
7
8  namespace Platform.Collections.Methods.Tests
9  {
10     /// <summary>
11     /// <para>
12     /// Represents the sized and threaded avl balanced tree.
13     /// </para>
14     /// <para></para>
15     /// </summary>
16     /// <seealso cref="SizedAndThreadedAVLBalancedTreeMethods{TElement}"/>
17     public class SizedAndThreadedAVLBalancedTree<TElement> :
       ↪ SizedAndThreadedAVLBalancedTreeMethods<TElement>
18     {
19         private struct TreeElement
20         {
21             /// <summary>
22             /// <para>
23             /// The size.
24             /// </para>
25             /// <para></para>
26             /// </summary>
27             public TElement Size;
28             /// <summary>
29             /// <para>
30             /// The left.
31             /// </para>
32             /// <para></para>
33             /// </summary>
34             public TElement Left;
35             /// <summary>
36             /// <para>
37             /// The right.
38             /// </para>
39             /// <para></para>
40             /// </summary>
41             public TElement Right;
42             /// <summary>
43             /// <para>
44             /// The balance.
45             /// </para>
46             /// <para></para>
47             /// </summary>
48             public sbyte Balance;
49             /// <summary>
50             /// <para>
51             /// The left is child.
52             /// </para>
53             /// <para></para>
54             /// </summary>
55             public bool LeftIsChild;
56             /// <summary>
57             /// <para>
58             /// The right is child.
59             /// </para>
60             /// <para></para>
61             /// </summary>
62             public bool RightIsChild;
63         }
64
65         private readonly TreeElement[] _elements;
66         private TElement _allocated;
67
68         /// <summary>
69         /// <para>
```

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

```
                }
            }
        }

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

        /// <summary>
        /// <para>
        /// Determines whether this instance first is to the left of second.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="first">
        /// <para>The first.</para>
        /// <para></para>
        /// </param>
        /// <param name="second">
        /// <para>The second.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        protected override bool FirstIsToTheLeftOfSecond(TElement first, TElement second) =>
        ↪  Comparer.Compare(first, second) < 0;

        /// <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>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        protected override bool FirstIsToTheRightOfSecond(TElement first, TElement second) =>
        ↪  Comparer.Compare(first, second) > 0;

        /// <summary>
        /// <para>
        /// Gets the balance using the specified node.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="node">
        /// <para>The node.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The sbyte</para>
        /// <para></para>
        /// </returns>
        protected override sbyte GetBalance(TElement node) => GetElement(node).Balance;
```

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

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

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

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

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

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

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

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

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

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

```
1  using System;
2  using System.Collections.Generic;
3  using Xunit;
4  using Platform.Collections.Methods.Trees;
5  using Platform.Converters;
6
7  namespace Platform.Collections.Methods.Tests
8  {
9      /// <summary>
10     /// <para>
11     /// Represents the test extensions.
12     /// </para>
13     /// <para></para>
14     /// </summary>
15     public static class TestExtensions
16     {
17         /// <summary>
18         /// <para>
19         /// Tests the multiple creations and deletions using the specified tree.
20         /// </para>
21         /// <para></para>
22         /// </summary>
23         /// <typeparam name="TElement">
24         /// <para>The element.</para>
25         /// <para></para>
26         /// </typeparam>
27         /// <param name="tree">
28         /// <para>The tree.</para>
29         /// <para></para>
30         /// </param>
31         /// <param name="allocate">
32         /// <para>The allocate.</para>
33         /// <para></para>
34         /// </param>
35         /// <param name="free">
36         /// <para>The free.</para>
37         /// <para></para>
38         /// </param>
39         /// <param name="root">
40         /// <para>The root.</para>
41         /// <para></para>
42         /// </param>
43         /// <param name="treeCount">
44         /// <para>The tree count.</para>
45         /// <para></para>
46         /// </param>
47         /// <param name="maximumOperationsPerCycle">
48         /// <para>The maximum operations per cycle.</para>
49         /// <para></para>
50         /// </param>
51         public static void TestMultipleCreationsAndDeletions<TElement>(this
            ↪  SizedBinaryTreeMethodsBase<TElement> tree, Func<TElement> allocate, Action<TElement>
            ↪  free, ref TElement root, Func<TElement> treeCount, int maximumOperationsPerCycle)
52         {
53             for (var N = 1; N < maximumOperationsPerCycle; N++)
54             {
55                 var currentCount = 0;
56                 for (var i = 0; i < N; i++)
57                 {
58                     var node = allocate();
59                     tree.Attach(ref root, node);
60                     currentCount++;
61                     Assert.Equal(currentCount, (int)UncheckedConverter<TElement,
                        ↪  int>.Default.Convert(treeCount()));
62                 }
63                 for (var i = 1; i <= N; i++)
64                 {
65                     TElement node = UncheckedConverter<int, TElement>.Default.Convert(i);
66                     if (tree.Contains(node, root))
67                     {
68                         tree.Detach(ref root, node);
69                         free(node);
```

```
                        currentCount--;
                        Assert.Equal(currentCount, (int)UncheckedConverter<TElement,
                        ↪  int>.Default.Convert(treeCount()));
                    }
                }
            }
        }

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

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

```
using Xunit;

namespace Platform.Collections.Methods.Tests
{
```

```
    /// <summary>
    /// <para>
    /// Represents the trees tests.
    /// </para>
    /// <para></para>
    /// </summary>
    public static class TreesTests
    {
        private const int _n = 500;

        /// <summary>
        /// <para>
        /// Tests that recursionless size balanced tree multiple attach and detach test.
        /// </para>
        /// <para></para>
        /// </summary>
        [Fact]
        public static void RecursionlessSizeBalancedTreeMultipleAttachAndDetachTest()
        {
            var recursionlessSizeBalancedTree = new RecursionlessSizeBalancedTree<uint>(10000);
            recursionlessSizeBalancedTree.TestMultipleCreationsAndDeletions(recursionlessSizeBal
            ↪  ancedTree.Allocate, recursionlessSizeBalancedTree.Free, ref
            ↪  recursionlessSizeBalancedTree.Root, () => recursionlessSizeBalancedTree.Count,
            ↪  _n);
        }

        /// <summary>
        /// <para>
        /// Tests that size balanced tree multiple attach and detach test.
        /// </para>
        /// <para></para>
        /// </summary>
        [Fact]
        public static void SizeBalancedTreeMultipleAttachAndDetachTest()
        {
            var sizeBalancedTree = new SizeBalancedTree<uint>(10000);
            sizeBalancedTree.TestMultipleCreationsAndDeletions(sizeBalancedTree.Allocate,
            ↪  sizeBalancedTree.Free, ref sizeBalancedTree.Root, () => sizeBalancedTree.Count,
            ↪  _n);
        }

        /// <summary>
        /// <para>
        /// Tests that sized and threaded avl balanced tree multiple attach and detach test.
        /// </para>
        /// <para></para>
        /// </summary>
        [Fact]
        public static void SizedAndThreadedAVLBalancedTreeMultipleAttachAndDetachTest()
        {
            var avlTree = new SizedAndThreadedAVLBalancedTree<uint>(10000);
            avlTree.TestMultipleCreationsAndDeletions(avlTree.Allocate, avlTree.Free, ref
            ↪  avlTree.Root, () => avlTree.Count, _n);
        }

        /// <summary>
        /// <para>
        /// Tests that recursionless size balanced tree multiple random attach and detach test.
        /// </para>
        /// <para></para>
        /// </summary>
        [Fact]
        public static void RecursionlessSizeBalancedTreeMultipleRandomAttachAndDetachTest()
        {
            var recursionlessSizeBalancedTree = new RecursionlessSizeBalancedTree<uint>(10000);
            recursionlessSizeBalancedTree.TestMultipleRandomCreationsAndDeletions(ref
            ↪  recursionlessSizeBalancedTree.Root, () => recursionlessSizeBalancedTree.Count,
            ↪  _n);
        }

        /// <summary>
        /// <para>
        /// Tests that size balanced tree multiple random attach and detach test.
        /// </para>
        /// <para></para>
        /// </summary>
        [Fact]
        public static void SizeBalancedTreeMultipleRandomAttachAndDetachTest()
```

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

Index