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 range between minimum and maximum values.</para>
    /// <para>Представляет диапазон между минимальным и максимальным значениями.</para>
    /// </summary>
    /// <remarks>
    /// <para>Based on <a href="http://stackoverflow.com/questions/5343006/is-there-a-c-sharp-type-for-representing-an-integer-range">the question at StackOverflow</a>.</para>
    /// <para>Основано на <a href="http://stackoverflow.com/questions/5343006/is-there-a-c-sharp-type-for-representing-an-integer-range">вопросе в StackOverflow</a>.</para>
    /// </remarks>
    public abstract class GenericCollectionMethodsBase<TElement>
    {
        /// <summary>
        /// <para>Presents the Range in readable format.</para>
        /// <para>Представляет диапазон в удобном для чтения формате.</para>
        /// </summary>
        /// <returns><para>String representation of the Range.</para><para>Строковое представление диапазона.</para></returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected virtual TElement GetZero() => default;

        /// <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 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>
```

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

```
125         /// <para>Представляет диапазон в удобном для чтения формате.</para>
126         /// </summary>
127         /// <returns><para>String representation of the Range.</para><para>Строковое
            ↪ представление диапазона.</para></returns>
128         [MethodImpl(MethodImplOptions.AggressiveInlining)]
129         protected virtual TElement Add(TElement first, TElement second) =>
            ↪ Arithmetic<TElement>.Add(first, second);
130
131         /// <summary>
132         /// <para>Presents the Range in readable format.</para>
133         /// <para>Представляет диапазон в удобном для чтения формате.</para>
134         /// </summary>
135         /// <returns><para>String representation of the Range.</para><para>Строковое
            ↪ представление диапазона.</para></returns>
136         [MethodImpl(MethodImplOptions.AggressiveInlining)]
137         protected virtual TElement Subtract(TElement first, TElement second) =>
            ↪ Arithmetic<TElement>.Subtract(first, second);
138
139         protected readonly TElement Zero;
140         protected readonly TElement One;
141         protected readonly TElement Two;
142         protected readonly EqualityComparer<TElement> EqualityComparer;
143         protected readonly Comparer<TElement> Comparer;
144
145         /// <summary>
146         /// <para>Presents the Range in readable format.</para>
147         /// <para>Представляет диапазон в удобном для чтения формате.</para>
148         /// </summary>
149         /// <returns><para>String representation of the Range.</para><para>Строковое
            ↪ представление диапазона.</para></returns>
150         protected GenericCollectionMethodsBase()
151         {
152             EqualityComparer = EqualityComparer<TElement>.Default;
153             Comparer = Comparer<TElement>.Default;
154             Zero = GetZero(); //-V3068
155             One = Increment(Zero); //-V3068
156             Two = Increment(One); //-V3068
157         }
158     }
159 }
```

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      public abstract class AbsoluteCircularDoublyLinkedListMethods<TElement> :
       ↪ AbsoluteDoublyLinkedListMethodsBase<TElement>
6      {
7          public void AttachBefore(TElement baseElement, TElement newElement)
8          {
9              var baseElementPrevious = GetPrevious(baseElement);
10             SetPrevious(newElement, baseElementPrevious);
11             SetNext(newElement, baseElement);
12             if (AreEqual(baseElement, GetFirst()))
13             {
14                 SetFirst(newElement);
15             }
16             SetNext(baseElementPrevious, newElement);
17             SetPrevious(baseElement, newElement);
18             IncrementSize();
19         }
20
21         public void AttachAfter(TElement baseElement, TElement newElement)
22         {
23             var baseElementNext = GetNext(baseElement);
24             SetPrevious(newElement, baseElement);
25             SetNext(newElement, baseElementNext);
26             if (AreEqual(baseElement, GetLast()))
27             {
28                 SetLast(newElement);
29             }
30             SetPrevious(baseElementNext, newElement);
31             SetNext(baseElement, newElement);
32             IncrementSize();
33         }
34
35         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);
            }
        }

        public void AttachAsLast(TElement element)
        {
            var last = GetLast();
            if (EqualToZero(last))
            {
                AttachAsFirst(element);
            }
            else
            {
                AttachAfter(last, element);
            }
        }

        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
{
    public abstract class AbsoluteDoublyLinkedListMethodsBase<TElement> :
    ↪ DoublyLinkedListMethodsBase<TElement>
    {
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected abstract TElement GetFirst();

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected abstract TElement GetLast();

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected abstract TElement GetSize();

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

```

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected abstract void SetLast(TElement element);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected abstract void SetSize(TElement size);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected void IncrementSize() => SetSize(Increment(GetSize()));

        [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
{
    public abstract class AbsoluteOpenDoublyLinkedListMethods<TElement> :
    ↪ AbsoluteDoublyLinkedListMethodsBase<TElement>
    {
        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();
        }

        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();
        }

        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);
            }
        }

        public void AttachAsLast(TElement element)
        {
            var last = GetLast();
            if (EqualToZero(last))
            {
```

```
                AttachAsFirst(element);
            }
            else
            {
                AttachAfter(last, element);
            }
        }

        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>
    {
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected abstract TElement GetPrevious(TElement element);

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected abstract void SetPrevious(TElement element, TElement previous);

        [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
{
    public abstract class RelativeCircularDoublyLinkedListMethods<TElement> :
    ↪  RelativeDoublyLinkedListMethodsBase<TElement>
    {
        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);
        }

        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);
        }

        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);
            }
        }

        public void AttachAsLast(TElement headElement, TElement element)
        {
            var last = GetLast(headElement);
            if (EqualToZero(last))
            {
                AttachAsFirst(headElement, element);
            }
            else
            {
                AttachAfter(headElement, last, element);
            }
        }

        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
{
    public abstract class RelativeDoublyLinkedListMethodsBase<TElement> :
    ↪ DoublyLinkedListMethodsBase<TElement>
    {
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected abstract TElement GetFirst(TElement headElement);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected abstract TElement GetLast(TElement headElement);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected abstract TElement GetSize(TElement headElement);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected abstract void SetFirst(TElement headElement, TElement element);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected abstract void SetLast(TElement headElement, TElement element);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected abstract void SetSize(TElement headElement, TElement size);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected void IncrementSize(TElement headElement) => SetSize(headElement,
        ↪ Increment(GetSize(headElement)));

        [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
{
    public abstract class RelativeOpenDoublyLinkedListMethods<TElement> :
    ↪ RelativeDoublyLinkedListMethodsBase<TElement>
    {
        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);
        }

        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);
        }

        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);
            }
        }

        public void AttachAsLast(TElement headElement, TElement element)
        {
            var last = GetLast(headElement);
            if (EqualToZero(last))
            {
                AttachAsFirst(headElement, element);
            }
            else
            {
                AttachAfter(headElement, last, element);
            }
        }

        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
{
    public abstract class RecursionlessSizeBalancedTreeMethods<TElement> :
    ↪ SizedBinaryTreeMethodsBase<TElement>
    {
        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;
                    }
                }
            }
        }

        protected override void DetachCore(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
                {
                    var decrementedLeftSize = Decrement(leftSize);
                    if (GreaterThan(GetSizeOrZero(GetRightOrDefault(right)),
                    ↪  decrementedLeftSize))
                    {
                        LeftRotate(ref root);
                    }
                    else if (GreaterThan(GetSizeOrZero(GetLeftOrDefault(right)),
                    ↪  decrementedLeftSize))
                    {
                        RightRotate(ref right);
                        LeftRotate(ref root);
                    }
                    else
                    {
                        DecrementSize(root);
                        root = ref left;
                    }
                }
                else if (FirstIsToTheRightOfSecond(node, root)) // node.Key greater than root.Key
                {
                    var decrementedRightSize = Decrement(rightSize);
                    if (GreaterThan(GetSizeOrZero(GetLeftOrDefault(left)), decrementedRightSize))
                    {
                        RightRotate(ref root);
                    }
                    else if (GreaterThan(GetSizeOrZero(GetRightOrDefault(left)),
                    ↪  decrementedRightSize))
                    {
                        LeftRotate(ref left);
                        RightRotate(ref root);
                    }
                    else
                    {
                        DecrementSize(root);
                        root = ref right;
                    }
                }
                else // key equals to root.Key
                {
                    if (GreaterThanZero(leftSize) && GreaterThanZero(rightSize))
                    {
                        TElement replacement;
                        if (GreaterThan(leftSize, rightSize))
                        {
                            replacement = GetRightest(left);
                            DetachCore(ref left, replacement);
                        }
                        else
                        {
                            replacement = GetLeftest(right);
                            DetachCore(ref right, replacement);
                        }
```

```
                        SetLeft(replacement, left);
                        SetRight(replacement, right);
                        SetSize(replacement, Add(leftSize, rightSize));
                        root = replacement;
                    }
                    else if (GreaterThanZero(leftSize))
                    {
                        root = left;
                    }
                    else if (GreaterThanZero(rightSize))
                    {
                        root = right;
                    }
                    else
                    {
                        root = Zero;
                    }
                    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
{
    public abstract class SizeBalancedTreeMethods<TElement> :
    ↪ SizedBinaryTreeMethodsBase<TElement>
    {
        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);
                }
            }
        }

        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

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override TElement GetRightest(TElement current)
        {
            var currentRight = GetRightOrDefault(current);
            while (!EqualToZero(currentRight))
```

```
            {
                current = currentRight;
                currentRight = GetRightOrDefault(current);
            }
            return current;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override TElement GetLeftest(TElement current)
        {
            var currentLeft = GetLeftOrDefault(current);
            while (!EqualToZero(currentLeft))
            {
                current = currentLeft;
                currentLeft = GetLeftOrDefault(current);
            }
            return current;
        }

        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;
        }

        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));
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected void IncrementBalance(TElement node) => SetBalance(node,
        ↪  (sbyte)(GetBalance(node) + 1));

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected void DecrementBalance(TElement node) => SetBalance(node,
        ↪  (sbyte)(GetBalance(node) - 1));

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override TElement GetLeftOrDefault(TElement node) => GetLeftIsChild(node) ?
        ↪  GetLeft(node) : default;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override TElement GetRightOrDefault(TElement node) => GetRightIsChild(node) ?
        ↪  GetRight(node) : default;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected abstract bool GetLeftIsChild(TElement node);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected abstract void SetLeftIsChild(TElement node, bool value);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected abstract bool GetRightIsChild(TElement node);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected abstract void SetRightIsChild(TElement node, bool value);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected abstract sbyte GetBalance(TElement node);
```

```
104
105         [MethodImpl(MethodImplOptions.AggressiveInlining)]
106         protected abstract void SetBalance(TElement node, sbyte value);
107
108         protected override void AttachCore(ref TElement root, TElement node)
109         {
110             unchecked
111             {
112                 // TODO: Check what is faster to use simple array or array from array pool
113                 // TODO: Try to use stackalloc as an optimization (requires code generation,
                    ↪  because of generics)
114 #if USEARRAYPOOL
115                 var path = ArrayPool.Allocate<TElement>(MaxPath);
116                 var pathPosition = 0;
117                 path[pathPosition++] = default;
118 #else
119                 var path = new TElement[_maxPath];
120                 var pathPosition = 1;
121 #endif
122                 var currentNode = root;
123                 while (true)
124                 {
125                     if (FirstIsToTheLeftOfSecond(node, currentNode))
126                     {
127                         if (GetLeftIsChild(currentNode))
128                         {
129                             IncrementSize(currentNode);
130                             path[pathPosition++] = currentNode;
131                             currentNode = GetLeft(currentNode);
132                         }
133                         else
134                         {
135                             // Threads
136                             SetLeft(node, GetLeft(currentNode));
137                             SetRight(node, currentNode);
138                             SetLeft(currentNode, node);
139                             SetLeftIsChild(currentNode, true);
140                             DecrementBalance(currentNode);
141                             SetSize(node, One);
142                             FixSize(currentNode); // Should be incremented already
143                             break;
144                         }
145                     }
146                     else if (FirstIsToTheRightOfSecond(node, currentNode))
147                     {
148                         if (GetRightIsChild(currentNode))
149                         {
150                             IncrementSize(currentNode);
151                             path[pathPosition++] = currentNode;
152                             currentNode = GetRight(currentNode);
153                         }
154                         else
155                         {
156                             // Threads
157                             SetRight(node, GetRight(currentNode));
158                             SetLeft(node, currentNode);
159                             SetRight(currentNode, node);
160                             SetRightIsChild(currentNode, true);
161                             IncrementBalance(currentNode);
162                             SetSize(node, One);
163                             FixSize(currentNode); // Should be incremented already
164                             break;
165                         }
166                     }
167                     else
168                     {
169                         throw new InvalidOperationException("Node with the same key already
                            ↪  attached to a tree.");
170                     }
171                 }
172                 // Restore balance. This is the goodness of a non-recursive
173                 // implementation, when we are done with balancing we 'break'
174                 // the loop and we are done.
175                 while (true)
176                 {
177                     var parent = path[--pathPosition];
178                     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;
            }
        }

        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;
            }
        }

        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;
            }
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override TElement GetNext(TElement node)
        {
            var current = GetRight(node);
            if (GetRightIsChild(node))
            {
                return GetLeftest(current);
            }
            return current;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override TElement GetPrevious(TElement node)
        {
            var current = GetLeft(node);
            if (GetLeftIsChild(node))
            {
                return GetRightest(current);
            }
            return current;
        }

        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
            }
        }

        [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
{
    public abstract class SizedBinaryTreeMethodsBase<TElement> :
    ↪ GenericCollectionMethodsBase<TElement>
    {
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected abstract ref TElement GetLeftReference(TElement node);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected abstract ref TElement GetRightReference(TElement node);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected abstract TElement GetLeft(TElement node);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected abstract TElement GetRight(TElement node);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected abstract TElement GetSize(TElement node);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected abstract void SetLeft(TElement node, TElement left);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected abstract void SetRight(TElement node, TElement right);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected abstract void SetSize(TElement node, TElement size);

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected abstract bool FirstIsToTheRightOfSecond(TElement first, TElement second);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected virtual TElement GetLeftOrDefault(TElement node) => AreEqual(node, default) ?
        ↪  default : GetLeft(node);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
```

```
49          protected virtual TElement GetRightOrDefault(TElement node) => AreEqual(node, default) ?
            ↪ default : GetRight(node);
50
51          [MethodImpl(MethodImplOptions.AggressiveInlining)]
52          protected void IncrementSize(TElement node) => SetSize(node, Increment(GetSize(node)));
53
54          [MethodImpl(MethodImplOptions.AggressiveInlining)]
55          protected void DecrementSize(TElement node) => SetSize(node, Decrement(GetSize(node)));
56
57          [MethodImpl(MethodImplOptions.AggressiveInlining)]
58          protected TElement GetLeftSize(TElement node) => GetSizeOrZero(GetLeftOrDefault(node));
59
60          [MethodImpl(MethodImplOptions.AggressiveInlining)]
61          protected TElement GetRightSize(TElement node) => GetSizeOrZero(GetRightOrDefault(node));
62
63          [MethodImpl(MethodImplOptions.AggressiveInlining)]
64          protected TElement GetSizeOrZero(TElement node) => EqualToZero(node) ? Zero :
            ↪ GetSize(node);
65
66          [MethodImpl(MethodImplOptions.AggressiveInlining)]
67          protected void FixSize(TElement node) => SetSize(node, Increment(Add(GetLeftSize(node),
            ↪ GetRightSize(node))));
68
69          [MethodImpl(MethodImplOptions.AggressiveInlining)]
70          protected void LeftRotate(ref TElement root) => root = LeftRotate(root);
71
72          [MethodImpl(MethodImplOptions.AggressiveInlining)]
73          protected TElement LeftRotate(TElement root)
74          {
75              var right = GetRight(root);
76  #if ENABLE_TREE_AUTO_DEBUG_AND_VALIDATION
77              if (EqualToZero(right))
78              {
79                  throw new InvalidOperationException("Right is null.");
80              }
81  #endif
82              SetRight(root, GetLeft(right));
83              SetLeft(right, root);
84              SetSize(right, GetSize(root));
85              FixSize(root);
86              return right;
87          }
88
89          [MethodImpl(MethodImplOptions.AggressiveInlining)]
90          protected void RightRotate(ref TElement root) => root = RightRotate(root);
91
92          [MethodImpl(MethodImplOptions.AggressiveInlining)]
93          protected TElement RightRotate(TElement root)
94          {
95              var left = GetLeft(root);
96  #if ENABLE_TREE_AUTO_DEBUG_AND_VALIDATION
97              if (EqualToZero(left))
98              {
99                  throw new InvalidOperationException("Left is null.");
100             }
101 #endif
102             SetLeft(root, GetRight(left));
103             SetRight(left, root);
104             SetSize(left, GetSize(root));
105             FixSize(root);
106             return left;
107         }
108
109         [MethodImpl(MethodImplOptions.AggressiveInlining)]
110         protected virtual TElement GetRightest(TElement current)
111         {
112             var currentRight = GetRight(current);
113             while (!EqualToZero(currentRight))
114             {
115                 current = currentRight;
116                 currentRight = GetRight(current);
117             }
118             return current;
119         }
120
121         [MethodImpl(MethodImplOptions.AggressiveInlining)]
122         protected virtual TElement GetLeftest(TElement current)
123         {
124             var currentLeft = GetLeft(current);
```

```
            while (!EqualToZero(currentLeft))
            {
                current = currentLeft;
                currentLeft = GetLeft(current);
            }
            return current;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected virtual TElement GetNext(TElement node) => GetLeftest(GetRight(node));

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected virtual TElement GetPrevious(TElement node) => GetRightest(GetLeft(node));

        [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;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected virtual void ClearNode(TElement node)
        {
            SetLeft(node, Zero);
            SetRight(node, Zero);
            SetSize(node, Zero);
        }

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

        protected abstract void AttachCore(ref TElement root, TElement node);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public void Detach(ref TElement root, TElement node)
        {
#if ENABLE_TREE_AUTO_DEBUG_AND_VALIDATION
```

```
204             ValidateSizes(root);
205             Debug.WriteLine("--BeforeDetach--");
206             Debug.WriteLine(PrintNodes(root));
207             Debug.WriteLine("----------------");
208             var sizeBefore = GetSize(root);
209             if (EqualToZero(root))
210             {
211                 throw new InvalidOperationException($"Элемент с {node} не содержится в
                    ↪  дереве.");
212             }
213 #endif
214             DetachCore(ref root, node);
215 #if ENABLE_TREE_AUTO_DEBUG_AND_VALIDATION
216             Debug.WriteLine("--AfterDetach--");
217             Debug.WriteLine(PrintNodes(root));
218             Debug.WriteLine("----------------");
219             ValidateSizes(root);
220             var sizeAfter = GetSize(root);
221             if (!AreEqual(Arithmetic.Decrement(sizeBefore), sizeAfter))
222             {
223                 throw new InvalidOperationException("Tree was broken after detach.");
224             }
225 #endif
226         }
227 
228         protected abstract void DetachCore(ref TElement root, TElement node);
229 
230         public void FixSizes(TElement node)
231         {
232             if (AreEqual(node, default))
233             {
234                 return;
235             }
236             FixSizes(GetLeft(node));
237             FixSizes(GetRight(node));
238             FixSize(node);
239         }
240 
241         public void ValidateSizes(TElement node)
242         {
243             if (AreEqual(node, default))
244             {
245                 return;
246             }
247             var size = GetSize(node);
248             var leftSize = GetLeftSize(node);
249             var rightSize = GetRightSize(node);
250             var expectedSize = Arithmetic.Increment(Arithmetic.Add(leftSize, rightSize));
251             if (!AreEqual(size, expectedSize))
252             {
253                 throw new InvalidOperationException($"Size of {node} is not valid. Expected
                    ↪  size: {expectedSize}, actual size: {size}.");
254             }
255             ValidateSizes(GetLeft(node));
256             ValidateSizes(GetRight(node));
257         }
258 
259         public void ValidateSize(TElement node)
260         {
261             var size = GetSize(node);
262             var leftSize = GetLeftSize(node);
263             var rightSize = GetRightSize(node);
264             var expectedSize = Arithmetic.Increment(Arithmetic.Add(leftSize, rightSize));
265             if (!AreEqual(size, expectedSize))
266             {
267                 throw new InvalidOperationException($"Size of {node} is not valid. Expected
                    ↪  size: {expectedSize}, actual size: {size}.");
268             }
269         }
270 
271         public string PrintNodes(TElement node)
272         {
273             var sb = new StringBuilder();
274             PrintNodes(node, sb);
275             return sb.ToString();
276         }
277 
278         [MethodImpl(MethodImplOptions.AggressiveInlining)]
```

```
279         public void PrintNodes(TElement node, StringBuilder sb) => PrintNodes(node, sb, 0);
280
281         public void PrintNodes(TElement node, StringBuilder sb, int level)
282         {
283             if (AreEqual(node, default))
284             {
285                 return;
286             }
287             PrintNodes(GetLeft(node), sb, level + 1);
288             PrintNode(node, sb, level);
289             sb.AppendLine();
290             PrintNodes(GetRight(node), sb, level + 1);
291         }
292
293         public string PrintNode(TElement node)
294         {
295             var sb = new StringBuilder();
296             PrintNode(node, sb);
297             return sb.ToString();
298         }
299
300         [MethodImpl(MethodImplOptions.AggressiveInlining)]
301         protected void PrintNode(TElement node, StringBuilder sb) => PrintNode(node, sb, 0);
302
303         protected virtual void PrintNode(TElement node, StringBuilder sb, int level)
304         {
305             sb.Append('\t', level);
306             sb.Append(node);
307             PrintNodeValue(node, sb);
308             sb.Append(' ');
309             sb.Append('s');
310             sb.Append(GetSize(node));
311         }
312
313         protected abstract void PrintNodeValue(TElement node, StringBuilder sb);
314     }
315 }
```

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

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

```
        }

        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;
                }
            }
        }

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

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

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

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

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

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

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

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

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

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

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

        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
{
    public class SizeBalancedTree<TElement> : SizeBalancedTreeMethods<TElement>
    {
        private struct TreeElement
        {
            public TElement Size;
            public TElement Left;
            public TElement Right;
        }

        private readonly TreeElement[] _elements;
        private TElement _allocated;

        public TElement Root;
```

```

        public TElement Count => GetSizeOrZero(Root);

        public SizeBalancedTree(int capacity) => (_elements, _allocated) = (new
        ↪  TreeElement[capacity], One);

        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.");
            }
        }

        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;
                }
            }
        }

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

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

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

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

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

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

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

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

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

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

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

        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)];
    }
}
```

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

```
using System;
using System.Collections.Generic;
using System.Text;
```

```
using Platform.Numbers;
using Platform.Collections.Methods.Trees;
using Platform.Converters;

namespace Platform.Collections.Methods.Tests
{
    public class SizedAndThreadedAVLBalancedTree<TElement> :
    ↪ SizedAndThreadedAVLBalancedTreeMethods<TElement>
    {
        private struct TreeElement
        {
            public TElement Size;
            public TElement Left;
            public TElement Right;
            public sbyte Balance;
            public bool LeftIsChild;
            public bool RightIsChild;
        }

        private readonly TreeElement[] _elements;
        private TElement _allocated;

        public TElement Root;

        public TElement Count => GetSizeOrZero(Root);

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

        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.");
            }
        }

        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;
                }
            }
        }

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

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

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

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

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

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

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

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

```

```
78          protected override ref TElement GetRightReference(TElement node) => ref
            ↪ GetElement(node).Right;
79
80          protected override TElement GetRight(TElement node) => GetElement(node).Right;
81
82          protected override TElement GetSize(TElement node) => GetElement(node).Size;
83
84          protected override void PrintNodeValue(TElement node, StringBuilder sb) =>
            ↪ sb.Append(node);
85
86          protected override void SetBalance(TElement node, sbyte value) =>
            ↪ GetElement(node).Balance = value;
87
88          protected override void SetLeft(TElement node, TElement left) => GetElement(node).Left =
            ↪ left;
89
90          protected override void SetLeftIsChild(TElement node, bool value) =>
            ↪ GetElement(node).LeftIsChild = value;
91
92          protected override void SetRight(TElement node, TElement right) =>
            ↪ GetElement(node).Right = right;
93
94          protected override void SetRightIsChild(TElement node, bool value) =>
            ↪ GetElement(node).RightIsChild = value;
95
96          protected override void SetSize(TElement node, TElement size) => GetElement(node).Size =
            ↪ size;
97
98          private ref TreeElement GetElement(TElement node) => ref
            ↪ _elements[UncheckedConverter<TElement, long>.Default.Convert(node)];
99      }
100 }
```

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

```
1   using System;
2   using System.Collections.Generic;
3   using Xunit;
4   using Platform.Collections.Methods.Trees;
5   using Platform.Converters;
6
7   namespace Platform.Collections.Methods.Tests
8   {
9       public static class TestExtensions
10      {
11          public static void TestMultipleCreationsAndDeletions<TElement>(this
            ↪ SizedBinaryTreeMethodsBase<TElement> tree, Func<TElement> allocate, Action<TElement>
            ↪ free, ref TElement root, Func<TElement> treeCount, int maximumOperationsPerCycle)
12          {
13              for (var N = 1; N < maximumOperationsPerCycle; N++)
14              {
15                  var currentCount = 0;
16                  for (var i = 0; i < N; i++)
17                  {
18                      var node = allocate();
19                      tree.Attach(ref root, node);
20                      currentCount++;
21                      Assert.Equal(currentCount, (int)UncheckedConverter<TElement,
                        ↪ int>.Default.Convert(treeCount()));
22                  }
23                  for (var i = 1; i <= N; i++)
24                  {
25                      TElement node = UncheckedConverter<int, TElement>.Default.Convert(i);
26                      if (tree.Contains(node, root))
27                      {
28                          tree.Detach(ref root, node);
29                          free(node);
30                          currentCount--;
31                          Assert.Equal(currentCount, (int)UncheckedConverter<TElement,
                            ↪ int>.Default.Convert(treeCount()));
32                      }
33                  }
34              }
35          }
36
37          public static void TestMultipleRandomCreationsAndDeletions<TElement>(this
            ↪ SizedBinaryTreeMethodsBase<TElement> tree, ref TElement root, Func<TElement>
            ↪ treeCount, int maximumOperationsPerCycle)
38          {
39              var random = new System.Random(0);
```

```
            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
{
    public static class TreesTests
    {
        private const int _n = 500;

        [Fact]
        public static void RecursionlessSizeBalancedTreeMultipleAttachAndDetachTest()
        {
            var recursionlessSizeBalancedTree = new RecursionlessSizeBalancedTree<uint>(10000);
            recursionlessSizeBalancedTree.TestMultipleCreationsAndDeletions(recursionlessSizeBal
            ↪  ancedTree.Allocate, recursionlessSizeBalancedTree.Free, ref
            ↪  recursionlessSizeBalancedTree.Root, () => recursionlessSizeBalancedTree.Count,
            ↪  _n);
        }

        [Fact]
        public static void SizeBalancedTreeMultipleAttachAndDetachTest()
        {
            var sizeBalancedTree = new SizeBalancedTree<uint>(10000);
            sizeBalancedTree.TestMultipleCreationsAndDeletions(sizeBalancedTree.Allocate,
            ↪  sizeBalancedTree.Free, ref sizeBalancedTree.Root, () => sizeBalancedTree.Count,
            ↪  _n);
        }

        [Fact]
        public static void SizedAndThreadedAVLBalancedTreeMultipleAttachAndDetachTest()
        {
            var avlTree = new SizedAndThreadedAVLBalancedTree<uint>(10000);
            avlTree.TestMultipleCreationsAndDeletions(avlTree.Allocate, avlTree.Free, ref
            ↪  avlTree.Root, () => avlTree.Count, _n);
        }

        [Fact]
        public static void RecursionlessSizeBalancedTreeMultipleRandomAttachAndDetachTest()
        {
            var recursionlessSizeBalancedTree = new RecursionlessSizeBalancedTree<uint>(10000);
            recursionlessSizeBalancedTree.TestMultipleRandomCreationsAndDeletions(ref
            ↪  recursionlessSizeBalancedTree.Root, () => recursionlessSizeBalancedTree.Count,
            ↪  _n);
        }

```

```
37          [Fact]
38          public static void SizeBalancedTreeMultipleRandomAttachAndDetachTest()
39          {
40              var sizeBalancedTree = new SizeBalancedTree<uint>(10000);
41              sizeBalancedTree.TestMultipleRandomCreationsAndDeletions(ref sizeBalancedTree.Root,
                ↪  () => sizeBalancedTree.Count, _n);
42          }
43
44          [Fact]
45          public static void SizedAndThreadedAVLBalancedTreeMultipleRandomAttachAndDetachTest()
46          {
47              var avlTree = new SizedAndThreadedAVLBalancedTree<uint>(10000);
48              avlTree.TestMultipleRandomCreationsAndDeletions(ref avlTree.Root, () =>
                ↪  avlTree.Count, _n);
49          }
50      }
51  }
```

Index