LinksPlatform's Platform.Collections.Methods Class Library

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

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

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

namespace Platform.Collections.Methods
{
    /// <summary>
    /// <para>Represents a 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>
        /// <returns><para>String representation of the Range.</para><para>Строковое
        ↪  представление диапазона.</para></returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected virtual bool GreaterOrEqualThanZero(TElement value) => Comparer.Compare(value,
        ↪  Zero) >= 0;

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

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

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

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

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

        /// <summary>
        /// <para>Presents the Range in readable format.</para>
        /// <para>Представляет диапазон в удобном для чтения формате.</para>
        /// </summary>
        /// <returns><para>String representation of the Range.</para><para>Строковое
        ↪  представление диапазона.</para></returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected virtual TElement Increment(TElement value) =>
        ↪  Arithmetic<TElement>.Increment(value);

        /// <summary>
        /// <para>Presents the Range in readable format.</para>
        /// <para>Представляет диапазон в удобном для чтения формате.</para>
        /// </summary>
        /// <returns><para>String representation of the Range.</para><para>Строковое
        ↪  представление диапазона.</para></returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected virtual TElement Decrement(TElement value) =>
        ↪  Arithmetic<TElement>.Decrement(value);

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

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

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

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

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

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

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

        /// <summary>
        /// <para>Presents the Range in readable format.</para>
        /// <para>Представляет диапазон в удобном для чтения формате.</para>
        /// </summary>
        /// <returns><para>String representation of the Range.</para><para>Строковое
        ↪  представление диапазона.</para></returns>
        protected GenericCollectionMethodsBase()
        {
            EqualityComparer = EqualityComparer<TElement>.Default;
            Comparer = Comparer<TElement>.Default;
            Zero = GetZero(); //-V3068
            One = Increment(Zero); //-V3068
            Two = Increment(One); //-V3068
        }
    }
}
```

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

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

namespace Platform.Collections.Methods.Lists
{
    public abstract class AbsoluteCircularDoublyLinkedListMethods<TElement> :
    ↪  AbsoluteDoublyLinkedListMethodsBase<TElement>
    {
        public void AttachBefore(TElement baseElement, TElement newElement)
        {
            var baseElementPrevious = GetPrevious(baseElement);
            SetPrevious(newElement, baseElementPrevious);
            SetNext(newElement, baseElement);
            if (AreEqual(baseElement, GetFirst()))
            {
                SetFirst(newElement);
            }
            SetNext(baseElementPrevious, newElement);
            SetPrevious(baseElement, newElement);
            IncrementSize();
        }

        public void AttachAfter(TElement baseElement, TElement newElement)
        {
            var baseElementNext = GetNext(baseElement);
            SetPrevious(newElement, baseElement);
            SetNext(newElement, baseElementNext);
            if (AreEqual(baseElement, GetLast()))
            {
                SetLast(newElement);
            }
            SetPrevious(baseElementNext, newElement);
            SetNext(baseElement, newElement);
            IncrementSize();
        }

        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);

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

        protected override void AttachCore(ref TElement root, TElement node)
        {
            unchecked
            {
                // TODO: Check what is faster to use simple array or array from array pool
                // TODO: Try to use stackalloc as an optimization (requires code generation,
                ↪  because of generics)
#if USEARRAYPOOL
                var path = ArrayPool.Allocate<TElement>(MaxPath);
                var pathPosition = 0;
                path[pathPosition++] = default;
#else
                var path = new TElement[_maxPath];
                var pathPosition = 1;
#endif
                var currentNode = root;
                while (true)
                {
                    if (FirstIsToTheLeftOfSecond(node, currentNode))
                    {
                        if (GetLeftIsChild(currentNode))
                        {
                            IncrementSize(currentNode);
                            path[pathPosition++] = currentNode;
                            currentNode = GetLeft(currentNode);
                        }
                        else
                        {
                            // Threads
                            SetLeft(node, GetLeft(currentNode));
                            SetRight(node, currentNode);
                            SetLeft(currentNode, node);
                            SetLeftIsChild(currentNode, true);
                            DecrementBalance(currentNode);
                            SetSize(node, One);
                            FixSize(currentNode); // Should be incremented already
                            break;
                        }
                    }
                    else if (FirstIsToTheRightOfSecond(node, currentNode))
                    {
                        if (GetRightIsChild(currentNode))
                        {
                            IncrementSize(currentNode);
                            path[pathPosition++] = currentNode;
                            currentNode = GetRight(currentNode);
                        }
                        else
                        {
                            // Threads
                            SetRight(node, GetRight(currentNode));
                            SetLeft(node, currentNode);
```

```
                                SetRight(currentNode, node);
                                SetRightIsChild(currentNode, true);
                                IncrementBalance(currentNode);
                                SetSize(node, One);
                                FixSize(currentNode); // Should be incremented already
                                break;
                            }
                        }
                        else
                        {
                            throw new InvalidOperationException("Node with the same key already
                             ↪  attached to a tree.");
                        }
                    }
                    // Restore balance. This is the goodness of a non-recursive
                    // implementation, when we are done with balancing we 'break'
                    // the loop and we are done.
                    while (true)
                    {
                        var parent = path[--pathPosition];
                        var isLeftNode = !AreEqual(parent, default) && AreEqual(currentNode,
                         ↪  GetLeft(parent));
                        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)]
        protected virtual TElement GetRightOrDefault(TElement node) => AreEqual(node, default) ?
        ↪  default : GetRight(node);

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected void DecrementSize(TElement node) => SetSize(node, Decrement(GetSize(node)));

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected TElement GetLeftSize(TElement node) => GetSizeOrZero(GetLeftOrDefault(node));

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected TElement GetRightSize(TElement node) => GetSizeOrZero(GetRightOrDefault(node));

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected TElement GetSizeOrZero(TElement node) => EqualToZero(node) ? Zero :
        ↪  GetSize(node);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected void FixSize(TElement node) => SetSize(node, Increment(Add(GetLeftSize(node),
        ↪  GetRightSize(node))));

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected void LeftRotate(ref TElement root) => root = LeftRotate(root);

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected void RightRotate(ref TElement root) => root = RightRotate(root);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected TElement RightRotate(TElement root)
        {
            var left = GetLeft(root);
#if ENABLE_TREE_AUTO_DEBUG_AND_VALIDATION
            if (EqualToZero(left))
            {
                throw new InvalidOperationException("Left is null.");
            }
#endif
```

```
            SetLeft(root, GetRight(left));
            SetRight(left, root);
            SetSize(left, GetSize(root));
            FixSize(root);
            return left;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected virtual TElement GetRightest(TElement current)
        {
            var currentRight = GetRight(current);
            while (!EqualToZero(currentRight))
            {
                current = currentRight;
                currentRight = GetRight(current);
            }
            return current;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected virtual TElement GetLeftest(TElement current)
        {
            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
            ValidateSizes(root);
            Debug.WriteLine("--BeforeDetach--");
            Debug.WriteLine(PrintNodes(root));
            Debug.WriteLine("----------------");
            var sizeBefore = GetSize(root);
            if (EqualToZero(root))
            {
                throw new InvalidOperationException($"Элемент с {node} не содержится в дереве.");
            }
#endif
            DetachCore(ref root, node);
#if ENABLE_TREE_AUTO_DEBUG_AND_VALIDATION
            Debug.WriteLine("--AfterDetach--");
            Debug.WriteLine(PrintNodes(root));
            Debug.WriteLine("----------------");
            ValidateSizes(root);
            var sizeAfter = GetSize(root);
            if (!AreEqual(Arithmetic.Decrement(sizeBefore), sizeAfter))
            {
                throw new InvalidOperationException("Tree was broken after detach.");
            }
#endif
        }

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

        public void FixSizes(TElement node)
        {
            if (AreEqual(node, default))
            {
                return;
            }
            FixSizes(GetLeft(node));
            FixSizes(GetRight(node));
            FixSize(node);
        }

        public void ValidateSizes(TElement node)
        {
            if (AreEqual(node, default))
            {
                return;
            }
            var size = GetSize(node);
            var leftSize = GetLeftSize(node);
            var rightSize = GetRightSize(node);
            var expectedSize = Arithmetic.Increment(Arithmetic.Add(leftSize, rightSize));
            if (!AreEqual(size, expectedSize))
            {
                throw new InvalidOperationException($"Size of {node} is not valid. Expected size: {expectedSize}, actual size: {size}.");
            }
            ValidateSizes(GetLeft(node));
            ValidateSizes(GetRight(node));
```

```csharp
        }

        public void ValidateSize(TElement node)
        {
            var size = GetSize(node);
            var leftSize = GetLeftSize(node);
            var rightSize = GetRightSize(node);
            var expectedSize = Arithmetic.Increment(Arithmetic.Add(leftSize, rightSize));
            if (!AreEqual(size, expectedSize))
            {
                throw new InvalidOperationException($"Size of {node} is not valid. Expected
                ↪  size: {expectedSize}, actual size: {size}.");
            }
        }

        public string PrintNodes(TElement node)
        {
            var sb = new StringBuilder();
            PrintNodes(node, sb);
            return sb.ToString();
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public void PrintNodes(TElement node, StringBuilder sb) => PrintNodes(node, sb, 0);

        public void PrintNodes(TElement node, StringBuilder sb, int level)
        {
            if (AreEqual(node, default))
            {
                return;
            }
            PrintNodes(GetLeft(node), sb, level + 1);
            PrintNode(node, sb, level);
            sb.AppendLine();
            PrintNodes(GetRight(node), sb, level + 1);
        }

        public string PrintNode(TElement node)
        {
            var sb = new StringBuilder();
            PrintNode(node, sb);
            return sb.ToString();
        }

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

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

        protected abstract void PrintNodeValue(TElement node, StringBuilder sb);
    }
}
```

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

```csharp
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 RecursionlessSizeBalancedTree<TElement> :
    ↪  RecursionlessSizeBalancedTreeMethods<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 RecursionlessSizeBalancedTree(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)];
    }
```

```
}
```

1.14 ./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;

        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 SetBalance(TElement node, sbyte value) =>
        ↪  GetElement(node).Balance = value;

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

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

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

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

        protected override void SetSize(TElement node, TElement size) => GetElement(node).Size =
        ↪  size;

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

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

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

namespace Platform.Collections.Methods.Tests
{
    public static class TestExtensions
    {
        public static void TestMultipleCreationsAndDeletions<TElement>(this
        ↪  SizedBinaryTreeMethodsBase<TElement> tree, Func<TElement> allocate, Action<TElement>
        ↪  free, ref TElement root, Func<TElement> treeCount, int maximumOperationsPerCycle)
        {
            for (var N = 1; N < maximumOperationsPerCycle; N++)
            {
                var currentCount = 0;
                for (var i = 0; i < N; i++)
                {
                    var node = allocate();
                    tree.Attach(ref root, node);
```

```csharp
                    currentCount++;
                    Assert.Equal(currentCount, (int)UncheckedConverter<TElement, int>.Default.Convert(treeCount()));
                }
                for (var i = 1; i <= N; i++)
                {
                    TElement node = UncheckedConverter<int, TElement>.Default.Convert(i);
                    if (tree.Contains(node, root))
                    {
                        tree.Detach(ref root, node);
                        free(node);
                        currentCount--;
                        Assert.Equal(currentCount, (int)UncheckedConverter<TElement, int>.Default.Convert(treeCount()));
                    }
                }
            }
        }

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

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

```csharp
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(recursionlessSizeBalancedTree.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);
        }

        [Fact]
        public static void SizeBalancedTreeMultipleRandomAttachAndDetachTest()
        {
            var sizeBalancedTree = new SizeBalancedTree<uint>(10000);
            sizeBalancedTree.TestMultipleRandomCreationsAndDeletions(ref sizeBalancedTree.Root, () => sizeBalancedTree.Count, _n);
        }

        [Fact]
        public static void SizedAndThreadedAVLBalancedTreeMultipleRandomAttachAndDetachTest()
        {
            var avlTree = new SizedAndThreadedAVLBalancedTree<uint>(10000);
            avlTree.TestMultipleRandomCreationsAndDeletions(ref avlTree.Root, () => avlTree.Count, _n);
        }
    }
}
```

Index