# LinksPlatform's Platform.Collections.Methods Class Library

## ./GenericCollectionMethodsBase.cs

```csharp
using System;
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
{
    public unsafe abstract class GenericCollectionMethodsBase<TElement>
    {
        private static readonly EqualityComparer<TElement> _equalityComparer =
        ↪  EqualityComparer<TElement>.Default;
        private static readonly Comparer<TElement> _comparer = Comparer<TElement>.Default;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected virtual TElement GetZero() => Integer<TElement>.Zero;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected virtual TElement GetOne() => Integer<TElement>.One;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected virtual TElement GetTwo() => Integer<TElement>.Two;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected virtual bool ValueEqualToZero(IntPtr pointer) => _equalityComparer.Equals(Syst
        ↪  em.Runtime.CompilerServices.Unsafe.Read<TElement>((void*)pointer),
        ↪  GetZero());

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected virtual bool EqualToZero(TElement value) => _equalityComparer.Equals(value,
        ↪  GetZero());

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected virtual bool IsEquals(TElement first, TElement second) =>
        ↪  _equalityComparer.Equals(first, second);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected virtual bool GreaterThanZero(TElement value) => _comparer.Compare(value,
        ↪  GetZero()) > 0;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected virtual bool GreaterThan(TElement first, TElement second) =>
        ↪  _comparer.Compare(first, second) > 0;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected virtual bool GreaterOrEqualThanZero(TElement value) =>
        ↪  _comparer.Compare(value, GetZero()) >= 0;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected virtual bool GreaterOrEqualThan(TElement first, TElement second) =>
        ↪  _comparer.Compare(first, second) >= 0;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected virtual bool LessOrEqualThanZero(TElement value) => _comparer.Compare(value,
        ↪  GetZero()) <= 0;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected virtual bool LessOrEqualThan(TElement first, TElement second) =>
        ↪  _comparer.Compare(first, second) <= 0;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected virtual bool LessThanZero(TElement value) => _comparer.Compare(value,
        ↪  GetZero()) < 0;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected virtual bool LessThan(TElement first, TElement second) =>
        ↪  _comparer.Compare(first, second) < 0;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected virtual TElement Increment(TElement value) =>
        ↪  Arithmetic<TElement>.Increment(value);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected virtual TElement Decrement(TElement value) =>
        ↪  Arithmetic<TElement>.Decrement(value);
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected virtual TElement Add(TElement first, TElement second) =>
         ↪  Arithmetic<TElement>.Add(first, second);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected virtual TElement Subtract(TElement first, TElement second) =>
         ↪  Arithmetic<TElement>.Subtract(first, second);
    }
}
```

./Lists/CircularDoublyLinkedListMethods.cs

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

namespace Platform.Collections.Methods.Lists
{
    public abstract class CircularDoublyLinkedListMethods<TElement> :
     ↪  DoublyLinkedListMethodsBase<TElement>
    {
        public void AttachBefore(TElement baseElement, TElement newElement)
        {
            var baseElementPrevious = GetPrevious(baseElement);
            SetPrevious(newElement, baseElementPrevious);
            SetNext(newElement, baseElement);
            if (IsEquals(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 (IsEquals(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 (IsEquals(elementNext, element))
            {
                SetFirst(GetZero());
                SetLast(GetZero());
            }
            else
            {
                SetNext(elementPrevious, elementNext);
                SetPrevious(elementNext, elementPrevious);
                if (IsEquals(element, GetFirst()))
                {
                    SetFirst(elementNext);
                }
                if (IsEquals(element, GetLast()))
                {
                    SetLast(elementPrevious);
                }
            }
            SetPrevious(element, GetZero());
            SetNext(element, GetZero());
            DecrementSize();
        }
    }
}
```

./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 GetFirst();

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

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

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

        [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 SetPrevious(TElement element, TElement previous);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected abstract void SetNext(TElement element, TElement next);

        [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()));
    }
}
```

./Lists/OpenDoublyLinkedListMethods.cs

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

namespace Platform.Collections.Methods.Lists
{
    public abstract class OpenDoublyLinkedListMethods<TElement> :
    ↪   DoublyLinkedListMethodsBase<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, GetZero());
                SetNext(element, GetZero());
                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, GetZero());
            SetNext(element, GetZero());
            DecrementSize();
        }
    }
}
```

./Trees/SizeBalancedTreeMethods2.cs

```
using System;

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

namespace Platform.Collections.Methods.Trees
{
    /// <summary>
    /// Experimental implementation, don't use it yet.
    /// </summary>
    public unsafe abstract class SizeBalancedTreeMethods2<TElement> : SizedBinaryTreeMethodsBase<TElement>
    {
        protected override void AttachCore(IntPtr root, TElement newNode)
        {
            if (ValueEqualToZero(root))
            {
                System.Runtime.CompilerServices.Unsafe.Write((void*)root, newNode);
                IncrementSize(System.Runtime.CompilerServices.Unsafe.Read<TElement>((void*)root));
            }
            else
            {
                IncrementSize(System.Runtime.CompilerServices.Unsafe.Read<TElement>((void*)root));
                if (FirstIsToTheLeftOfSecond(newNode, System.Runtime.CompilerServices.Unsafe.Read<TElement>((void*)root)))
                {
                    AttachCore(GetLeftPointer(System.Runtime.CompilerServices.Unsafe.Read<TElement>((void*)root)), newNode);
                    LeftMaintain(root);
                }
                else
                {
                    AttachCore(GetRightPointer(System.Runtime.CompilerServices.Unsafe.Read<TElement>((void*)root)), newNode);
                    RightMaintain(root);
                }
            }
        }

        protected override void DetachCore(IntPtr root, TElement nodeToDetach)
        {
            if (ValueEqualToZero(root))
            {
                return;
            }
            var currentNode = root;
            var parent = IntPtr.Zero; /* Изначально зануление, так как родителя может и не быть (Корень дерева). */
            var replacementNode = GetZero();
            while (!IsEquals(System.Runtime.CompilerServices.Unsafe.Read<TElement>((void*)currentNode), nodeToDetach))
            {
                SetSize(System.Runtime.CompilerServices.Unsafe.Read<TElement>((void*)currentNode), Decrement(GetSize(System.Runtime.CompilerServices.Unsafe.Read<TElement>((void*)currentNode))));
                if (FirstIsToTheLeftOfSecond(nodeToDetach, System.Runtime.CompilerServices.Unsafe.Read<TElement>((void*)currentNode)))
                {
                    parent = currentNode;
                    currentNode = GetLeftPointer(System.Runtime.CompilerServices.Unsafe.Read<TElement>((void*)currentNode));
                }
                else if (FirstIsToTheRightOfSecond(nodeToDetach, System.Runtime.CompilerServices.Unsafe.Read<TElement>((void*)currentNode)))
                {
                    parent = currentNode;
                    currentNode = GetRightPointer(System.Runtime.CompilerServices.Unsafe.Read<TElement>((void*)currentNode));
                }
                else
                {
                    throw new InvalidOperationException("Duplicate link found in the tree.");
                }
            }
            if (!ValueEqualToZero(GetLeftPointer(nodeToDetach)) && !ValueEqualToZero(GetRightPointer(nodeToDetach)))
            {
                var minNode = GetRightValue(nodeToDetach);
                while (!EqualToZero(GetLeftValue(minNode)))
                {
                    minNode = GetLeftValue(minNode); /* Передвигаемся до минимума */
                }
                DetachCore(GetRightPointer(nodeToDetach), minNode);
                SetLeft(minNode, GetLeftValue(nodeToDetach));
                if (!ValueEqualToZero(GetRightPointer(nodeToDetach)))
                {
                    SetRight(minNode, GetRightValue(nodeToDetach));
                    SetSize(minNode, Increment(Add(GetSize(GetLeftValue(nodeToDetach)), GetSize(GetRightValue(nodeToDetach)))));
                }
                else
                {
                    SetSize(minNode, Increment(GetSize(GetLeftValue(nodeToDetach))));
                }
                replacementNode = minNode;
            }
            else if (!ValueEqualToZero(GetLeftPointer(nodeToDetach)))
            {
                replacementNode = GetLeftValue(nodeToDetach);
            }
            else if (!ValueEqualToZero(GetRightPointer(nodeToDetach)))
            {
                replacementNode = GetRightValue(nodeToDetach);
            }
            if (parent == IntPtr.Zero)
            {
                System.Runtime.CompilerServices.Unsafe.Write((void*)root, replacementNode);
            }
            else if (IsEquals(GetLeftValue(System.Runtime.CompilerServices.Unsafe.Read<TElement>((void*)parent)), nodeToDetach))
            {
                SetLeft(System.Runtime.CompilerServices.Unsafe.Read<TElement>((void*)parent), replacementNode);
            }
            else if (IsEquals(GetRightValue(System.Runtime.CompilerServices.Unsafe.Read<TElement>((void*)parent)), nodeToDetach))
            {
                SetRight(System.Runtime.CompilerServices.Unsafe.Read<TElement>((void*)parent), replacementNode);
            }
            ClearNode(nodeToDetach);
        }

        private void LeftMaintain(IntPtr root)
        {
            if (!ValueEqualToZero(root))
            {
                var rootLeftNode = GetLeftPointer(System.Runtime.CompilerServices.Unsafe.Read<TElement>((void*)root));
                if (!ValueEqualToZero(rootLeftNode))
                {
                    var rootRightNode = GetRightPointer(System.Runtime.CompilerServices.Unsafe.Read<TElement>((void*)root));
                    var rootLeftNodeLeftNode = GetLeftPointer(System.Runtime.CompilerServices.Unsafe.Read<TElement>((void*)rootLeftNode));
                    if (!ValueEqualToZero(rootLeftNodeLeftNode) &&
                        (ValueEqualToZero(rootRightNode) || GreaterThan(GetSize(System.Runtime.CompilerServices.Unsafe.Read<TElement>((void*)rootLeftNodeLeftNode)), GetSize(System.Runtime.CompilerServices.Unsafe.Read<TElement>((void*)rootRightNode)))))
                    {
                        RightRotate(root);
                    }
                    else
                    {
                        var rootLeftNodeRightNode = GetRightPointer(System.Runtime.CompilerServices.Unsafe.Read<TElement>((void*)rootLeftNode));
                        if (!ValueEqualToZero(rootLeftNodeRightNode) &&
                            (ValueEqualToZero(rootRightNode) || GreaterThan(GetSize(System.Runtime.CompilerServices.Unsafe.Read<TElement>((void*)rootLeftNodeRightNode)), GetSize(System.Runtime.CompilerServices.Unsafe.Read<TElement>((void*)rootRightNode)))))
                        {
                            LeftRotate(GetLeftPointer(System.Runtime.CompilerServices.Unsafe.Read<TElement>((void*)root)));
                            RightRotate(root);
                        }
                        else
                        {
                            return;
                        }
                    }
                    LeftMaintain(GetLeftPointer(System.Runtime.CompilerServices.Unsafe.Read<TElement>((void*)root)));
                    RightMaintain(GetRightPointer(System.Runtime.CompilerServices.Unsafe.Read<TElement>((void*)root)));
                    LeftMaintain(root);
                    RightMaintain(root);
                }
            }
        }

        private void RightMaintain(IntPtr root)
        {
            if (!ValueEqualToZero(root))
            {
                var rootRightNode = GetRightPointer(System.Runtime.CompilerServices.Unsafe.Read<TElement>((void*)root));
                if (!ValueEqualToZero(rootRightNode))
                {
                    var rootLeftNode = GetLeftPointer(System.Runtime.CompilerServices.Unsafe.Read<TElement>((void*)root));
                    var rootRightNodeRightNode = GetRightPointer(System.Runtime.CompilerServices.Unsafe.Read<TElement>((void*)rootRightNode));
                    if (!ValueEqualToZero(rootRightNodeRightNode) &&
                        (ValueEqualToZero(rootLeftNode) || GreaterThan(GetSize(System.Runtime.CompilerServices.Unsafe.Read<TElement>((void*)rootRightNodeRightNode)), GetSize(System.Runtime.CompilerServices.Unsafe.Read<TElement>((void*)rootLeftNode)))))
                    {
                        LeftRotate(root);
                    }
                    else
                    {
                        var rootRightNodeLeftNode = GetLeftPointer(System.Runtime.CompilerServices.Unsafe.Read<TElement>((void*)rootRightNode));
                        if (!ValueEqualToZero(rootRightNodeLeftNode) &&
                            (ValueEqualToZero(rootLeftNode) || GreaterThan(GetSize(System.Runtime.CompilerServices.Unsafe.Read<TElement>((void*)rootRightNodeLeftNode)), GetSize(System.Runtime.CompilerServices.Unsafe.Read<TElement>((void*)rootLeftNode)))))
                        {
                            RightRotate(GetRightPointer(System.Runtime.CompilerServices.Unsafe.Read<TElement>((void*)root)));
                            LeftRotate(root);
                        }
                        else
                        {
                            return;
                        }
                    }
                    LeftMaintain(GetLeftPointer(System.Runtime.CompilerServices.Unsafe.Read<TElement>((void*)root)));
                    RightMaintain(GetRightPointer(System.Runtime.CompilerServices.Unsafe.Read<TElement>((void*)root)));
                    LeftMaintain(root);
                    RightMaintain(root);
                }
            }
        }
    }
}
```

./Trees/SizeBalancedTreeMethods.cs

```
using System;

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

namespace Platform.Collections.Methods.Trees
{
    public unsafe abstract class SizeBalancedTreeMethods<TElement> : SizedBinaryTreeMethodsBase<TElement>
    {
        protected override void AttachCore(IntPtr root, TElement node)
        {
            while (true)
            {
                var left = GetLeftPointer(System.Runtime.CompilerServices.Unsafe.Read<TElement>((void*)root));
                var leftSize = GetSizeOrZero(System.Runtime.CompilerServices.Unsafe.Read<TElement>((void*)left));
                var right = GetRightPointer(System.Runtime.CompilerServices.Unsafe.Read<TElement>((void*)root));
                var rightSize = GetSizeOrZero(System.Runtime.CompilerServices.Unsafe.Read<TElement>((void*)right));
                if (FirstIsToTheLeftOfSecond(node, System.Runtime.CompilerServices.Unsafe.Read<TElement>((void*)root))) // node.Key less than root.Key
                {
                    if (EqualToZero(System.Runtime.CompilerServices.Unsafe.Read<TElement>((void*)left)))
                    {
                        IncrementSize(System.Runtime.CompilerServices.Unsafe.Read<TElement>((void*)root));
                        SetSize(node, GetOne());
                        System.Runtime.CompilerServices.Unsafe.Write((void*)left, node);
                        break;
                    }
                    if (FirstIsToTheRightOfSecond(node, System.Runtime.CompilerServices.Unsafe.Read<TElement>((void*)left))) // node.Key greater than left.Key
                    {
                        var leftRight = GetRightValue(System.Runtime.CompilerServices.Unsafe.Read<TElement>((void*)left));
                        var leftRightSize = GetSizeOrZero(leftRight);
                        if (GreaterThan(Increment(leftRightSize), rightSize))
                        {
                            if (EqualToZero(leftRightSize) && EqualToZero(rightSize))
                            {
                                SetLeft(node, System.Runtime.CompilerServices.Unsafe.Read<TElement>((void*)left));
                                SetRight(node, System.Runtime.CompilerServices.Unsafe.Read<TElement>((void*)root));
```

```csharp
                                SetSize(node, Add(GetSize(System.Runtime.CompilerServices.Unsafe.Read<TElement>((void*)left)), GetTwo())); // Two (2) - размер ветки *root (right) и самого node
                                SetLeft(System.Runtime.CompilerServices.Unsafe.Read<TElement>((void*)root), GetZero());
                                SetSize(System.Runtime.CompilerServices.Unsafe.Read<TElement>((void*)root), GetOne());
                                System.Runtime.CompilerServices.Unsafe.Write((void*)root, node);
                                break;
                            }
                            LeftRotate(left);
                            RightRotate(root);
                        }
                        else
                        {
                            IncrementSize(System.Runtime.CompilerServices.Unsafe.Read<TElement>((void*)root));
                            root = left;
                        }
                    }
                    else // node.Key less than left.Key
                    {
                        var leftLeft = GetLeftValue(System.Runtime.CompilerServices.Unsafe.Read<TElement>((void*)left));
                        var leftLeftSize = GetSizeOrZero(leftLeft);
                        if (GreaterThan(Increment(leftLeftSize), rightSize))
                        {
                            RightRotate(root);
                        }
                        else
                        {
                            IncrementSize(System.Runtime.CompilerServices.Unsafe.Read<TElement>((void*)root));
                            root = left;
                        }
                    }
                }
                else // node.Key greater than root.Key
                {
                    if (EqualToZero(System.Runtime.CompilerServices.Unsafe.Read<TElement>((void*)right)))
                    {
                        IncrementSize(System.Runtime.CompilerServices.Unsafe.Read<TElement>((void*)root));
                        SetSize(node, GetOne());
                        System.Runtime.CompilerServices.Unsafe.Write((void*)right, node);
                        break;
                    }
                    if (FirstIsToTheRightOfSecond(node, System.Runtime.CompilerServices.Unsafe.Read<TElement>((void*)right))) // node.Key greater than right.Key
                    {
                        var rightRight = GetRightValue(System.Runtime.CompilerServices.Unsafe.Read<TElement>((void*)right));
                        var rightRightSize = GetSizeOrZero(rightRight);
                        if (GreaterThan(Increment(rightRightSize), leftSize))
                        {
                            LeftRotate(root);
                        }
                        else
                        {
                            IncrementSize(System.Runtime.CompilerServices.Unsafe.Read<TElement>((void*)root));
                            root = right;
                        }
                    }
                    else // node.Key less than right.Key
                    {
                        var rightLeft = GetLeftValue(System.Runtime.CompilerServices.Unsafe.Read<TElement>((void*)right));
                        var rightLeftSize = GetSizeOrZero(rightLeft);
                        if (GreaterThan(Increment(rightLeftSize), leftSize))
                        {
                            if (EqualToZero(rightLeftSize) && EqualToZero(leftSize))
                            {
                                SetLeft(node, System.Runtime.CompilerServices.Unsafe.Read<TElement>((void*)root));
                                SetRight(node, System.Runtime.CompilerServices.Unsafe.Read<TElement>((void*)right));
                                SetSize(node, Add(GetSize(System.Runtime.CompilerServices.Unsafe.Read<TElement>((void*)right)), GetTwo())); // Two (2) - размер ветки *root (left) и самого node
                                SetRight(System.Runtime.CompilerServices.Unsafe.Read<TElement>((void*)root), GetZero());
                                SetSize(System.Runtime.CompilerServices.Unsafe.Read<TElement>((void*)root), GetOne());
                                System.Runtime.CompilerServices.Unsafe.Write((void*)root, node);
                                break;
                            }
                            RightRotate(right);
                            LeftRotate(root);
                        }
                        else
                        {
                            IncrementSize(System.Runtime.CompilerServices.Unsafe.Read<TElement>((void*)root));
                            root = right;
                        }
                    }
                }
            }
        }

        protected override void DetachCore(IntPtr root, TElement node)
        {
            while (true)
            {
                var left = GetLeftPointer(System.Runtime.CompilerServices.Unsafe.Read<TElement>((void*)root));
                var leftSize = GetSizeOrZero(System.Runtime.CompilerServices.Unsafe.Read<TElement>((void*)left));
                var right = GetRightPointer(System.Runtime.CompilerServices.Unsafe.Read<TElement>((void*)root));
                var rightSize = GetSizeOrZero(System.Runtime.CompilerServices.Unsafe.Read<TElement>((void*)right));
                if (FirstIsToTheLeftOfSecond(node, System.Runtime.CompilerServices.Unsafe.Read<TElement>((void*)root))) // node.Key less than root.Key
                {
                    EnsureNodeInTheTree(node, left);
                    var rightLeft = GetLeftValue(System.Runtime.CompilerServices.Unsafe.Read<TElement>((void*)right));
                    var rightLeftSize = GetSizeOrZero(rightLeft);
                    var rightRight = GetRightValue(System.Runtime.CompilerServices.Unsafe.Read<TElement>((void*)right));
                    var rightRightSize = GetSizeOrZero(rightRight);
                    if (GreaterThan(rightRightSize, Decrement(leftSize)))
                    {
                        LeftRotate(root);
                    }
                    else if (GreaterThan(rightLeftSize, Decrement(leftSize)))
                    {
                        RightRotate(right);
                        LeftRotate(root);
                    }
                    else
                    {
                        DecrementSize(System.Runtime.CompilerServices.Unsafe.Read<TElement>((void*)root));
                        root = left;
                    }
                }
                else if (FirstIsToTheRightOfSecond(node, System.Runtime.CompilerServices.Unsafe.Read<TElement>((void*)root))) // node.Key greater than root.Key
                {
                    EnsureNodeInTheTree(node, right);
                    var leftLeft = GetLeftValue(System.Runtime.CompilerServices.Unsafe.Read<TElement>((void*)left));
                    var leftLeftSize = GetSizeOrZero(leftLeft);
                    var leftRight = GetRightValue(System.Runtime.CompilerServices.Unsafe.Read<TElement>((void*)left));
                    var leftRightSize = GetSizeOrZero(leftRight);
                    if (GreaterThan(leftLeftSize, Decrement(rightSize)))
                    {
                        RightRotate(root);
                    }
                    else if (GreaterThan(leftRightSize, Decrement(rightSize)))
                    {
                        LeftRotate(left);
                        RightRotate(root);
                    }
                    else
                    {
                        DecrementSize(System.Runtime.CompilerServices.Unsafe.Read<TElement>((void*)root));
                        root = right;
                    }
                }
                else // key equals to root.Key
                {
                    if (GreaterThanZero(leftSize) && GreaterThanZero(rightSize))
                    {
                        if (GreaterThan(leftSize, rightSize))
                        {
                            var replacement = System.Runtime.CompilerServices.Unsafe.Read<TElement>((void*)left);
                            while (!EqualToZero(GetRightValue(replacement)))
                            {
                                replacement = GetRightValue(replacement);
                            }
                            DetachCore(left, replacement);
                            SetLeft(replacement, System.Runtime.CompilerServices.Unsafe.Read<TElement>((void*)left));
                            SetRight(replacement, System.Runtime.CompilerServices.Unsafe.Read<TElement>((void*)right));
                            FixSize(replacement);
                            System.Runtime.CompilerServices.Unsafe.Write((void*)root, replacement);
                        }
                        else
                        {
                            var replacement = System.Runtime.CompilerServices.Unsafe.Read<TElement>((void*)right);
                            while (!EqualToZero(GetLeftValue(replacement)))
                            {
                                replacement = GetLeftValue(replacement);
                            }
                            DetachCore(right, replacement);
                            SetLeft(replacement, System.Runtime.CompilerServices.Unsafe.Read<TElement>((void*)left));
                            SetRight(replacement, System.Runtime.CompilerServices.Unsafe.Read<TElement>((void*)right));
                            FixSize(replacement);
                            System.Runtime.CompilerServices.Unsafe.Write((void*)root, replacement);
                        }
                    }
                    else if (GreaterThanZero(leftSize))
                    {
                        System.Runtime.CompilerServices.Unsafe.Write((void*)root, System.Runtime.CompilerServices.Unsafe.Read<TElement>((void*)left));
                    }
                    else if (GreaterThanZero(rightSize))
                    {
                        System.Runtime.CompilerServices.Unsafe.Write((void*)root, System.Runtime.CompilerServices.Unsafe.Read<TElement>((void*)right));
                    }
                    else
                    {
                        System.Runtime.CompilerServices.Unsafe.Write((void*)root, GetZero());
                    }
                    ClearNode(node);
                    break;
                }
            }
        }
        private void EnsureNodeInTheTree(TElement node, IntPtr branch)
        {
            if (EqualToZero(System.Runtime.CompilerServices.Unsafe.Read<TElement>((void*)branch)))
            {
                throw new InvalidOperationException($"Элемент {node} не содержится в дереве.");
            }
        }
    }
}
```

./Trees/SizedAndThreadedAVLBalancedTreeMethods.cs

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

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

namespace Platform.Collections.Methods.Trees
{
    /// <summary>
    /// Combination of Size, Height (AVL), and threads.
    /// </summary>
    /// <remarks>
    /// Based on: <a href="https://github.com/programmatom/TreeLib/blob/master/TreeLib/TreeLib/Generated/AVLTreeList.cs">TreeLib.AVLTreeList</a>.
    /// Which itself based on: <a href="https://github.com/GNOME/glib/blob/master/glib/gtree.c">GNOME/glib/gtree</a>.
    /// </remarks>
    public unsafe abstract class SizedAndThreadedAVLBalancedTreeMethods<TElement> : SizedBinaryTreeMethodsBase<TElement>
    {
        // TODO: Link with size of TElement
        private const int MaxPath = 92;

        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) ? base.GetLeftOrDefault(node) : default;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override TElement GetRightOrDefault(TElement node) => GetRightIsChild(node) ? base.GetRightOrDefault(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(IntPtr 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 rootPointer = (void*)root;
                var currentNode =
                 ↪  System.Runtime.CompilerServices.Unsafe.Read<TElement>(rootPointer);
                while (true)
                {
                    if (FirstIsToTheLeftOfSecond(node, currentNode))
                    {
                        if (GetLeftIsChild(currentNode))
                        {
                            IncrementSize(currentNode);
                            path[pathPosition++] = currentNode;
                            currentNode = GetLeftValue(currentNode);
                        }
                        else
                        {
                            // Threads
                            SetLeft(node, GetLeftValue(currentNode));
                            SetRight(node, currentNode);
                            SetLeft(currentNode, node);
                            SetLeftIsChild(currentNode, true);
                            DecrementBalance(currentNode);
                            SetSize(node, GetOne());
                            FixSize(currentNode); // Should be incremented already
                            break;
                        }
                    }
                    else if (FirstIsToTheRightOfSecond(node, currentNode))
                    {
                        if (GetRightIsChild(currentNode))
                        {
                            IncrementSize(currentNode);
                            path[pathPosition++] = currentNode;
                            currentNode = GetRightValue(currentNode);
                        }
                        else
                        {
                            // Threads
                            SetRight(node, GetRightValue(currentNode));
                            SetLeft(node, currentNode);
                            SetRight(currentNode, node);
                            SetRightIsChild(currentNode, true);
                            IncrementBalance(currentNode);
                            SetSize(node, GetOne());
                            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 = !IsEquals(parent, default) && IsEquals(currentNode,
                     ↪  GetLeftValue(parent));
                    var currentNodeBalance = GetBalance(currentNode);
                    if (currentNodeBalance < -1 || currentNodeBalance > 1)
                    {
                        currentNode = Balance(currentNode);
                        if (IsEquals(parent, default))
                        {
                            System.Runtime.CompilerServices.Unsafe.Write((void*)root,
                             ↪  currentNode);
                        }
                        else if (isLeftNode)
                        {
                            SetLeft(parent, currentNode);
                            FixSize(parent);
                        }
                        else
                        {
                            SetRight(parent, currentNode);
                            FixSize(parent);
                        }
                    }
                    currentNodeBalance = GetBalance(currentNode);
                    if (currentNodeBalance == 0 || IsEquals(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 = GetLeftValue(node);
                    if (GetBalance(left) > 0)
                    {
                        SetLeft(node, LeftRotateWithBalance(left));
                        FixSize(node);
                    }
                    node = RightRotateWithBalance(node);
                }
                else if (rootBalance > 1)
                {
                    var right = GetRightValue(node);
                    if (GetBalance(right) < 0)
                    {
                        SetRight(node, RightRotateWithBalance(right));
                        FixSize(node);
                    }
                    node = LeftRotateWithBalance(node);
                }
                return node;
            }
        }

        protected TElement LeftRotateWithBalance(TElement node)
        {
            unchecked
            {
                var right = GetRightValue(node);
                if (GetLeftIsChild(right))
                {
                    SetRight(node, GetLeftValue(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 = GetLeftValue(node);
                if (GetRightIsChild(left))
                {
                    SetLeft(node, GetRightValue(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;
            }
        }

        protected TElement GetNext(TElement node)
        {
            unchecked
            {
                var current = GetRightValue(node);
                if (GetRightIsChild(node))
                {
                    while (GetLeftIsChild(current))
                    {
                        current = GetLeftValue(current);
                    }
                }
                return current;
            }
        }

        protected TElement GetPrevious(TElement node)
        {
            unchecked
            {
                var current = GetLeftValue(node);
                if (GetLeftIsChild(node))
                {
                    while (GetRightIsChild(current))
                    {
                        current = GetRightValue(current);
                    }
                }
                return current;
            }
        }

        protected override void DetachCore(IntPtr 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 rootPointer = (void*)root;
                var currentNode =
                ↪  System.Runtime.CompilerServices.Unsafe.Read<TElement>(rootPointer);
                while (true)
                {
                    if (FirstIsToTheLeftOfSecond(node, currentNode))
                    {
                        if (!GetLeftIsChild(currentNode))
                        {
                            throw new InvalidOperationException("Cannot find a node.");
                        }
                        DecrementSize(currentNode);
                        path[pathPosition++] = currentNode;
                        currentNode = GetLeftValue(currentNode);
                    }
                    else if (FirstIsToTheRightOfSecond(node, currentNode))
                    {
                        if (!GetRightIsChild(currentNode))
                        {
                            throw new InvalidOperationException("Cannot find a node.");
                        }
                    DecrementSize(currentNode);
                    path[pathPosition++] = currentNode;
                    currentNode = GetRightValue(currentNode);
                }
                else
                {
                    break;
                }
            }
            var parent = path[--pathPosition];
            var balanceNode = parent;
            var isLeftNode = !IsEquals(parent, default) && IsEquals(currentNode,
             ↪  GetLeftValue(parent));
            if (!GetLeftIsChild(currentNode))
            {
                if (!GetRightIsChild(currentNode)) // node has no children
                {
                    if (IsEquals(parent, default))
                    {
                        System.Runtime.CompilerServices.Unsafe.Write(rootPointer, GetZero());
                    }
                    else if (isLeftNode)
                    {
                        SetLeftIsChild(parent, false);
                        SetLeft(parent, GetLeftValue(currentNode));
                        IncrementBalance(parent);
                    }
                    else
                    {
                        SetRightIsChild(parent, false);
                        SetRight(parent, GetRightValue(currentNode));
                        DecrementBalance(parent);
                    }
                }
                else // node has a right child
                {
                    var successor = GetNext(currentNode);
                    SetLeft(successor, GetLeftValue(currentNode));
                    var right = GetRightValue(currentNode);
                    if (IsEquals(parent, default))
                    {
                        System.Runtime.CompilerServices.Unsafe.Write(rootPointer, 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, GetRightValue(currentNode));
                    var leftValue = GetLeftValue(currentNode);
                    if (IsEquals(parent, default))
                    {
                        System.Runtime.CompilerServices.Unsafe.Write(rootPointer, 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 = GetLeftValue(currentNode);
                        var successor = GetRightValue(currentNode);
                        var successorParent = currentNode;
                        int previousPathPosition = ++pathPosition;
                        // find the immediately next node (and its parent)
                        while (GetLeftIsChild(successor))
                        {
                            path[++pathPosition] = successorParent = successor;
                            successor = GetLeftValue(successor);
                            if (!IsEquals(successorParent, currentNode))
                            {
                                DecrementSize(successorParent);
                            }
                        }
                        path[previousPathPosition] = successor;
                        balanceNode = path[pathPosition];
                        // remove 'successor' from the tree
                        if (!IsEquals(successorParent, currentNode))
                        {
                            if (!GetRightIsChild(successor))
                            {
                                SetLeftIsChild(successorParent, false);
                            }
                            else
                            {
                                SetLeft(successorParent, GetRightValue(successor));
                            }
                            IncrementBalance(successorParent);
                            SetRightIsChild(successor, true);
                            SetRight(successor, GetRightValue(currentNode));
                        }
                        else
                        {
                            DecrementBalance(currentNode);
                        }
                        // set the predecessor's successor link to point to the right place
                        while (GetRightIsChild(predecessor))
                        {
                            predecessor = GetRightValue(predecessor);
                        }
                        SetRight(predecessor, successor);
                        // prepare 'successor' to replace 'node'
                        var left = GetLeftValue(currentNode);
                        SetLeftIsChild(successor, true);
                        SetLeft(successor, left);
                        SetBalance(successor, GetBalance(currentNode));
                        FixSize(successor);
                        if (IsEquals(parent, default))
                        {
                            System.Runtime.CompilerServices.Unsafe.Write(rootPointer, successor);
                        }
                        else if (isLeftNode)
                        {
                            SetLeft(parent, successor);
                        }
                        else
                        {
                            SetRight(parent, successor);
                        }
                    }
                }
                // restore balance
                if (!IsEquals(balanceNode, default))
                {
                    while (true)
                    {
                        var balanceParent = path[--pathPosition];
                        isLeftNode = !IsEquals(balanceParent, default) && IsEquals(balanceNode,
                         ↪  GetLeftValue(balanceParent));
                        var currentNodeBalance = GetBalance(balanceNode);
                        if (currentNodeBalance < -1 || currentNodeBalance > 1)
                        {
                            balanceNode = Balance(balanceNode);
                            if (IsEquals(balanceParent, default))
                            {
                                System.Runtime.CompilerServices.Unsafe.Write(rootPointer,
                                 ↪  balanceNode);
                            }
                            else if (isLeftNode)
                            {
                                SetLeft(balanceParent, balanceNode);
                            }
                            else
                            {
                                SetRight(balanceParent, balanceNode);
                            }
                        }
                        currentNodeBalance = GetBalance(balanceNode);
                        if (currentNodeBalance != 0 || IsEquals(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, GetZero());
            SetRight(node, GetZero());
            SetSize(node, GetZero());
            SetLeftIsChild(node, false);
            SetRightIsChild(node, false);
            SetBalance(node, 0);
        }
    }
}
```

./Trees/SizedBinaryTreeMethodsBase.cs

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

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

namespace Platform.Collections.Methods.Trees
{
    public unsafe abstract class SizedBinaryTreeMethodsBase<TElement> :
    ↪   GenericCollectionMethodsBase<TElement>
    {
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected abstract IntPtr GetLeftPointer(TElement node);

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

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected abstract TElement GetRightValue(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) => GetLeftPointer(node) !=
        ↪  IntPtr.Zero ? GetLeftValue(node) : default;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected virtual TElement GetRightOrDefault(TElement node) => GetRightPointer(node) !=
        ↪  IntPtr.Zero ? GetRightValue(node) : default;

        [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) ? GetZero() :
        ↪  GetSize(node);

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected void LeftRotate(IntPtr root)
        {
            var rootPointer = (void*)root;
            System.Runtime.CompilerServices.Unsafe.Write(rootPointer,
            ↪  LeftRotate(System.Runtime.CompilerServices.Unsafe.Read<TElement>(rootPointer)));
        }

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected void RightRotate(IntPtr root)
        {
            var rootPointer = (void*)root;
            System.Runtime.CompilerServices.Unsafe.Write(rootPointer,
            ↪  RightRotate(System.Runtime.CompilerServices.Unsafe.Read<TElement>(rootPointer)));
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected TElement RightRotate(TElement root)
        {
            var left = GetLeftValue(root);
#if ENABLE_TREE_AUTO_DEBUG_AND_VALIDATION
            if (EqualToZero(left))
            {
                throw new Exception("Left is null.");
            }
#endif
            SetLeft(root, GetRightValue(left));
            SetRight(left, root);
            SetSize(left, GetSize(root));
            FixSize(root);
            return left;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public 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;
        }

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

        public void Attach(IntPtr 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 (ValueEqualToZero(root))
            {
                SetSize(node, GetOne());
                System.Runtime.CompilerServices.Unsafe.Write((void*)root, node);
                return;
            }
            AttachCore(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 (!IsEquals(MathHelpers.Increment(sizeBefore), sizeAfter))
            {
                throw new Exception("Tree was broken after attach.");
            }
#endif
        }

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

        public void Detach(IntPtr 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 (ValueEqualToZero(root))
            {
                throw new Exception($"Элемент с {node} не содержится в дереве.");
            }
#endif
            DetachCore(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 (!IsEquals(MathHelpers.Decrement(sizeBefore), sizeAfter))
            {
                throw new Exception("Tree was broken after detach.");
            }
#endif
        }

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public TElement GetSize(IntPtr root) => root == IntPtr.Zero ? GetZero() :
        ↪  GetSizeOrZero(System.Runtime.CompilerServices.Unsafe.Read<TElement>((void*)root));

        public void FixSizes(IntPtr root)
        {
            if (root != IntPtr.Zero)
            {
                FixSizes(System.Runtime.CompilerServices.Unsafe.Read<TElement>((void*)root));
            }
        }

        public void FixSizes(TElement node)
        {
            if (IsEquals(node, default))
            {
                return;
            }
            FixSizes(GetLeftOrDefault(node));
            FixSizes(GetRightOrDefault(node));
            FixSize(node);
        }

        public void ValidateSizes(IntPtr root)
        {
            if (root != IntPtr.Zero)
            {
                ValidateSizes(System.Runtime.CompilerServices.Unsafe.Read<TElement>((void*)root)
                ↪  );
            }
        }

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

        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 (!IsEquals(size, expectedSize))
            {
                throw new InvalidOperationException($"Size of {node} is not valid. Expected
                 ↪  size: {expectedSize}, actual size: {size}.");
            }
        }

        public string PrintNodes(IntPtr root)
        {
            if (root != IntPtr.Zero)
            {
                var sb = new StringBuilder();
                PrintNodes(System.Runtime.CompilerServices.Unsafe.Read<TElement>((void*)root),
                 ↪  sb);
                return sb.ToString();
            }
            return "";
        }

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

        public void PrintNodes(TElement node, StringBuilder sb) => PrintNodes(node, sb, 0);

        public void PrintNodes(TElement node, StringBuilder sb, int level)
        {
            if (IsEquals(node, default))
            {
                return;
            }
            PrintNodes(GetLeftOrDefault(node), sb, level + 1);
            PrintNode(node, sb, level);
            sb.AppendLine();
            PrintNodes(GetRightOrDefault(node), sb, level + 1);
        }

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

        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);
    }
}
```

## Index