./GenericCollectionMethodsBase.cs

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

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

namespace Platform.Collections.Methods
{
    public 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(pointer.GetValue<TElement>(), 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

```
#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;
using Platform.Unsafe;

#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 abstract class SizeBalancedTreeMethods2<TElement> :
    ↪ SizedBinaryTreeMethodsBase<TElement>
    {
        protected override void AttachCore(IntPtr root, TElement newNode)
        {
            if (ValueEqualToZero(root))
            {
                root.SetValue(newNode);
                IncrementSize(root.GetValue<TElement>());
            }
            else
            {
                IncrementSize(root.GetValue<TElement>());
                if (FirstIsToTheLeftOfSecond(newNode, root.GetValue<TElement>()))
                {
                    AttachCore(GetLeftPointer(root.GetValue<TElement>()), newNode);
                    LeftMaintain(root);
                }
                else
                {
                    AttachCore(GetRightPointer(root.GetValue<TElement>()), 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(currentNode.GetValue<TElement>(), nodeToDetach))
            {
                SetSize(currentNode.GetValue<TElement>(),
                ↪ Decrement(GetSize(currentNode.GetValue<TElement>())));
                if (FirstIsToTheLeftOfSecond(nodeToDetach, currentNode.GetValue<TElement>()))
                {
                    parent = currentNode;
                    currentNode = GetLeftPointer(currentNode.GetValue<TElement>());
                }
                else if (FirstIsToTheRightOfSecond(nodeToDetach,
                ↪ currentNode.GetValue<TElement>()))
                {
                    parent = currentNode;
                    currentNode = GetRightPointer(currentNode.GetValue<TElement>());
                }
                else
                {
                    throw new InvalidOperationException("Duplicate link found in the tree.");
                }
            }
```

```
63              if (!ValueEqualToZero(GetLeftPointer(nodeToDetach)) &&
                ↪  !ValueEqualToZero(GetRightPointer(nodeToDetach)))
64              {
65                  var minNode = GetRightValue(nodeToDetach);
66                  while (!EqualToZero(GetLeftValue(minNode)))
67                  {
68                      minNode = GetLeftValue(minNode); /* Передвигаемся до минимума */
69                  }
70                  DetachCore(GetRightPointer(nodeToDetach), minNode);
71                  SetLeft(minNode, GetLeftValue(nodeToDetach));
72                  if (!ValueEqualToZero(GetRightPointer(nodeToDetach)))
73                  {
74                      SetRight(minNode, GetRightValue(nodeToDetach));
75                      SetSize(minNode, Increment(Add(GetSize(GetLeftValue(nodeToDetach)),
                        ↪  GetSize(GetRightValue(nodeToDetach)))));
76                  }
77                  else
78                  {
79                      SetSize(minNode, Increment(GetSize(GetLeftValue(nodeToDetach))));
80                  }
81                  replacementNode = minNode;
82              }
83              else if (!ValueEqualToZero(GetLeftPointer(nodeToDetach)))
84              {
85                  replacementNode = GetLeftValue(nodeToDetach);
86              }
87              else if (!ValueEqualToZero(GetRightPointer(nodeToDetach)))
88              {
89                  replacementNode = GetRightValue(nodeToDetach);
90              }
91              if (parent == IntPtr.Zero)
92              {
93                  root.SetValue(replacementNode);
94              }
95              else if (IsEquals(GetLeftValue(parent.GetValue<TElement>()), nodeToDetach))
96              {
97                  SetLeft(parent.GetValue<TElement>(), replacementNode);
98              }
99              else if (IsEquals(GetRightValue(parent.GetValue<TElement>()), nodeToDetach))
100             {
101                 SetRight(parent.GetValue<TElement>(), replacementNode);
102             }
103             ClearNode(nodeToDetach);
104         }
105
106         private void LeftMaintain(IntPtr root)
107         {
108             if (!ValueEqualToZero(root))
109             {
110                 var rootLeftNode = GetLeftPointer(root.GetValue<TElement>());
111                 if (!ValueEqualToZero(rootLeftNode))
112                 {
113                     var rootRightNode = GetRightPointer(root.GetValue<TElement>());
114                     var rootLeftNodeLeftNode = GetLeftPointer(rootLeftNode.GetValue<TElement>());
115                     if (!ValueEqualToZero(rootLeftNodeLeftNode) &&
116                         (ValueEqualToZero(rootRightNode) ||
                            ↪  GreaterThan(GetSize(rootLeftNodeLeftNode.GetValue<TElement>()),
                            ↪  GetSize(rootRightNode.GetValue<TElement>()))))
117                     {
118                         RightRotate(root);
119                     }
120                     else
121                     {
122                         var rootLeftNodeRightNode =
                            ↪  GetRightPointer(rootLeftNode.GetValue<TElement>());
123                         if (!ValueEqualToZero(rootLeftNodeRightNode) &&
124                             (ValueEqualToZero(rootRightNode) ||
                                ↪  GreaterThan(GetSize(rootLeftNodeRightNode.GetValue<TElement>()),
                                ↪  GetSize(rootRightNode.GetValue<TElement>()))))
125                         {
126                             LeftRotate(GetLeftPointer(root.GetValue<TElement>()));
127                             RightRotate(root);
128                         }
129                         else
130                         {
131                             return;
132                         }
133                     }
```

```
                    LeftMaintain(GetLeftPointer(root.GetValue<TElement>()));
                    RightMaintain(GetRightPointer(root.GetValue<TElement>()));
                    LeftMaintain(root);
                    RightMaintain(root);
                }
            }
        }

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

./Trees/SizeBalancedTreeMethods.cs

```
using System;
using Platform.Unsafe;

#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(IntPtr root, TElement node)
        {
            while (true)
            {
                var left = GetLeftPointer(root.GetValue<TElement>());
                var leftSize = GetSizeOrZero(left.GetValue<TElement>());
                var right = GetRightPointer(root.GetValue<TElement>());
                var rightSize = GetSizeOrZero(right.GetValue<TElement>());
                if (FirstIsToTheLeftOfSecond(node, root.GetValue<TElement>())) // node.Key less
                ↪ than root.Key
                {
                    if (EqualToZero(left.GetValue<TElement>()))
                    {
                        IncrementSize(root.GetValue<TElement>());
                        SetSize(node, GetOne());
```

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

```
                                SetLeft(node, root.GetValue<TElement>());
                                SetRight(node, right.GetValue<TElement>());
                                SetSize(node, Add(GetSize(right.GetValue<TElement>()),
                                ↪  GetTwo())); // Two (2) - размер ветки *root (left) и самого
                                ↪  node
                                SetRight(root.GetValue<TElement>(), GetZero());
                                SetSize(root.GetValue<TElement>(), GetOne());
                                root.SetValue(node);
                                break;
                            }
                            RightRotate(right);
                            LeftRotate(root);
                        }
                        else
                        {
                            IncrementSize(root.GetValue<TElement>());
                            root = right;
                        }
                    }
                }
            }

            protected override void DetachCore(IntPtr root, TElement node)
            {
                while (true)
                {
                    var left = GetLeftPointer(root.GetValue<TElement>());
                    var leftSize = GetSizeOrZero(left.GetValue<TElement>());
                    var right = GetRightPointer(root.GetValue<TElement>());
                    var rightSize = GetSizeOrZero(right.GetValue<TElement>());
                    if (FirstIsToTheLeftOfSecond(node, root.GetValue<TElement>())) // node.Key less
                    ↪  than root.Key
                    {
                        EnsureNodeInTheTree(node, left);
                        var rightLeft = GetLeftValue(right.GetValue<TElement>());
                        var rightLeftSize = GetSizeOrZero(rightLeft);
                        var rightRight = GetRightValue(right.GetValue<TElement>());
                        var rightRightSize = GetSizeOrZero(rightRight);
                        if (GreaterThan(rightRightSize, Decrement(leftSize)))
                        {
                            LeftRotate(root);
                        }
                        else if (GreaterThan(rightLeftSize, Decrement(leftSize)))
                        {
                            RightRotate(right);
                            LeftRotate(root);
                        }
                        else
                        {
                            DecrementSize(root.GetValue<TElement>());
                            root = left;
                        }
                    }
                    else if (FirstIsToTheRightOfSecond(node, root.GetValue<TElement>())) // node.Key
                    ↪  greater than root.Key
                    {
                        EnsureNodeInTheTree(node, right);
                        var leftLeft = GetLeftValue(left.GetValue<TElement>());
                        var leftLeftSize = GetSizeOrZero(leftLeft);
                        var leftRight = GetRightValue(left.GetValue<TElement>());
                        var leftRightSize = GetSizeOrZero(leftRight);
                        if (GreaterThan(leftLeftSize, Decrement(rightSize)))
                        {
                            RightRotate(root);
                        }
                        else if (GreaterThan(leftRightSize, Decrement(rightSize)))
                        {
                            LeftRotate(left);
                            RightRotate(root);
                        }
                        else
                        {
                            DecrementSize(root.GetValue<TElement>());
                            root = right;
                        }
                    }
                    else // key equals to root.Key
```

```
                {
                    if (GreaterThanZero(leftSize) && GreaterThanZero(rightSize))
                    {
                        if (GreaterThan(leftSize, rightSize))
                        {
                            var replacement = left.GetValue<TElement>();
                            while (!EqualToZero(GetRightValue(replacement)))
                            {
                                replacement = GetRightValue(replacement);
                            }
                            DetachCore(left, replacement);
                            SetLeft(replacement, left.GetValue<TElement>());
                            SetRight(replacement, right.GetValue<TElement>());
                            FixSize(replacement);
                            root.SetValue(replacement);
                        }
                        else
                        {
                            var replacement = right.GetValue<TElement>();
                            while (!EqualToZero(GetLeftValue(replacement)))
                            {
                                replacement = GetLeftValue(replacement);
                            }
                            DetachCore(right, replacement);
                            SetLeft(replacement, left.GetValue<TElement>());
                            SetRight(replacement, right.GetValue<TElement>());
                            FixSize(replacement);
                            root.SetValue(replacement);
                        }
                    }
                    else if (GreaterThanZero(leftSize))
                    {
                        root.SetValue(left.GetValue<TElement>());
                    }
                    else if (GreaterThanZero(rightSize))
                    {
                        root.SetValue(right.GetValue<TElement>());
                    }
                    else
                    {
                        root.SetValue(GetZero());
                    }
                    ClearNode(node);
                    break;
                }
            }
        }

        private void EnsureNodeInTheTree(TElement node, IntPtr branch)
        {
            if (EqualToZero(branch.GetValue<TElement>()))
            {
                throw new InvalidOperationException($"Элемент {node} не содержится в дереве.");
            }
        }
    }
}
```

./Trees/SizedAndThreadedAVLBalancedTreeMethods.cs

```
using System;
using System.Runtime.CompilerServices;
using System.Text;
using Platform.Unsafe;
#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/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>.
```

```
19      /// </remarks>
20      public abstract class SizedAndThreadedAVLBalancedTreeMethods<TElement> :
        ↪  SizedBinaryTreeMethodsBase<TElement>
21      {
22          // TODO: Link with size of TElement
23          private const int MaxPath = 92;
24
25          protected override void PrintNode(TElement node, StringBuilder sb, int level)
26          {
27              base.PrintNode(node, sb, level);
28              sb.Append(' ');
29              sb.Append(GetLeftIsChild(node) ? 'l' : 'L');
30              sb.Append(GetRightIsChild(node) ? 'r' : 'R');
31              sb.Append(' ');
32              sb.Append(GetBalance(node));
33          }
34
35          [MethodImpl(MethodImplOptions.AggressiveInlining)]
36          protected void IncrementBalance(TElement node) => SetBalance(node,
            ↪  (sbyte)(GetBalance(node) + 1));
37
38          [MethodImpl(MethodImplOptions.AggressiveInlining)]
39          protected void DecrementBalance(TElement node) => SetBalance(node,
            ↪  (sbyte)(GetBalance(node) - 1));
40
41          [MethodImpl(MethodImplOptions.AggressiveInlining)]
42          protected override TElement GetLeftOrDefault(TElement node) => GetLeftIsChild(node) ?
            ↪  base.GetLeftOrDefault(node) : default;
43
44          [MethodImpl(MethodImplOptions.AggressiveInlining)]
45          protected override TElement GetRightOrDefault(TElement node) => GetRightIsChild(node) ?
            ↪  base.GetRightOrDefault(node) : default;
46
47          protected abstract bool GetLeftIsChild(TElement node);
48          protected abstract void SetLeftIsChild(TElement node, bool value);
49          protected abstract bool GetRightIsChild(TElement node);
50          protected abstract void SetRightIsChild(TElement node, bool value);
51          protected abstract sbyte GetBalance(TElement node);
52          protected abstract void SetBalance(TElement node, sbyte value);
53
54          protected override void AttachCore(IntPtr root, TElement node)
55          {
56              unchecked
57              {
58                  // TODO: Check what is faster to use simple array or array from array pool
59                  // TODO: Try to use stackalloc as an optimization (requires code generation,
                    ↪  because of generics)
60  #if USEARRAYPOOL
61                  var path = ArrayPool.Allocate<TElement>(MaxPath);
62                  var pathPosition = 0;
63                  path[pathPosition++] = default;
64  #else
65                  var path = new TElement[MaxPath];
66                  var pathPosition = 1;
67  #endif
68                  var currentNode = root.GetValue<TElement>();
69                  while (true)
70                  {
71                      if (FirstIsToTheLeftOfSecond(node, currentNode))
72                      {
73                          if (GetLeftIsChild(currentNode))
74                          {
75                              IncrementSize(currentNode);
76                              path[pathPosition++] = currentNode;
77                              currentNode = GetLeftValue(currentNode);
78                          }
79                          else
80                          {
81                              // Threads
82                              SetLeft(node, GetLeftValue(currentNode));
83                              SetRight(node, currentNode);
84                              SetLeft(currentNode, node);
85                              SetLeftIsChild(currentNode, true);
86                              DecrementBalance(currentNode);
87                              SetSize(node, GetOne());
88                              FixSize(currentNode); // Should be incremented already
89                              break;
90                          }
91                      }
```

```
                    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))
                        {
                            root.SetValue(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
```

```
168             {
169                 var rootBalance = GetBalance(node);
170                 if (rootBalance < -1)
171                 {
172                     var left = GetLeftValue(node);
173                     if (GetBalance(left) > 0)
174                     {
175                         SetLeft(node, LeftRotateWithBalance(left));
176                         FixSize(node);
177                     }
178                     node = RightRotateWithBalance(node);
179                 }
180                 else if (rootBalance > 1)
181                 {
182                     var right = GetRightValue(node);
183                     if (GetBalance(right) < 0)
184                     {
185                         SetRight(node, RightRotateWithBalance(right));
186                         FixSize(node);
187                     }
188                     node = LeftRotateWithBalance(node);
189                 }
190                 return node;
191             }
192         }
193 
194         protected TElement LeftRotateWithBalance(TElement node)
195         {
196             unchecked
197             {
198                 var right = GetRightValue(node);
199                 if (GetLeftIsChild(right))
200                 {
201                     SetRight(node, GetLeftValue(right));
202                 }
203                 else
204                 {
205                     SetRightIsChild(node, false);
206                     SetLeftIsChild(right, true);
207                 }
208                 SetLeft(right, node);
209                 // Fix size
210                 SetSize(right, GetSize(node));
211                 FixSize(node);
212                 // Fix balance
213                 var rootBalance = GetBalance(node);
214                 var rightBalance = GetBalance(right);
215                 if (rightBalance <= 0)
216                 {
217                     if (rootBalance >= 1)
218                     {
219                         SetBalance(right, (sbyte)(rightBalance - 1));
220                     }
221                     else
222                     {
223                         SetBalance(right, (sbyte)(rootBalance + rightBalance - 2));
224                     }
225                     SetBalance(node, (sbyte)(rootBalance - 1));
226                 }
227                 else
228                 {
229                     if (rootBalance <= rightBalance)
230                     {
231                         SetBalance(right, (sbyte)(rootBalance - 2));
232                     }
233                     else
234                     {
235                         SetBalance(right, (sbyte)(rightBalance - 1));
236                     }
237                     SetBalance(node, (sbyte)(rootBalance - rightBalance - 1));
238                 }
239                 return right;
240             }
241         }
242 
243         protected TElement RightRotateWithBalance(TElement node)
244         {
245             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 currentNode = root.GetValue<TElement>();
                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))
                        {
                            root.SetValue(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))
                        {
                            root.SetValue(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))
                    {
                        root.SetValue(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))
                        {
                            root.SetValue(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))
                            {
                                root.SetValue(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;
using Platform.Unsafe;

//#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 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) => root.SetValue(LeftRotate(root.GetValue<TElement>()));

        [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) =>
        ↪ root.SetValue(RightRotate(root.GetValue<TElement>()));

        [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());
                root.SetValue(node);
                return;
            }
            AttachCore(root, node);
#if ENABLE_TREE_AUTO_DEBUG_AND_VALIDATION
```

```
154             Debug.WriteLine("--AfterAttach--");
155             Debug.WriteLine(PrintNodes(root));
156             Debug.WriteLine("----------------");
157             ValidateSizes(root);
158             var sizeAfter = GetSize(root);
159             if (!IsEquals(MathHelpers.Increment(sizeBefore), sizeAfter))
160             {
161                 throw new Exception("Tree was broken after attach.");
162             }
163 #endif
164         }
165 
166         protected abstract void AttachCore(IntPtr root, TElement node);
167 
168         public void Detach(IntPtr root, TElement node)
169         {
170 #if ENABLE_TREE_AUTO_DEBUG_AND_VALIDATION
171             ValidateSizes(root);
172             Debug.WriteLine("--BeforeDetach--");
173             Debug.WriteLine(PrintNodes(root));
174             Debug.WriteLine("----------------");
175             var sizeBefore = GetSize(root);
176             if (ValueEqualToZero(root))
177             {
178                 throw new Exception($"Элемент с {node} не содержится в дереве.");
179             }
180 #endif
181             DetachCore(root, node);
182 #if ENABLE_TREE_AUTO_DEBUG_AND_VALIDATION
183             Debug.WriteLine("--AfterDetach--");
184             Debug.WriteLine(PrintNodes(root));
185             Debug.WriteLine("----------------");
186             ValidateSizes(root);
187             var sizeAfter = GetSize(root);
188             if (!IsEquals(MathHelpers.Decrement(sizeBefore), sizeAfter))
189             {
190                 throw new Exception("Tree was broken after detach.");
191             }
192 #endif
193         }
194 
195         protected abstract void DetachCore(IntPtr root, TElement node);
196 
197         public TElement GetSize(IntPtr root) => root == IntPtr.Zero ? GetZero() :
        ↪  GetSizeOrZero(root.GetValue<TElement>());
198 
199         public void FixSizes(IntPtr root)
200         {
201             if (root != IntPtr.Zero)
202             {
203                 FixSizes(root.GetValue<TElement>());
204             }
205         }
206 
207         public void FixSizes(TElement node)
208         {
209             if (IsEquals(node, default))
210             {
211                 return;
212             }
213             FixSizes(GetLeftOrDefault(node));
214             FixSizes(GetRightOrDefault(node));
215             FixSize(node);
216         }
217 
218         public void ValidateSizes(IntPtr root)
219         {
220             if (root != IntPtr.Zero)
221             {
222                 ValidateSizes(root.GetValue<TElement>());
223             }
224         }
225 
226         public void ValidateSizes(TElement node)
227         {
228             if (IsEquals(node, default))
229             {
230                 return;
231             }
```

```
232             var size = GetSize(node);
233             var leftSize = GetLeftSize(node);
234             var rightSize = GetRightSize(node);
235             var expectedSize = Arithmetic.Increment(Arithmetic.Add(leftSize, rightSize));
236             if (!IsEquals(size, expectedSize))
237             {
238                 throw new InvalidOperationException($"Size of {node} is not valid. Expected
                    ↪  size: {expectedSize}, actual size: {size}.");
239             }
240             ValidateSizes(GetLeftOrDefault(node));
241             ValidateSizes(GetRightOrDefault(node));
242         }
243 
244         public void ValidateSize(TElement node)
245         {
246             var size = GetSize(node);
247             var leftSize = GetLeftSize(node);
248             var rightSize = GetRightSize(node);
249             var expectedSize = Arithmetic.Increment(Arithmetic.Add(leftSize, rightSize));
250             if (!IsEquals(size, expectedSize))
251             {
252                 throw new InvalidOperationException($"Size of {node} is not valid. Expected
                    ↪  size: {expectedSize}, actual size: {size}.");
253             }
254         }
255 
256         public string PrintNodes(IntPtr root)
257         {
258             if (root != IntPtr.Zero)
259             {
260                 var sb = new StringBuilder();
261                 PrintNodes(root.GetValue<TElement>(), sb);
262                 return sb.ToString();
263             }
264             return "";
265         }
266 
267         public string PrintNodes(TElement node)
268         {
269             var sb = new StringBuilder();
270             PrintNodes(node, sb);
271             return sb.ToString();
272         }
273 
274         public void PrintNodes(TElement node, StringBuilder sb) => PrintNodes(node, sb, 0);
275 
276         public void PrintNodes(TElement node, StringBuilder sb, int level)
277         {
278             if (IsEquals(node, default))
279             {
280                 return;
281             }
282             PrintNodes(GetLeftOrDefault(node), sb, level + 1);
283             PrintNode(node, sb, level);
284             sb.AppendLine();
285             PrintNodes(GetRightOrDefault(node), sb, level + 1);
286         }
287 
288         public string PrintNode(TElement node)
289         {
290             var sb = new StringBuilder();
291             PrintNode(node, sb);
292             return sb.ToString();
293         }
294 
295         protected void PrintNode(TElement node, StringBuilder sb) => PrintNode(node, sb, 0);
296 
297         protected virtual void PrintNode(TElement node, StringBuilder sb, int level)
298         {
299             sb.Append('\t', level);
300             sb.Append(node);
301             PrintNodeValue(node, sb);
302             sb.Append(' ');
303             sb.Append('s');
304             sb.Append(GetSize(node));
305         }
306 
307         protected abstract void PrintNodeValue(TElement node, StringBuilder sb);
```

```
308         }
309     }
```

Index