LinksPlatform's Platform.Collections.Methods Class Library

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

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

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

namespace Platform.Collections.Methods
{
    public abstract class GenericCollectionMethodsBase<TElement>
    {
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected virtual TElement GetZero() => default;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected virtual bool EqualToZero(TElement value) => EqualityComparer.Equals(value,
            Zero);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected virtual bool AreEqual(TElement first, TElement second) =>
            EqualityComparer.Equals(first, second);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected virtual bool GreaterThanZero(TElement value) => Comparer.Compare(value, Zero)
            > 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,
            Zero) >= 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,
            Zero) <= 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, Zero) < 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);

        protected readonly TElement Zero;
        protected readonly TElement One;
        protected readonly TElement Two;
        protected readonly EqualityComparer<TElement> EqualityComparer;
        protected readonly Comparer<TElement> Comparer;

        protected GenericCollectionMethodsBase()
        {
            EqualityComparer = EqualityComparer<TElement>.Default;
```

```
65            Comparer = Comparer<TElement>.Default;
66            Zero = GetZero(); //-V3068
67            One = Increment(Zero); //-V3068
68            Two = Increment(One); //-V3068
69        }
70     }
71  }
```

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

```
1   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
2
3   namespace Platform.Collections.Methods.Lists
4   {
5       public abstract class AbsoluteCircularDoublyLinkedListMethods<TElement> :
       ↪   AbsoluteDoublyLinkedListMethodsBase<TElement>
6       {
7           public void AttachBefore(TElement baseElement, TElement newElement)
8           {
9               var baseElementPrevious = GetPrevious(baseElement);
10              SetPrevious(newElement, baseElementPrevious);
11              SetNext(newElement, baseElement);
12              if (AreEqual(baseElement, GetFirst()))
13              {
14                  SetFirst(newElement);
15              }
16              SetNext(baseElementPrevious, newElement);
17              SetPrevious(baseElement, newElement);
18              IncrementSize();
19          }
20
21          public void AttachAfter(TElement baseElement, TElement newElement)
22          {
23              var baseElementNext = GetNext(baseElement);
24              SetPrevious(newElement, baseElement);
25              SetNext(newElement, baseElementNext);
26              if (AreEqual(baseElement, GetLast()))
27              {
28                  SetLast(newElement);
29              }
30              SetPrevious(baseElementNext, newElement);
31              SetNext(baseElement, newElement);
32              IncrementSize();
33          }
34
35          public void AttachAsFirst(TElement element)
36          {
37              var first = GetFirst();
38              if (EqualToZero(first))
39              {
40                  SetFirst(element);
41                  SetLast(element);
42                  SetPrevious(element, element);
43                  SetNext(element, element);
44                  IncrementSize();
45              }
46              else
47              {
48                  AttachBefore(first, element);
49              }
50          }
51
52          public void AttachAsLast(TElement element)
53          {
54              var last = GetLast();
55              if (EqualToZero(last))
56              {
57                  AttachAsFirst(element);
58              }
59              else
60              {
61                  AttachAfter(last, element);
62              }
63          }
64
65          public void Detach(TElement element)
66          {
67              var elementPrevious = GetPrevious(element);
68              var elementNext = GetNext(element);
```

```
69              if (AreEqual(elementNext, element))
70              {
71                  SetFirst(Zero);
72                  SetLast(Zero);
73              }
74              else
75              {
76                  SetNext(elementPrevious, elementNext);
77                  SetPrevious(elementNext, elementPrevious);
78                  if (AreEqual(element, GetFirst()))
79                  {
80                      SetFirst(elementNext);
81                  }
82                  if (AreEqual(element, GetLast()))
83                  {
84                      SetLast(elementPrevious);
85                  }
86              }
87              SetPrevious(element, Zero);
88              SetNext(element, Zero);
89              DecrementSize();
90          }
91      }
92  }
```

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

```
1   using System.Runtime.CompilerServices;
2
3   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
5   namespace Platform.Collections.Methods.Lists
6   {
7       public abstract class AbsoluteDoublyLinkedListMethodsBase<TElement> :
        ↪   DoublyLinkedListMethodsBase<TElement>
8       {
9           [MethodImpl(MethodImplOptions.AggressiveInlining)]
10          protected abstract TElement GetFirst();
11
12          [MethodImpl(MethodImplOptions.AggressiveInlining)]
13          protected abstract TElement GetLast();
14
15          [MethodImpl(MethodImplOptions.AggressiveInlining)]
16          protected abstract TElement GetSize();
17
18          [MethodImpl(MethodImplOptions.AggressiveInlining)]
19          protected abstract void SetFirst(TElement element);
20
21          [MethodImpl(MethodImplOptions.AggressiveInlining)]
22          protected abstract void SetLast(TElement element);
23
24          [MethodImpl(MethodImplOptions.AggressiveInlining)]
25          protected abstract void SetSize(TElement size);
26
27          [MethodImpl(MethodImplOptions.AggressiveInlining)]
28          protected void IncrementSize() => SetSize(Increment(GetSize()));
29
30          [MethodImpl(MethodImplOptions.AggressiveInlining)]
31          protected void DecrementSize() => SetSize(Decrement(GetSize()));
32      }
33  }
```

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

```
1   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
2
3   namespace Platform.Collections.Methods.Lists
4   {
5       public abstract class AbsoluteOpenDoublyLinkedListMethods<TElement> :
        ↪   AbsoluteDoublyLinkedListMethodsBase<TElement>
6       {
7           public void AttachBefore(TElement baseElement, TElement newElement)
8           {
9               var baseElementPrevious = GetPrevious(baseElement);
10              SetPrevious(newElement, baseElementPrevious);
11              SetNext(newElement, baseElement);
12              if (EqualToZero(baseElementPrevious))
13              {
14                  SetFirst(newElement);
15              }
16              else
```

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

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

        public void AttachAsFirst(TElement element)
        {
            var first = GetFirst();
            if (EqualToZero(first))
            {
                SetFirst(element);
                SetLast(element);
                SetPrevious(element, Zero);
                SetNext(element, Zero);
                IncrementSize();
            }
            else
            {
                AttachBefore(first, element);
            }
        }

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

        public void Detach(TElement element)
        {
            var elementPrevious = GetPrevious(element);
            var elementNext = GetNext(element);
            if (EqualToZero(elementPrevious))
            {
                SetFirst(elementNext);
            }
            else
            {
                SetNext(elementPrevious, elementNext);
            }
            if (EqualToZero(elementNext))
            {
                SetLast(elementPrevious);
            }
            else
            {
                SetPrevious(elementNext, elementPrevious);
            }
            SetPrevious(element, Zero);
            SetNext(element, Zero);
            DecrementSize();
        }
```

```
95          }
96    }
```

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

```csharp
1  using System.Runtime.CompilerServices;
2
3  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
5  namespace Platform.Collections.Methods.Lists
6  {
7      /// <remarks>
8      /// Based on <a href="https://en.wikipedia.org/wiki/Doubly_linked_list">doubly linked
9  ↪    list</a> implementation.
10     /// </remarks>
11     public abstract class DoublyLinkedListMethodsBase<TElement> :
      ↪    GenericCollectionMethodsBase<TElement>
12     {
13         [MethodImpl(MethodImplOptions.AggressiveInlining)]
14         protected abstract TElement GetPrevious(TElement element);
15
16         [MethodImpl(MethodImplOptions.AggressiveInlining)]
17         protected abstract TElement GetNext(TElement element);
18
19         [MethodImpl(MethodImplOptions.AggressiveInlining)]
20         protected abstract void SetPrevious(TElement element, TElement previous);
21
22         [MethodImpl(MethodImplOptions.AggressiveInlining)]
23         protected abstract void SetNext(TElement element, TElement next);
24     }
25 }
```

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

```csharp
1  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
2
3  namespace Platform.Collections.Methods.Lists
4  {
5      public abstract class RelativeCircularDoublyLinkedListMethods<TElement> :
      ↪    RelativeDoublyLinkedListMethodsBase<TElement>
6      {
7          public void AttachBefore(TElement headElement, TElement baseElement, TElement newElement)
8          {
9              var baseElementPrevious = GetPrevious(baseElement);
10             SetPrevious(newElement, baseElementPrevious);
11             SetNext(newElement, baseElement);
12             if (AreEqual(baseElement, GetFirst(headElement)))
13             {
14                 SetFirst(headElement, newElement);
15             }
16             SetNext(baseElementPrevious, newElement);
17             SetPrevious(baseElement, newElement);
18             IncrementSize(headElement);
19         }
20
21         public void AttachAfter(TElement headElement, TElement baseElement, TElement newElement)
22         {
23             var baseElementNext = GetNext(baseElement);
24             SetPrevious(newElement, baseElement);
25             SetNext(newElement, baseElementNext);
26             if (AreEqual(baseElement, GetLast(headElement)))
27             {
28                 SetLast(headElement, newElement);
29             }
30             SetPrevious(baseElementNext, newElement);
31             SetNext(baseElement, newElement);
32             IncrementSize(headElement);
33         }
34
35         public void AttachAsFirst(TElement headElement, TElement element)
36         {
37             var first = GetFirst(headElement);
38             if (EqualToZero(first))
39             {
40                 SetFirst(headElement, element);
41                 SetLast(headElement, element);
42                 SetPrevious(element, element);
43                 SetNext(element, element);
44                 IncrementSize(headElement);
45             }
```

```csharp
                    else
                    {
                        AttachBefore(headElement, first, element);
                    }
            }

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

            public void Detach(TElement headElement, TElement element)
            {
                var elementPrevious = GetPrevious(element);
                var elementNext = GetNext(element);
                if (AreEqual(elementNext, element))
                {
                    SetFirst(headElement, Zero);
                    SetLast(headElement, Zero);
                }
                else
                {
                    SetNext(elementPrevious, elementNext);
                    SetPrevious(elementNext, elementPrevious);
                    if (AreEqual(element, GetFirst(headElement)))
                    {
                        SetFirst(headElement, elementNext);
                    }
                    if (AreEqual(element, GetLast(headElement)))
                    {
                        SetLast(headElement, elementPrevious);
                    }
                }
                SetPrevious(element, Zero);
                SetNext(element, Zero);
                DecrementSize(headElement);
            }
        }
    }
```

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

```csharp
using System.Runtime.CompilerServices;

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

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

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

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

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

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

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

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

```
29
30             [MethodImpl(MethodImplOptions.AggressiveInlining)]
31             protected void DecrementSize(TElement headElement) => SetSize(headElement,
   ↪    Decrement(GetSize(headElement)));
32       }
33  }
```

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

```
1  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
2
3  namespace Platform.Collections.Methods.Lists
4  {
5      public abstract class RelativeOpenDoublyLinkedListMethods<TElement> :
   ↪    RelativeDoublyLinkedListMethodsBase<TElement>
6      {
7          public void AttachBefore(TElement headElement, TElement baseElement, TElement newElement)
8          {
9              var baseElementPrevious = GetPrevious(baseElement);
10             SetPrevious(newElement, baseElementPrevious);
11             SetNext(newElement, baseElement);
12             if (EqualToZero(baseElementPrevious))
13             {
14                 SetFirst(headElement, newElement);
15             }
16             else
17             {
18                 SetNext(baseElementPrevious, newElement);
19             }
20             SetPrevious(baseElement, newElement);
21             IncrementSize(headElement);
22         }
23
24         public void AttachAfter(TElement headElement, TElement baseElement, TElement newElement)
25         {
26             var baseElementNext = GetNext(baseElement);
27             SetPrevious(newElement, baseElement);
28             SetNext(newElement, baseElementNext);
29             if (EqualToZero(baseElementNext))
30             {
31                 SetLast(headElement, newElement);
32             }
33             else
34             {
35                 SetPrevious(baseElementNext, newElement);
36             }
37             SetNext(baseElement, newElement);
38             IncrementSize(headElement);
39         }
40
41         public void AttachAsFirst(TElement headElement, TElement element)
42         {
43             var first = GetFirst(headElement);
44             if (EqualToZero(first))
45             {
46                 SetFirst(headElement, element);
47                 SetLast(headElement, element);
48                 SetPrevious(element, Zero);
49                 SetNext(element, Zero);
50                 IncrementSize(headElement);
51             }
52             else
53             {
54                 AttachBefore(headElement, first, element);
55             }
56         }
57
58         public void AttachAsLast(TElement headElement, TElement element)
59         {
60             var last = GetLast(headElement);
61             if (EqualToZero(last))
62             {
63                 AttachAsFirst(headElement, element);
64             }
65             else
66             {
67                 AttachAfter(headElement, last, element);
68             }
69         }
```

```
71        public void Detach(TElement headElement, TElement element)
72        {
73            var elementPrevious = GetPrevious(element);
74            var elementNext = GetNext(element);
75            if (EqualToZero(elementPrevious))
76            {
77                SetFirst(headElement, elementNext);
78            }
79            else
80            {
81                SetNext(elementPrevious, elementNext);
82            }
83            if (EqualToZero(elementNext))
84            {
85                SetLast(headElement, elementPrevious);
86            }
87            else
88            {
89                SetPrevious(elementNext, elementPrevious);
90            }
91            SetPrevious(element, Zero);
92            SetNext(element, Zero);
93            DecrementSize(headElement);
94        }
95    }
96 }
```

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

```
1  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
2
3  namespace Platform.Collections.Methods.Trees
4  {
5      public abstract class RecursionlessSizeBalancedTreeMethods<TElement> :
       ↪  SizedBinaryTreeMethodsBase<TElement>
6      {
7          protected override void AttachCore(ref TElement root, TElement node)
8          {
9              while (true)
10             {
11                 ref var left = ref GetLeftReference(root);
12                 var leftSize = GetSizeOrZero(left);
13                 ref var right = ref GetRightReference(root);
14                 var rightSize = GetSizeOrZero(right);
15                 if (FirstIsToTheLeftOfSecond(node, root)) // node.Key less than root.Key
16                 {
17                     if (EqualToZero(left))
18                     {
19                         IncrementSize(root);
20                         SetSize(node, One);
21                         left = node;
22                         return;
23                     }
24                     if (FirstIsToTheLeftOfSecond(node, left)) // node.Key less than left.Key
25                     {
26                         if (GreaterThan(Increment(leftSize), rightSize))
27                         {
28                             RightRotate(ref root);
29                         }
30                         else
31                         {
32                             IncrementSize(root);
33                             root = ref left;
34                         }
35                     }
36                     else  // node.Key greater than left.Key
37                     {
38                         var leftRightSize = GetSizeOrZero(GetRight(left));
39                         if (GreaterThan(Increment(leftRightSize), rightSize))
40                         {
41                             if (EqualToZero(leftRightSize) && EqualToZero(rightSize))
42                             {
43                                 SetLeft(node, left);
44                                 SetRight(node, root);
45                                 SetSize(node, Add(leftSize, Two)); // Two (2) - node the size of
                                  ↪  root and a node itself
46                                 SetLeft(root, Zero);
47                                 SetSize(root, One);
```

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

        protected override void DetachCore(ref TElement root, TElement node)
        {
            while (true)
            {
                ref var left = ref GetLeftReference(root);
                var leftSize = GetSizeOrZero(left);
                ref var right = ref GetRightReference(root);
                var rightSize = GetSizeOrZero(right);
                if (FirstIsToTheLeftOfSecond(node, root)) // node.Key less than root.Key
                {
                    var decrementedLeftSize = Decrement(leftSize);
                    if (GreaterThan(GetSizeOrZero(GetRightOrDefault(right)),
                    ↪  decrementedLeftSize))
                    {
```

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

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

```csharp
using System;

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

namespace Platform.Collections.Methods.Trees
```

```csharp
{
    public abstract class SizeBalancedTreeMethods<TElement> :
    ↪   SizedBinaryTreeMethodsBase<TElement>
    {
        protected override void AttachCore(ref TElement root, TElement node)
        {
            if (EqualToZero(root))
            {
                root = node;
                IncrementSize(root);
            }
            else
            {
                IncrementSize(root);
                if (FirstIsToTheLeftOfSecond(node, root))
                {
                    AttachCore(ref GetLeftReference(root), node);
                    LeftMaintain(ref root);
                }
                else
                {
                    AttachCore(ref GetRightReference(root), node);
                    RightMaintain(ref root);
                }
            }
        }

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

```
 83                        {
 84                            root = replacementNode;
 85                        }
 86                        else if (AreEqual(GetLeft(parent), nodeToDetach))
 87                        {
 88                            SetLeft(parent, replacementNode);
 89                        }
 90                        else if (AreEqual(GetRight(parent), nodeToDetach))
 91                        {
 92                            SetRight(parent, replacementNode);
 93                        }
 94                        ClearNode(nodeToDetach);
 95                }

 97        private void LeftMaintain(ref TElement root)
 98        {
 99            if (!EqualToZero(root))
100            {
101                var rootLeftNode = GetLeft(root);
102                if (!EqualToZero(rootLeftNode))
103                {
104                    var rootRightNode = GetRight(root);
105                    var rootRightNodeSize = GetSize(rootRightNode);
106                    var rootLeftNodeLeftNode = GetLeft(rootLeftNode);
107                    if (!EqualToZero(rootLeftNodeLeftNode) &&
108                        (EqualToZero(rootRightNode) ||
                        ↪  GreaterThan(GetSize(rootLeftNodeLeftNode), rootRightNodeSize)))
109                    {
110                        RightRotate(ref root);
111                    }
112                    else
113                    {
114                        var rootLeftNodeRightNode = GetRight(rootLeftNode);
115                        if (!EqualToZero(rootLeftNodeRightNode) &&
116                            (EqualToZero(rootRightNode) ||
                            ↪  GreaterThan(GetSize(rootLeftNodeRightNode), rootRightNodeSize)))
117                        {
118                            LeftRotate(ref GetLeftReference(root));
119                            RightRotate(ref root);
120                        }
121                        else
122                        {
123                            return;
124                        }
125                    }
126                    LeftMaintain(ref GetLeftReference(root));
127                    RightMaintain(ref GetRightReference(root));
128                    LeftMaintain(ref root);
129                    RightMaintain(ref root);
130                }
131            }
132        }

134        private void RightMaintain(ref TElement root)
135        {
136            if (!EqualToZero(root))
137            {
138                var rootRightNode = GetRight(root);
139                if (!EqualToZero(rootRightNode))
140                {
141                    var rootLeftNode = GetLeft(root);
142                    var rootLeftNodeSize = GetSize(rootLeftNode);
143                    var rootRightNodeRightNode = GetRight(rootRightNode);
144                    if (!EqualToZero(rootRightNodeRightNode) &&
145                        (EqualToZero(rootLeftNode) ||
                        ↪  GreaterThan(GetSize(rootRightNodeRightNode), rootLeftNodeSize)))
146                    {
147                        LeftRotate(ref root);
148                    }
149                    else
150                    {
151                        var rootRightNodeLeftNode = GetLeft(rootRightNode);
152                        if (!EqualToZero(rootRightNodeLeftNode) &&
153                            (EqualToZero(rootLeftNode) ||
                            ↪  GreaterThan(GetSize(rootRightNodeLeftNode), rootLeftNodeSize)))
154                        {
155                            RightRotate(ref GetRightReference(root));
156                            LeftRotate(ref root);
```

```
157                         }
158                         else
159                         {
160                             return;
161                         }
162                     }
163                     LeftMaintain(ref GetLeftReference(root));
164                     RightMaintain(ref GetRightReference(root));
165                     LeftMaintain(ref root);
166                     RightMaintain(ref root);
167                 }
168             }
169         }
170     }
171 }
```

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

```csharp
1   using System;
2   using System.Runtime.CompilerServices;
3   using System.Text;
4   #if USEARRAYPOOL
5   using Platform.Collections;
6   #endif
7   using Platform.Reflection;
8
9   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
10
11  namespace Platform.Collections.Methods.Trees
12  {
13      /// <summary>
14      /// Combination of Size, Height (AVL), and threads.
15      /// </summary>
16      /// <remarks>
17      /// Based on: <a href="https://github.com/programmatom/TreeLib/blob/master/TreeLib/TreeLib/G⌋
        ↪   enerated/AVLTreeList.cs">TreeLib.AVLTreeList</a>.
18      /// Which itself based on: <a
        ↪   href="https://github.com/GNOME/glib/blob/master/glib/gtree.c">GNOME/glib/gtree</a>.
19      /// </remarks>
20      public abstract class SizedAndThreadedAVLBalancedTreeMethods<TElement> :
        ↪   SizedBinaryTreeMethodsBase<TElement>
21      {
22          private static readonly int _maxPath = 11 * NumericType<TElement>.BytesSize + 4;
23
24          [MethodImpl(MethodImplOptions.AggressiveInlining)]
25          protected override TElement GetRightest(TElement current)
26          {
27              var currentRight = GetRightOrDefault(current);
28              while (!EqualToZero(currentRight))
29              {
30                  current = currentRight;
31                  currentRight = GetRightOrDefault(current);
32              }
33              return current;
34          }
35
36          [MethodImpl(MethodImplOptions.AggressiveInlining)]
37          protected override TElement GetLeftest(TElement current)
38          {
39              var currentLeft = GetLeftOrDefault(current);
40              while (!EqualToZero(currentLeft))
41              {
42                  current = currentLeft;
43                  currentLeft = GetLeftOrDefault(current);
44              }
45              return current;
46          }
47
48          public override bool Contains(TElement node, TElement root)
49          {
50              while (!EqualToZero(root))
51              {
52                  if (FirstIsToTheLeftOfSecond(node, root)) // node.Key < root.Key
53                  {
54                      root = GetLeftOrDefault(root);
55                  }
56                  else if (FirstIsToTheRightOfSecond(node, root)) // node.Key > root.Key
57                  {
58                      root = GetRightOrDefault(root);
59                  }
```

```csharp
                       else // node.Key == root.Key
                       {
                           return true;
                       }
               }
               return false;
           }

           protected override void PrintNode(TElement node, StringBuilder sb, int level)
           {
               base.PrintNode(node, sb, level);
               sb.Append(' ');
               sb.Append(GetLeftIsChild(node) ? 'l' : 'L');
               sb.Append(GetRightIsChild(node) ? 'r' : 'R');
               sb.Append(' ');
               sb.Append(GetBalance(node));
           }

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

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

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

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

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

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

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

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

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

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

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

```csharp
                    {
                        // Threads
                        SetLeft(node, GetLeft(currentNode));
                        SetRight(node, currentNode);
                        SetLeft(currentNode, node);
                        SetLeftIsChild(currentNode, true);
                        DecrementBalance(currentNode);
                        SetSize(node, One);
                        FixSize(currentNode); // Should be incremented already
                        break;
                    }
                }
                else if (FirstIsToTheRightOfSecond(node, currentNode))
                {
                    if (GetRightIsChild(currentNode))
                    {
                        IncrementSize(currentNode);
                        path[pathPosition++] = currentNode;
                        currentNode = GetRight(currentNode);
                    }
                    else
                    {
                        // Threads
                        SetRight(node, GetRight(currentNode));
                        SetLeft(node, currentNode);
                        SetRight(currentNode, node);
                        SetRightIsChild(currentNode, true);
                        IncrementBalance(currentNode);
                        SetSize(node, One);
                        FixSize(currentNode); // Should be incremented already
                        break;
                    }
                }
                else
                {
                    throw new InvalidOperationException("Node with the same key already
                    ↪ attached to a tree.");
                }
            }
            // Restore balance. This is the goodness of a non-recursive
            // implementation, when we are done with balancing we 'break'
            // the loop and we are done.
            while (true)
            {
                var parent = path[--pathPosition];
                var isLeftNode = !AreEqual(parent, default) && AreEqual(currentNode,
                ↪ GetLeft(parent));
                var currentNodeBalance = GetBalance(currentNode);
                if (currentNodeBalance < -1 || currentNodeBalance > 1)
                {
                    currentNode = Balance(currentNode);
                    if (AreEqual(parent, default))
                    {
                        root = currentNode;
                    }
                    else if (isLeftNode)
                    {
                        SetLeft(parent, currentNode);
                        FixSize(parent);
                    }
                    else
                    {
                        SetRight(parent, currentNode);
                        FixSize(parent);
                    }
                }
                currentNodeBalance = GetBalance(currentNode);
                if (currentNodeBalance == 0 || AreEqual(parent, default))
                {
                    break;
                }
                if (isLeftNode)
                {
                    DecrementBalance(parent);
                }
                else
                {
                    IncrementBalance(parent);
```

```
                    }
                    currentNode = parent;
                }
#if USEARRAYPOOL
                ArrayPool.Free(path);
#endif
            }
        }

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

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

```csharp
                        SetBalance(right, (sbyte)(rightBalance - 1));
                    }
                    SetBalance(node, (sbyte)(rootBalance - rightBalance - 1));
                }
                return right;
            }
        }

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

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

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

```csharp
            protected override void DetachCore(ref TElement root, TElement node)
            {
                unchecked
                {
#if USEARRAYPOOL
                    var path = ArrayPool.Allocate<TElement>(MaxPath);
                    var pathPosition = 0;
                    path[pathPosition++] = default;
#else
                    var path = new TElement[_maxPath];
                    var pathPosition = 1;
#endif
                    var currentNode = root;
                    while (true)
                    {
                        if (FirstIsToTheLeftOfSecond(node, currentNode))
                        {
                            if (!GetLeftIsChild(currentNode))
                            {
                                throw new InvalidOperationException("Cannot find a node.");
                            }
                            DecrementSize(currentNode);
                            path[pathPosition++] = currentNode;
                            currentNode = GetLeft(currentNode);
                        }
                        else if (FirstIsToTheRightOfSecond(node, currentNode))
                        {
                            if (!GetRightIsChild(currentNode))
                            {
                                throw new InvalidOperationException("Cannot find a node.");
                            }
                            DecrementSize(currentNode);
                            path[pathPosition++] = currentNode;
                            currentNode = GetRight(currentNode);
                        }
                        else
                        {
                            break;
                        }
                    }
                    var parent = path[--pathPosition];
                    var balanceNode = parent;
                    var isLeftNode = !AreEqual(parent, default) && AreEqual(currentNode,
                    ↪ GetLeft(parent));
                    if (!GetLeftIsChild(currentNode))
                    {
                        if (!GetRightIsChild(currentNode)) // node has no children
                        {
                            if (AreEqual(parent, default))
                            {
                                root = Zero;
                            }
                            else if (isLeftNode)
                            {
                                SetLeftIsChild(parent, false);
                                SetLeft(parent, GetLeft(currentNode));
                                IncrementBalance(parent);
                            }
                            else
                            {
                                SetRightIsChild(parent, false);
                                SetRight(parent, GetRight(currentNode));
                                DecrementBalance(parent);
                            }
                        }
                        else // node has a right child
                        {
                            var successor = GetNext(currentNode);
                            SetLeft(successor, GetLeft(currentNode));
                            var right = GetRight(currentNode);
                            if (AreEqual(parent, default))
                            {
                                root = right;
                            }
                            else if (isLeftNode)
                            {
                                SetLeft(parent, right);
                                IncrementBalance(parent);
                            }
```

```
446                        else
447                        {
448                            SetRight(parent, right);
449                            DecrementBalance(parent);
450                        }
451                    }
452                }
453                else // node has a left child
454                {
455                    if (!GetRightIsChild(currentNode))
456                    {
457                        var predecessor = GetPrevious(currentNode);
458                        SetRight(predecessor, GetRight(currentNode));
459                        var leftValue = GetLeft(currentNode);
460                        if (AreEqual(parent, default))
461                        {
462                            root = leftValue;
463                        }
464                        else if (isLeftNode)
465                        {
466                            SetLeft(parent, leftValue);
467                            IncrementBalance(parent);
468                        }
469                        else
470                        {
471                            SetRight(parent, leftValue);
472                            DecrementBalance(parent);
473                        }
474                    }
475                    else // node has a both children (left and right)
476                    {
477                        var predecessor = GetLeft(currentNode);
478                        var successor = GetRight(currentNode);
479                        var successorParent = currentNode;
480                        int previousPathPosition = ++pathPosition;
481                        // find the immediately next node (and its parent)
482                        while (GetLeftIsChild(successor))
483                        {
484                            path[++pathPosition] = successorParent = successor;
485                            successor = GetLeft(successor);
486                            if (!AreEqual(successorParent, currentNode))
487                            {
488                                DecrementSize(successorParent);
489                            }
490                        }
491                        path[previousPathPosition] = successor;
492                        balanceNode = path[pathPosition];
493                        // remove 'successor' from the tree
494                        if (!AreEqual(successorParent, currentNode))
495                        {
496                            if (!GetRightIsChild(successor))
497                            {
498                                SetLeftIsChild(successorParent, false);
499                            }
500                            else
501                            {
502                                SetLeft(successorParent, GetRight(successor));
503                            }
504                            IncrementBalance(successorParent);
505                            SetRightIsChild(successor, true);
506                            SetRight(successor, GetRight(currentNode));
507                        }
508                        else
509                        {
510                            DecrementBalance(currentNode);
511                        }
512                        // set the predecessor's successor link to point to the right place
513                        while (GetRightIsChild(predecessor))
514                        {
515                            predecessor = GetRight(predecessor);
516                        }
517                        SetRight(predecessor, successor);
518                        // prepare 'successor' to replace 'node'
519                        var left = GetLeft(currentNode);
520                        SetLeftIsChild(successor, true);
521                        SetLeft(successor, left);
522                        SetBalance(successor, GetBalance(currentNode));
523                        FixSize(successor);
```

```csharp
                            if (AreEqual(parent, default))
                            {
                                root = successor;
                            }
                            else if (isLeftNode)
                            {
                                SetLeft(parent, successor);
                            }
                            else
                            {
                                SetRight(parent, successor);
                            }
                        }
                    }
                    // restore balance
                    if (!AreEqual(balanceNode, default))
                    {
                        while (true)
                        {
                            var balanceParent = path[--pathPosition];
                            isLeftNode = !AreEqual(balanceParent, default) && AreEqual(balanceNode,
                            ↳   GetLeft(balanceParent));
                            var currentNodeBalance = GetBalance(balanceNode);
                            if (currentNodeBalance < -1 || currentNodeBalance > 1)
                            {
                                balanceNode = Balance(balanceNode);
                                if (AreEqual(balanceParent, default))
                                {
                                    root = balanceNode;
                                }
                                else if (isLeftNode)
                                {
                                    SetLeft(balanceParent, balanceNode);
                                }
                                else
                                {
                                    SetRight(balanceParent, balanceNode);
                                }
                            }
                            currentNodeBalance = GetBalance(balanceNode);
                            if (currentNodeBalance != 0 || AreEqual(balanceParent, default))
                            {
                                break;
                            }
                            if (isLeftNode)
                            {
                                IncrementBalance(balanceParent);
                            }
                            else
                            {
                                DecrementBalance(balanceParent);
                            }
                            balanceNode = balanceParent;
                        }
                    }
                    ClearNode(node);
#if USEARRAYPOOL
                    ArrayPool.Free(path);
#endif
                }
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void ClearNode(TElement node)
        {
            SetLeft(node, Zero);
            SetRight(node, Zero);
            SetSize(node, Zero);
            SetLeftIsChild(node, false);
            SetRightIsChild(node, false);
            SetBalance(node, 0);
        }
    }
}
```

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

```csharp
//#define ENABLE_TREE_AUTO_DEBUG_AND_VALIDATION

using System;
```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected TElement LeftRotate(TElement root)
        {
            var right = GetRight(root);
#if ENABLE_TREE_AUTO_DEBUG_AND_VALIDATION
            if (EqualToZero(right))
            {
```

```csharp
                    throw new InvalidOperationException("Right is null.");
                }
#endif
            SetRight(root, GetLeft(right));
            SetLeft(right, root);
            SetSize(right, GetSize(root));
            FixSize(root);
            return right;
        }

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

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

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

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

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

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public virtual bool Contains(TElement node, TElement root)
        {
            while (!EqualToZero(root))
            {
                if (FirstIsToTheLeftOfSecond(node, root)) // node.Key < root.Key
                {
                    root = GetLeft(root);
                }
                else if (FirstIsToTheRightOfSecond(node, root)) // node.Key > root.Key
                {
                    root = GetRight(root);
                }
                else // node.Key == root.Key
                {
                    return true;
                }
            }
            return false;
```

```csharp
            }

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

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

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

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

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

            public void FixSizes(TElement node)
            {
                if (AreEqual(node, default))
                {
                    return;
                }
```

```csharp
                FixSizes(GetLeft(node));
                FixSizes(GetRight(node));
                FixSize(node);
        }

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

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

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

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

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

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

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

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

```csharp
312
313            protected abstract void PrintNodeValue(TElement node, StringBuilder sb);
314        }
315    }
```

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

```csharp
1   using System;
2   using System.Collections.Generic;
3   using System.Text;
4   using Platform.Numbers;
5   using Platform.Collections.Methods.Trees;
6   using Platform.Converters;
7
8   namespace Platform.Collections.Methods.Tests
9   {
10      public class RecursionlessSizeBalancedTree<TElement> :
         ↪  RecursionlessSizeBalancedTreeMethods<TElement>
11      {
12          private struct TreeElement
13          {
14              public TElement Size;
15              public TElement Left;
16              public TElement Right;
17          }
18
19          private readonly TreeElement[] _elements;
20          private TElement _allocated;
21
22          public TElement Root;
23
24          public TElement Count => GetSizeOrZero(Root);
25
26          public RecursionlessSizeBalancedTree(int capacity) => (_elements, _allocated) = (new
         ↪  TreeElement[capacity], One);
27
28          public TElement Allocate()
29          {
30              var newNode = _allocated;
31              if (IsEmpty(newNode))
32              {
33                  _allocated = Arithmetic.Increment(_allocated);
34                  return newNode;
35              }
36              else
37              {
38                  throw new InvalidOperationException("Allocated tree element is not empty.");
39              }
40          }
41
42          public void Free(TElement node)
43          {
44              while (!EqualityComparer.Equals(_allocated, One) && IsEmpty(node))
45              {
46                  var lastNode = Arithmetic.Decrement(_allocated);
47                  if (EqualityComparer.Equals(lastNode, node))
48                  {
49                      _allocated = lastNode;
50                      node = Arithmetic.Decrement(node);
51                  }
52                  else
53                  {
54                      return;
55                  }
56              }
57          }
58
59          public bool IsEmpty(TElement node) =>
         ↪  EqualityComparer<TreeElement>.Default.Equals(GetElement(node), default);
60
61          protected override bool FirstIsToTheLeftOfSecond(TElement first, TElement second) =>
         ↪  Comparer.Compare(first, second) < 0;
62
63          protected override bool FirstIsToTheRightOfSecond(TElement first, TElement second) =>
         ↪  Comparer.Compare(first, second) > 0;
64
65          protected override ref TElement GetLeftReference(TElement node) => ref
         ↪  GetElement(node).Left;
66
67          protected override TElement GetLeft(TElement node) => GetElement(node).Left;
68
```

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

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

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

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

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

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

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

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

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

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

namespace Platform.Collections.Methods.Tests
{
    public class SizeBalancedTree<TElement> : SizeBalancedTreeMethods<TElement>
    {
        private struct TreeElement
        {
            public TElement Size;
            public TElement Left;
            public TElement Right;
        }

        private readonly TreeElement[] _elements;
        private TElement _allocated;

        public TElement Root;

        public TElement Count => GetSizeOrZero(Root);

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

        public TElement Allocate()
        {
            var newNode = _allocated;
            if (IsEmpty(newNode))
            {
                _allocated = Arithmetic.Increment(_allocated);
                return newNode;
            }
            else
            {
                throw new InvalidOperationException("Allocated tree element is not empty.");
            }
        }

        public void Free(TElement node)
        {
            while (!EqualityComparer.Equals(_allocated, One) && IsEmpty(node))
            {
                var lastNode = Arithmetic.Decrement(_allocated);
                if (EqualityComparer.Equals(lastNode, node))
                {
                    _allocated = lastNode;
                    node = Arithmetic.Decrement(node);
                }
                else
                {
                    return;
```

```
55                    }
56                }
57            }
58
59        public bool IsEmpty(TElement node) =>
            ↪  EqualityComparer<TreeElement>.Default.Equals(GetElement(node), default);
60
61        protected override bool FirstIsToTheLeftOfSecond(TElement first, TElement second) =>
            ↪  Comparer.Compare(first, second) < 0;
62
63        protected override bool FirstIsToTheRightOfSecond(TElement first, TElement second) =>
            ↪  Comparer.Compare(first, second) > 0;
64
65        protected override ref TElement GetLeftReference(TElement node) => ref
            ↪  GetElement(node).Left;
66
67        protected override TElement GetLeft(TElement node) => GetElement(node).Left;
68
69        protected override ref TElement GetRightReference(TElement node) => ref
            ↪  GetElement(node).Right;
70
71        protected override TElement GetRight(TElement node) => GetElement(node).Right;
72
73        protected override TElement GetSize(TElement node) => GetElement(node).Size;
74
75        protected override void PrintNodeValue(TElement node, StringBuilder sb) =>
            ↪  sb.Append(node);
76
77        protected override void SetLeft(TElement node, TElement left) => GetElement(node).Left =
            ↪  left;
78
79        protected override void SetRight(TElement node, TElement right) =>
            ↪  GetElement(node).Right = right;
80
81        protected override void SetSize(TElement node, TElement size) => GetElement(node).Size =
            ↪  size;
82
83        private ref TreeElement GetElement(TElement node) => ref
            ↪  _elements[UncheckedConverter<TElement, long>.Default.Convert(node)];
84    }
85 }
```

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

```
1  using System;
2  using System.Collections.Generic;
3  using System.Text;
4  using Platform.Numbers;
5  using Platform.Collections.Methods.Trees;
6  using Platform.Converters;
7
8  namespace Platform.Collections.Methods.Tests
9  {
10     public class SizedAndThreadedAVLBalancedTree<TElement> :
       ↪  SizedAndThreadedAVLBalancedTreeMethods<TElement>
11     {
12         private struct TreeElement
13         {
14             public TElement Size;
15             public TElement Left;
16             public TElement Right;
17             public sbyte Balance;
18             public bool LeftIsChild;
19             public bool RightIsChild;
20         }
21
22         private readonly TreeElement[] _elements;
23         private TElement _allocated;
24
25         public TElement Root;
26
27         public TElement Count => GetSizeOrZero(Root);
28
29         public SizedAndThreadedAVLBalancedTree(int capacity) => (_elements, _allocated) = (new
       ↪  TreeElement[capacity], One);
30
31         public TElement Allocate()
32         {
33             var newNode = _allocated;
34             if (IsEmpty(newNode))
35             {
```

```csharp
                _allocated = Arithmetic.Increment(_allocated);
                return newNode;
            }
            else
            {
                throw new InvalidOperationException("Allocated tree element is not empty.");
            }
        }

        public void Free(TElement node)
        {
            while (!EqualityComparer.Equals(_allocated, One) && IsEmpty(node))
            {
                var lastNode = Arithmetic.Decrement(_allocated);
                if (EqualityComparer.Equals(lastNode, node))
                {
                    _allocated = lastNode;
                    node = Arithmetic.Decrement(node);
                }
                else
                {
                    return;
                }
            }
        }

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

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

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

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

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

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

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

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

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

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

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

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

        protected override void SetBalance(TElement node, sbyte value) =>
            GetElement(node).Balance = value;

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

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

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

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

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

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

```csharp
1   using System;
2   using System.Collections.Generic;
3   using Xunit;
4   using Platform.Collections.Methods.Trees;
5   using Platform.Converters;
6
7   namespace Platform.Collections.Methods.Tests
8   {
9       public static class TestExtensions
10      {
11          public static void TestMultipleCreationsAndDeletions<TElement>(this
            ↪  SizedBinaryTreeMethodsBase<TElement> tree, Func<TElement> allocate, Action<TElement>
            ↪  free, ref TElement root, Func<TElement> treeCount, int maximumOperationsPerCycle)
12          {
13              for (var N = 1; N < maximumOperationsPerCycle; N++)
14              {
15                  var currentCount = 0;
16                  for (var i = 0; i < N; i++)
17                  {
18                      var node = allocate();
19                      tree.Attach(ref root, node);
20                      currentCount++;
21                      Assert.Equal(currentCount, (int)UncheckedConverter<TElement,
                        ↪  int>.Default.Convert(treeCount()));
22                  }
23                  for (var i = 1; i <= N; i++)
24                  {
25                      TElement node = UncheckedConverter<int, TElement>.Default.Convert(i);
26                      if (tree.Contains(node, root))
27                      {
28                          tree.Detach(ref root, node);
29                          free(node);
30                          currentCount--;
31                          Assert.Equal(currentCount, (int)UncheckedConverter<TElement,
                            ↪  int>.Default.Convert(treeCount()));
32                      }
33                  }
34              }
35          }
36
37          public static void TestMultipleRandomCreationsAndDeletions<TElement>(this
            ↪  SizedBinaryTreeMethodsBase<TElement> tree, ref TElement root, Func<TElement>
            ↪  treeCount, int maximumOperationsPerCycle)
38          {
39              var random = new System.Random(0);
40              var added = new HashSet<TElement>();
41              var currentCount = 0;
42              for (var N = 1; N < maximumOperationsPerCycle; N++)
43              {
44                  for (var i = 0; i < N; i++)
45                  {
46                      var node = UncheckedConverter<int, TElement>.Default.Convert(random.Next(1,
                        ↪  N));
47                      if (added.Add(node))
48                      {
49                          tree.Attach(ref root, node);
50                          currentCount++;
51                          Assert.Equal(currentCount, (int)UncheckedConverter<TElement,
                            ↪  int>.Default.Convert(treeCount()));
52                      }
53                  }
54                  for (var i = 1; i <= N; i++)
55                  {
56                      TElement node = UncheckedConverter<int,
                        ↪  TElement>.Default.Convert(random.Next(1, N));
57                      if (tree.Contains(node, root))
58                      {
59                          tree.Detach(ref root, node);
60                          currentCount--;
61                          Assert.Equal(currentCount, (int)UncheckedConverter<TElement,
                            ↪  int>.Default.Convert(treeCount()));
62                          added.Remove(node);
63                      }
64                  }
65              }
66          }
67      }
```

```
68    }
```

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

```csharp
1    using Xunit;
2
3    namespace Platform.Collections.Methods.Tests
4    {
5        public static class TreesTests
6        {
7            private const int _n = 500;
8
9            [Fact]
10           public static void RecursionlessSizeBalancedTreeMultipleAttachAndDetachTest()
11           {
12               var recursionlessSizeBalancedTree = new RecursionlessSizeBalancedTree<uint>(10000);
13               recursionlessSizeBalancedTree.TestMultipleCreationsAndDeletions(recursionlessSizeBal
           ↪    ancedTree.Allocate, recursionlessSizeBalancedTree.Free, ref
           ↪    recursionlessSizeBalancedTree.Root, () => recursionlessSizeBalancedTree.Count,
           ↪    _n);
14           }
15
16           [Fact]
17           public static void SizeBalancedTreeMultipleAttachAndDetachTest()
18           {
19               var sizeBalancedTree = new SizeBalancedTree<uint>(10000);
20               sizeBalancedTree.TestMultipleCreationsAndDeletions(sizeBalancedTree.Allocate,
           ↪    sizeBalancedTree.Free, ref sizeBalancedTree.Root, () => sizeBalancedTree.Count,
           ↪    _n);
21           }
22
23           [Fact]
24           public static void SizedAndThreadedAVLBalancedTreeMultipleAttachAndDetachTest()
25           {
26               var avlTree = new SizedAndThreadedAVLBalancedTree<uint>(10000);
27               avlTree.TestMultipleCreationsAndDeletions(avlTree.Allocate, avlTree.Free, ref
           ↪    avlTree.Root, () => avlTree.Count, _n);
28           }
29
30           [Fact]
31           public static void RecursionlessSizeBalancedTreeMultipleRandomAttachAndDetachTest()
32           {
33               var recursionlessSizeBalancedTree = new RecursionlessSizeBalancedTree<uint>(10000);
34               recursionlessSizeBalancedTree.TestMultipleRandomCreationsAndDeletions(ref
           ↪    recursionlessSizeBalancedTree.Root, () => recursionlessSizeBalancedTree.Count,
           ↪    _n);
35           }
36
37           [Fact]
38           public static void SizeBalancedTreeMultipleRandomAttachAndDetachTest()
39           {
40               var sizeBalancedTree = new SizeBalancedTree<uint>(10000);
41               sizeBalancedTree.TestMultipleRandomCreationsAndDeletions(ref sizeBalancedTree.Root,
           ↪    () => sizeBalancedTree.Count, _n);
42           }
43
44           [Fact]
45           public static void SizedAndThreadedAVLBalancedTreeMultipleRandomAttachAndDetachTest()
46           {
47               var avlTree = new SizedAndThreadedAVLBalancedTree<uint>(10000);
48               avlTree.TestMultipleRandomCreationsAndDeletions(ref avlTree.Root, () =>
           ↪    avlTree.Count, _n);
49           }
50       }
51    }
```

Index