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

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

namespace Platform.Collections.Methods
{
    /// <summary>
    /// <para>Represents a base implementation of methods for a collection of elements of type
    ///     TElement.</para>
    /// <para>Представляет базовую реализацию методов коллекции элементов типа TElement.</para>
    /// </summary>
    public abstract class GenericCollectionMethodsBase<TElement>
    {
        /// <summary>
        /// <para>Presents the Range in readable format.</para>
        /// <para>Представляет диапазон в удобном для чтения формате.</para>
        /// </summary>
        /// <returns><para>String representation of the Range.</para><para>Строковое
        ///     представление диапазона.</para></returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected virtual TElement GetZero() => default;

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

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

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

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

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

        /// <summary>
```

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

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

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

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

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

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

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

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

```csharp
126
127          /// <summary>
128          /// <para>Presents the Range in readable format.</para>
129          /// <para>Представляет диапазон в удобном для чтения формате.</para>
130          /// </summary>
131          /// <returns><para>String representation of the Range.</para><para>Строковое
             ↪  представление диапазона.</para></returns>
132          [MethodImpl(MethodImplOptions.AggressiveInlining)]
133          protected virtual TElement Subtract(TElement first, TElement second) =>
             ↪  Arithmetic<TElement>.Subtract(first, second);
134
135          /// <summary>
136          /// <para>Returns minimum value of the range.</para>
137          /// <para>Возвращает минимальное значение диапазона.</para>
138          /// </summary>
139          protected readonly TElement Zero;
140
141          /// <summary>
142          /// <para>Returns minimum value of the range.</para>
143          /// <para>Возвращает минимальное значение диапазона.</para>
144          /// </summary>
145          protected readonly TElement One;
146
147           /// <summary>
148          /// <para>Returns minimum value of the range.</para>
149          /// <para>Возвращает минимальное значение диапазона.</para>
150          /// </summary>
151          protected readonly TElement Two;
152
153          /// <summary>
154          /// <para>Returns minimum value of the range.</para>
155          /// <para>Возвращает минимальное значение диапазона.</para>
156          /// </summary>
157          protected readonly EqualityComparer<TElement> EqualityComparer;
158
159          /// <summary>
160          /// <para>Returns minimum value of the range.</para>
161          /// <para>Возвращает минимальное значение диапазона.</para>
162          /// </summary>
163          protected readonly Comparer<TElement> Comparer;
164
165          /// <summary>
166          /// <para>Presents the Range in readable format.</para>
167          /// <para>Представляет диапазон в удобном для чтения формате.</para>
168          /// </summary>
169          /// <returns><para>String representation of the Range.</para><para>Строковое
             ↪  представление диапазона.</para></returns>
170          protected GenericCollectionMethodsBase()
171          {
172              EqualityComparer = EqualityComparer<TElement>.Default;
173              Comparer = Comparer<TElement>.Default;
174              Zero = GetZero(); //-V3068
175              One = Increment(Zero); //-V3068
176              Two = Increment(One); //-V3068
177          }
178      }
179  }
```

1.2 ./csharp/Platform.Collections.Methods/Lists/AbsoluteCircularDoublyLinkedListMethods.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 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         }
```

```csharp
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

```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      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
17             {
18                 SetNext(baseElementPrevious, newElement);
19             }
20             SetPrevious(baseElement, newElement);
21             IncrementSize();
22         }
23
24         public void AttachAfter(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(newElement);
32             }
33             else
34             {
35                 SetPrevious(baseElementNext, newElement);
36             }
37             SetNext(baseElement, newElement);
38             IncrementSize();
39         }
40
41         public void AttachAsFirst(TElement element)
42         {
43             var first = GetFirst();
44             if (EqualToZero(first))
45             {
```

```csharp
                    SetFirst(element);
                    SetLast(element);
                    SetPrevious(element, Zero);
                    SetNext(element, Zero);
                    IncrementSize();
                }
                else
                {
                    AttachBefore(first, element);
                }
            }

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

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

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

```csharp
using System.Runtime.CompilerServices;

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

namespace Platform.Collections.Methods.Lists
{
    /// <remarks>
    /// Based on <a href="https://en.wikipedia.org/wiki/Doubly_linked_list">doubly linked
    ///     list</a> implementation.
    /// </remarks>
    public abstract class DoublyLinkedListMethodsBase<TElement> :
        GenericCollectionMethodsBase<TElement>
    {
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected abstract TElement GetPrevious(TElement element);

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

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

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

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

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

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

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

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

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

```
                            SetPrevious(elementNext, elementPrevious);
                            if (AreEqual(element, GetFirst(headElement)))
                            {
                                SetFirst(headElement, elementNext);
                            }
                            if (AreEqual(element, GetLast(headElement)))
                            {
                                SetLast(headElement, elementPrevious);
                            }
                        }
                        SetPrevious(element, Zero);
                        SetNext(element, Zero);
                        DecrementSize(headElement);
                    }
                }
            }
```

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

```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)));

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

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

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

namespace Platform.Collections.Methods.Lists
{
    public abstract class RelativeOpenDoublyLinkedListMethods<TElement> :
    ↪  RelativeDoublyLinkedListMethodsBase<TElement>
    {
        public void AttachBefore(TElement headElement, TElement baseElement, TElement newElement)
        {
            var baseElementPrevious = GetPrevious(baseElement);
            SetPrevious(newElement, baseElementPrevious);
            SetNext(newElement, baseElement);
            if (EqualToZero(baseElementPrevious))
            {
                SetFirst(headElement, newElement);
            }
            else
            {
                SetNext(baseElementPrevious, newElement);
            }
            SetPrevious(baseElement, newElement);
            IncrementSize(headElement);
```

```csharp
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          }
70
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

```csharp
1  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
2
```

```csharp
namespace Platform.Collections.Methods.Trees
{
    public abstract class RecursionlessSizeBalancedTreeMethods<TElement> :
        SizedBinaryTreeMethodsBase<TElement>
    {
        protected override void AttachCore(ref TElement root, TElement node)
        {
            while (true)
            {
                ref var left = ref GetLeftReference(root);
                var leftSize = GetSizeOrZero(left);
                ref var right = ref GetRightReference(root);
                var rightSize = GetSizeOrZero(right);
                if (FirstIsToTheLeftOfSecond(node, root)) // node.Key less than root.Key
                {
                    if (EqualToZero(left))
                    {
                        IncrementSize(root);
                        SetSize(node, One);
                        left = node;
                        return;
                    }
                    if (FirstIsToTheLeftOfSecond(node, left)) // node.Key less than left.Key
                    {
                        if (GreaterThan(Increment(leftSize), rightSize))
                        {
                            RightRotate(ref root);
                        }
                        else
                        {
                            IncrementSize(root);
                            root = ref left;
                        }
                    }
                    else  // node.Key greater than left.Key
                    {
                        var leftRightSize = GetSizeOrZero(GetRight(left));
                        if (GreaterThan(Increment(leftRightSize), rightSize))
                        {
                            if (EqualToZero(leftRightSize) && EqualToZero(rightSize))
                            {
                                SetLeft(node, left);
                                SetRight(node, root);
                                SetSize(node, Add(leftSize, Two)); // Two (2) - node the size of
                                                                   //  root and a node itself
                                SetLeft(root, Zero);
                                SetSize(root, One);
                                root = node;
                                return;
                            }
                            LeftRotate(ref left);
                            RightRotate(ref root);
                        }
                        else
                        {
                            IncrementSize(root);
                            root = ref left;
                        }
                    }
                }
                else // node.Key greater than root.Key
                {
                    if (EqualToZero(right))
                    {
                        IncrementSize(root);
                        SetSize(node, One);
                        right = node;
                        return;
                    }
                    if (FirstIsToTheRightOfSecond(node, right)) // node.Key greater than
                                                               //  right.Key
                    {
                        if (GreaterThan(Increment(rightSize), leftSize))
                        {
                            LeftRotate(ref root);
                        }
                        else
                        {
                            IncrementSize(root);
```

```
 79                            root = ref right;
 80                        }
 81                    }
 82                    else // node.Key less than right.Key
 83                    {
 84                        var rightLeftSize = GetSizeOrZero(GetLeft(right));
 85                        if (GreaterThan(Increment(rightLeftSize), leftSize))
 86                        {
 87                            if (EqualToZero(rightLeftSize) && EqualToZero(leftSize))
 88                            {
 89                                SetLeft(node, root);
 90                                SetRight(node, right);
 91                                SetSize(node, Add(rightSize, Two)); // Two (2) - node the size
                                  ↪  of root and a node itself
 92                                SetRight(root, Zero);
 93                                SetSize(root, One);
 94                                root = node;
 95                                return;
 96                            }
 97                            RightRotate(ref right);
 98                            LeftRotate(ref root);
 99                        }
100                        else
101                        {
102                            IncrementSize(root);
103                            root = ref right;
104                        }
105                    }
106                }
107            }
108        }
109
110        protected override void DetachCore(ref TElement root, TElement node)
111        {
112            while (true)
113            {
114                ref var left = ref GetLeftReference(root);
115                var leftSize = GetSizeOrZero(left);
116                ref var right = ref GetRightReference(root);
117                var rightSize = GetSizeOrZero(right);
118                if (FirstIsToTheLeftOfSecond(node, root)) // node.Key less than root.Key
119                {
120                    var decrementedLeftSize = Decrement(leftSize);
121                    if (GreaterThan(GetSizeOrZero(GetRightOrDefault(right)),
                          ↪  decrementedLeftSize))
122                    {
123                        LeftRotate(ref root);
124                    }
125                    else if (GreaterThan(GetSizeOrZero(GetLeftOrDefault(right)),
                          ↪  decrementedLeftSize))
126                    {
127                        RightRotate(ref right);
128                        LeftRotate(ref root);
129                    }
130                    else
131                    {
132                        DecrementSize(root);
133                        root = ref left;
134                    }
135                }
136                else if (FirstIsToTheRightOfSecond(node, root)) // node.Key greater than root.Key
137                {
138                    var decrementedRightSize = Decrement(rightSize);
139                    if (GreaterThan(GetSizeOrZero(GetLeftOrDefault(left)), decrementedRightSize))
140                    {
141                        RightRotate(ref root);
142                    }
143                    else if (GreaterThan(GetSizeOrZero(GetRightOrDefault(left)),
                          ↪  decrementedRightSize))
144                    {
145                        LeftRotate(ref left);
146                        RightRotate(ref root);
147                    }
148                    else
149                    {
150                        DecrementSize(root);
151                        root = ref right;
152                    }
```

```
153                     }
154                     else // key equals to root.Key
155                     {
156                         if (GreaterThanZero(leftSize) && GreaterThanZero(rightSize))
157                         {
158                             TElement replacement;
159                             if (GreaterThan(leftSize, rightSize))
160                             {
161                                 replacement = GetRightest(left);
162                                 DetachCore(ref left, replacement);
163                             }
164                             else
165                             {
166                                 replacement = GetLeftest(right);
167                                 DetachCore(ref right, replacement);
168                             }
169                             SetLeft(replacement, left);
170                             SetRight(replacement, right);
171                             SetSize(replacement, Add(leftSize, rightSize));
172                             root = replacement;
173                         }
174                         else if (GreaterThanZero(leftSize))
175                         {
176                             root = left;
177                         }
178                         else if (GreaterThanZero(rightSize))
179                         {
180                             root = right;
181                         }
182                         else
183                         {
184                             root = Zero;
185                         }
186                         ClearNode(node);
187                         return;
188                     }
189                 }
190             }
191         }
192 }
```

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

```csharp
1  using System;
2
3  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
5  namespace Platform.Collections.Methods.Trees
6  {
7      public abstract class SizeBalancedTreeMethods<TElement> :
       ↪  SizedBinaryTreeMethodsBase<TElement>
8      {
9          protected override void AttachCore(ref TElement root, TElement node)
10         {
11             if (EqualToZero(root))
12             {
13                 root = node;
14                 IncrementSize(root);
15             }
16             else
17             {
18                 IncrementSize(root);
19                 if (FirstIsToTheLeftOfSecond(node, root))
20                 {
21                     AttachCore(ref GetLeftReference(root), node);
22                     LeftMaintain(ref root);
23                 }
24                 else
25                 {
26                     AttachCore(ref GetRightReference(root), node);
27                     RightMaintain(ref root);
28                 }
29             }
30         }
31
32         protected override void DetachCore(ref TElement root, TElement nodeToDetach)
33         {
34             ref var currentNode = ref root;
35             ref var parent = ref root;
36             var replacementNode = Zero;
```

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

        private void LeftMaintain(ref TElement root)
        {
            if (!EqualToZero(root))
            {
                var rootLeftNode = GetLeft(root);
                if (!EqualToZero(rootLeftNode))
                {
                    var rootRightNode = GetRight(root);
                    var rootRightNodeSize = GetSize(rootRightNode);
                    var rootLeftNodeLeftNode = GetLeft(rootLeftNode);
                    if (!EqualToZero(rootLeftNodeLeftNode) &&
                        (EqualToZero(rootRightNode) ||
                          ↪ GreaterThan(GetSize(rootLeftNodeLeftNode), rootRightNodeSize)))
                    {
                        RightRotate(ref root);
                    }
                    else
```

```csharp
                    {
                        var rootLeftNodeRightNode = GetRight(rootLeftNode);
                        if (!EqualToZero(rootLeftNodeRightNode) &&
                            (EqualToZero(rootRightNode) ||
                             ↪  GreaterThan(GetSize(rootLeftNodeRightNode), rootRightNodeSize)))
                        {
                            LeftRotate(ref GetLeftReference(root));
                            RightRotate(ref root);
                        }
                        else
                        {
                            return;
                        }
                    }
                    LeftMaintain(ref GetLeftReference(root));
                    RightMaintain(ref GetRightReference(root));
                    LeftMaintain(ref root);
                    RightMaintain(ref root);
                }
            }
        }

        private void RightMaintain(ref TElement root)
        {
            if (!EqualToZero(root))
            {
                var rootRightNode = GetRight(root);
                if (!EqualToZero(rootRightNode))
                {
                    var rootLeftNode = GetLeft(root);
                    var rootLeftNodeSize = GetSize(rootLeftNode);
                    var rootRightNodeRightNode = GetRight(rootRightNode);
                    if (!EqualToZero(rootRightNodeRightNode) &&
                        (EqualToZero(rootLeftNode) ||
                         ↪  GreaterThan(GetSize(rootRightNodeRightNode), rootLeftNodeSize)))
                    {
                        LeftRotate(ref root);
                    }
                    else
                    {
                        var rootRightNodeLeftNode = GetLeft(rootRightNode);
                        if (!EqualToZero(rootRightNodeLeftNode) &&
                            (EqualToZero(rootLeftNode) ||
                             ↪  GreaterThan(GetSize(rootRightNodeLeftNode), rootLeftNodeSize)))
                        {
                            RightRotate(ref GetRightReference(root));
                            LeftRotate(ref root);
                        }
                        else
                        {
                            return;
                        }
                    }
                    LeftMaintain(ref GetLeftReference(root));
                    RightMaintain(ref GetRightReference(root));
                    LeftMaintain(ref root);
                    RightMaintain(ref root);
                }
            }
        }
    }
}
```

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

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

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

namespace Platform.Collections.Methods.Trees
{
    /// <summary>
    /// Combination of Size, Height (AVL), and threads.
    /// </summary>
```

```csharp
    /// <remarks>
    /// Based on: <a href="https://github.com/programmatom/TreeLib/blob/master/TreeLib/TreeLib/G↲
    ↪    enerated/AVLTreeList.cs">TreeLib.AVLTreeList</a>.
    /// Which itself based on: <a
    ↪    href="https://github.com/GNOME/glib/blob/master/glib/gtree.c">GNOME/glib/gtree</a>.
    /// </remarks>
    public abstract class SizedAndThreadedAVLBalancedTreeMethods<TElement> :
    ↪    SizedBinaryTreeMethodsBase<TElement>
    {
        private static readonly int _maxPath = 11 * NumericType<TElement>.BytesSize + 4;

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

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

        public override bool Contains(TElement node, TElement root)
        {
            while (!EqualToZero(root))
            {
                if (FirstIsToTheLeftOfSecond(node, root)) // node.Key < root.Key
                {
                    root = GetLeftOrDefault(root);
                }
                else if (FirstIsToTheRightOfSecond(node, root)) // node.Key > root.Key
                {
                    root = GetRightOrDefault(root);
                }
                else // node.Key == root.Key
                {
                    return true;
                }
            }
            return false;
        }

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

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

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

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
```

```csharp
        protected override TElement GetRightOrDefault(TElement node) => GetRightIsChild(node) ?
        ↪  GetRight(node) : default;

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

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

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

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

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

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

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

```csharp
                        }
                    }
                    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);
```

```
241                        }
242                        node = LeftRotateWithBalance(node);
243                    }
244                    return node;
245                }
246            }
247
248        protected TElement LeftRotateWithBalance(TElement node)
249        {
250            unchecked
251            {
252                var right = GetRight(node);
253                if (GetLeftIsChild(right))
254                {
255                    SetRight(node, GetLeft(right));
256                }
257                else
258                {
259                    SetRightIsChild(node, false);
260                    SetLeftIsChild(right, true);
261                }
262                SetLeft(right, node);
263                // Fix size
264                SetSize(right, GetSize(node));
265                FixSize(node);
266                // Fix balance
267                var rootBalance = GetBalance(node);
268                var rightBalance = GetBalance(right);
269                if (rightBalance <= 0)
270                {
271                    if (rootBalance >= 1)
272                    {
273                        SetBalance(right, (sbyte)(rightBalance - 1));
274                    }
275                    else
276                    {
277                        SetBalance(right, (sbyte)(rootBalance + rightBalance - 2));
278                    }
279                    SetBalance(node, (sbyte)(rootBalance - 1));
280                }
281                else
282                {
283                    if (rootBalance <= rightBalance)
284                    {
285                        SetBalance(right, (sbyte)(rootBalance - 2));
286                    }
287                    else
288                    {
289                        SetBalance(right, (sbyte)(rightBalance - 1));
290                    }
291                    SetBalance(node, (sbyte)(rootBalance - rightBalance - 1));
292                }
293                return right;
294            }
295        }
296
297        protected TElement RightRotateWithBalance(TElement node)
298        {
299            unchecked
300            {
301                var left = GetLeft(node);
302                if (GetRightIsChild(left))
303                {
304                    SetLeft(node, GetRight(left));
305                }
306                else
307                {
308                    SetLeftIsChild(node, false);
309                    SetRightIsChild(left, true);
310                }
311                SetRight(left, node);
312                // Fix size
313                SetSize(left, GetSize(node));
314                FixSize(node);
315                // Fix balance
316                var rootBalance = GetBalance(node);
317                var leftBalance = GetBalance(left);
318                if (leftBalance <= 0)
```

```csharp
                    {
                        if (leftBalance > rootBalance)
                        {
                            SetBalance(left, (sbyte)(leftBalance + 1));
                        }
                        else
                        {
                            SetBalance(left, (sbyte)(rootBalance + 2));
                        }
                        SetBalance(node, (sbyte)(rootBalance - leftBalance + 1));
                    }
                    else
                    {
                        if (rootBalance <= -1)
                        {
                            SetBalance(left, (sbyte)(leftBalance + 1));
                        }
                        else
                        {
                            SetBalance(left, (sbyte)(rootBalance + leftBalance + 2));
                        }
                        SetBalance(node, (sbyte)(rootBalance + 1));
                    }
                    return left;
                }
            }

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

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

        protected override void DetachCore(ref TElement root, TElement node)
        {
            unchecked
            {
#if USEARRAYPOOL
                var path = ArrayPool.Allocate<TElement>(MaxPath);
                var pathPosition = 0;
                path[pathPosition++] = default;
#else
                var path = new TElement[_maxPath];
                var pathPosition = 1;
#endif
                var currentNode = root;
                while (true)
                {
                    if (FirstIsToTheLeftOfSecond(node, currentNode))
                    {
                        if (!GetLeftIsChild(currentNode))
                        {
                            throw new InvalidOperationException("Cannot find a node.");
                        }
                        DecrementSize(currentNode);
                        path[pathPosition++] = currentNode;
                        currentNode = GetLeft(currentNode);
                    }
                    else if (FirstIsToTheRightOfSecond(node, currentNode))
                    {
                        if (!GetRightIsChild(currentNode))
                        {
                            throw new InvalidOperationException("Cannot find a node.");
```

```csharp
                    }
                    DecrementSize(currentNode);
                    path[pathPosition++] = currentNode;
                    currentNode = GetRight(currentNode);
                }
                else
                {
                    break;
                }
            }
            var parent = path[--pathPosition];
            var balanceNode = parent;
            var isLeftNode = !AreEqual(parent, default) && AreEqual(currentNode,
            ↪  GetLeft(parent));
            if (!GetLeftIsChild(currentNode))
            {
                if (!GetRightIsChild(currentNode)) // node has no children
                {
                    if (AreEqual(parent, default))
                    {
                        root = Zero;
                    }
                    else if (isLeftNode)
                    {
                        SetLeftIsChild(parent, false);
                        SetLeft(parent, GetLeft(currentNode));
                        IncrementBalance(parent);
                    }
                    else
                    {
                        SetRightIsChild(parent, false);
                        SetRight(parent, GetRight(currentNode));
                        DecrementBalance(parent);
                    }
                }
                else // node has a right child
                {
                    var successor = GetNext(currentNode);
                    SetLeft(successor, GetLeft(currentNode));
                    var right = GetRight(currentNode);
                    if (AreEqual(parent, default))
                    {
                        root = right;
                    }
                    else if (isLeftNode)
                    {
                        SetLeft(parent, right);
                        IncrementBalance(parent);
                    }
                    else
                    {
                        SetRight(parent, right);
                        DecrementBalance(parent);
                    }
                }
            }
            else // node has a left child
            {
                if (!GetRightIsChild(currentNode))
                {
                    var predecessor = GetPrevious(currentNode);
                    SetRight(predecessor, GetRight(currentNode));
                    var leftValue = GetLeft(currentNode);
                    if (AreEqual(parent, default))
                    {
                        root = leftValue;
                    }
                    else if (isLeftNode)
                    {
                        SetLeft(parent, leftValue);
                        IncrementBalance(parent);
                    }
                    else
                    {
                        SetRight(parent, leftValue);
                        DecrementBalance(parent);
                    }
                }
            }
```

```
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);
524                             if (AreEqual(parent, default))
525                             {
526                                 root = successor;
527                             }
528                             else if (isLeftNode)
529                             {
530                                 SetLeft(parent, successor);
531                             }
532                             else
533                             {
534                                 SetRight(parent, successor);
535                             }
536                         }
537                     }
538                     // restore balance
539                     if (!AreEqual(balanceNode, default))
540                     {
541                         while (true)
542                         {
543                             var balanceParent = path[--pathPosition];
544                             isLeftNode = !AreEqual(balanceParent, default) && AreEqual(balanceNode,
     ↪  GetLeft(balanceParent));
545                             var currentNodeBalance = GetBalance(balanceNode);
546                             if (currentNodeBalance < -1 || currentNodeBalance > 1)
547                             {
548                                 balanceNode = Balance(balanceNode);
549                                 if (AreEqual(balanceParent, default))
550                                 {
551                                     root = balanceNode;
```

```
552                                    }
553                                    else if (isLeftNode)
554                                    {
555                                        SetLeft(balanceParent, balanceNode);
556                                    }
557                                    else
558                                    {
559                                        SetRight(balanceParent, balanceNode);
560                                    }
561                                }
562                                currentNodeBalance = GetBalance(balanceNode);
563                                if (currentNodeBalance != 0 || AreEqual(balanceParent, default))
564                                {
565                                    break;
566                                }
567                                if (isLeftNode)
568                                {
569                                    IncrementBalance(balanceParent);
570                                }
571                                else
572                                {
573                                    DecrementBalance(balanceParent);
574                                }
575                                balanceNode = balanceParent;
576                            }
577                        }
578                        ClearNode(node);
579  #if USEARRAYPOOL
580                        ArrayPool.Free(path);
581  #endif
582                    }
583            }
584
585            [MethodImpl(MethodImplOptions.AggressiveInlining)]
586            protected override void ClearNode(TElement node)
587            {
588                SetLeft(node, Zero);
589                SetRight(node, Zero);
590                SetSize(node, Zero);
591                SetLeftIsChild(node, false);
592                SetRightIsChild(node, false);
593                SetBalance(node, 0);
594            }
595        }
596  }
```

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

```
1   //#define ENABLE_TREE_AUTO_DEBUG_AND_VALIDATION
2
3   using System;
4   using System.Diagnostics;
5   using System.Runtime.CompilerServices;
6   using System.Text;
7   using Platform.Numbers;
8
9   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
10
11  namespace Platform.Collections.Methods.Trees
12  {
13      public abstract class SizedBinaryTreeMethodsBase<TElement> :
        ↪   GenericCollectionMethodsBase<TElement>
14      {
15          [MethodImpl(MethodImplOptions.AggressiveInlining)]
16          protected abstract ref TElement GetLeftReference(TElement node);
17
18          [MethodImpl(MethodImplOptions.AggressiveInlining)]
19          protected abstract ref TElement GetRightReference(TElement node);
20
21          [MethodImpl(MethodImplOptions.AggressiveInlining)]
22          protected abstract TElement GetLeft(TElement node);
23
24          [MethodImpl(MethodImplOptions.AggressiveInlining)]
25          protected abstract TElement GetRight(TElement node);
26
27          [MethodImpl(MethodImplOptions.AggressiveInlining)]
28          protected abstract TElement GetSize(TElement node);
29
30          [MethodImpl(MethodImplOptions.AggressiveInlining)]
31          protected abstract void SetLeft(TElement node, TElement left);
32
```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```csharp
108
109            [MethodImpl(MethodImplOptions.AggressiveInlining)]
110            protected virtual TElement GetRightest(TElement current)
111            {
112                var currentRight = GetRight(current);
113                while (!EqualToZero(currentRight))
114                {
115                    current = currentRight;
116                    currentRight = GetRight(current);
117                }
118                return current;
119            }
120
121            [MethodImpl(MethodImplOptions.AggressiveInlining)]
122            protected virtual TElement GetLeftest(TElement current)
123            {
124                var currentLeft = GetLeft(current);
125                while (!EqualToZero(currentLeft))
126                {
127                    current = currentLeft;
128                    currentLeft = GetLeft(current);
129                }
130                return current;
131            }
132
133            [MethodImpl(MethodImplOptions.AggressiveInlining)]
134            protected virtual TElement GetNext(TElement node) => GetLeftest(GetRight(node));
135
136            [MethodImpl(MethodImplOptions.AggressiveInlining)]
137            protected virtual TElement GetPrevious(TElement node) => GetRightest(GetLeft(node));
138
139            [MethodImpl(MethodImplOptions.AggressiveInlining)]
140            public virtual bool Contains(TElement node, TElement root)
141            {
142                while (!EqualToZero(root))
143                {
144                    if (FirstIsToTheLeftOfSecond(node, root)) // node.Key < root.Key
145                    {
146                        root = GetLeft(root);
147                    }
148                    else if (FirstIsToTheRightOfSecond(node, root)) // node.Key > root.Key
149                    {
150                        root = GetRight(root);
151                    }
152                    else // node.Key == root.Key
153                    {
154                        return true;
155                    }
156                }
157                return false;
158            }
159
160            [MethodImpl(MethodImplOptions.AggressiveInlining)]
161            protected virtual void ClearNode(TElement node)
162            {
163                SetLeft(node, Zero);
164                SetRight(node, Zero);
165                SetSize(node, Zero);
166            }
167
168            [MethodImpl(MethodImplOptions.AggressiveInlining)]
169            public void Attach(ref TElement root, TElement node)
170            {
171 #if ENABLE_TREE_AUTO_DEBUG_AND_VALIDATION
172                ValidateSizes(root);
173                Debug.WriteLine("--BeforeAttach--");
174                Debug.WriteLine(PrintNodes(root));
175                Debug.WriteLine("----------------");
176                var sizeBefore = GetSize(root);
177 #endif
178                if (EqualToZero(root))
179                {
180                    SetSize(node, One);
181                    root = node;
182                    return;
183                }
184                AttachCore(ref root, node);
185 #if ENABLE_TREE_AUTO_DEBUG_AND_VALIDATION
186                Debug.WriteLine("--AfterAttach--");
```

```csharp
                    Debug.WriteLine(PrintNodes(root));
                    Debug.WriteLine("----------------");
                    ValidateSizes(root);
                    var sizeAfter = GetSize(root);
                    if (!AreEqual(Arithmetic.Increment(sizeBefore), sizeAfter))
                    {
                        throw new InvalidOperationException("Tree was broken after attach.");
                    }
#endif
        }

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

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

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

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

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

        public void ValidateSize(TElement node)
        {
            var size = GetSize(node);
            var leftSize = GetLeftSize(node);
```

```
263                    var rightSize = GetRightSize(node);
264                    var expectedSize = Arithmetic.Increment(Arithmetic.Add(leftSize, rightSize));
265                    if (!AreEqual(size, expectedSize))
266                    {
267                        throw new InvalidOperationException($"Size of {node} is not valid. Expected
       ↪     size: {expectedSize}, actual size: {size}.");
268                    }
269            }
270
271            public string PrintNodes(TElement node)
272            {
273                    var sb = new StringBuilder();
274                    PrintNodes(node, sb);
275                    return sb.ToString();
276            }
277
278            [MethodImpl(MethodImplOptions.AggressiveInlining)]
279            public void PrintNodes(TElement node, StringBuilder sb) => PrintNodes(node, sb, 0);
280
281            public void PrintNodes(TElement node, StringBuilder sb, int level)
282            {
283                    if (AreEqual(node, default))
284                    {
285                        return;
286                    }
287                    PrintNodes(GetLeft(node), sb, level + 1);
288                    PrintNode(node, sb, level);
289                    sb.AppendLine();
290                    PrintNodes(GetRight(node), sb, level + 1);
291            }
292
293            public string PrintNode(TElement node)
294            {
295                    var sb = new StringBuilder();
296                    PrintNode(node, sb);
297                    return sb.ToString();
298            }
299
300            [MethodImpl(MethodImplOptions.AggressiveInlining)]
301            protected void PrintNode(TElement node, StringBuilder sb) => PrintNode(node, sb, 0);
302
303            protected virtual void PrintNode(TElement node, StringBuilder sb, int level)
304            {
305                    sb.Append('\t', level);
306                    sb.Append(node);
307                    PrintNodeValue(node, sb);
308                    sb.Append(' ');
309                    sb.Append('s');
310                    sb.Append(GetSize(node));
311            }
312
313            protected abstract void PrintNodeValue(TElement node, StringBuilder sb);
314        }
315    }
```

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

```
1    using System;
2    using System.Collections.Generic;
3    using System.Text;
4    using Platform.Numbers;
5    using Platform.Collections.Methods.Trees;
6    using Platform.Converters;
7
8    namespace Platform.Collections.Methods.Tests
9    {
10        public class RecursionlessSizeBalancedTree<TElement> :
       ↪    RecursionlessSizeBalancedTreeMethods<TElement>
11        {
12            private struct TreeElement
13            {
14                public TElement Size;
15                public TElement Left;
16                public TElement Right;
17            }
18
19            private readonly TreeElement[] _elements;
20            private TElement _allocated;
21
22            public TElement Root;
23
```

```
24        public TElement Count => GetSizeOrZero(Root);

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

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        }

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        }

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

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

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

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

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

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

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

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

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

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

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

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

83        private ref TreeElement GetElement(TElement node) => ref
   ↪     _elements[UncheckedConverter<TElement, long>.Default.Convert(node)];
84    }
85 }
```

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

```
1 using System;
2 using System.Collections.Generic;
3 using System.Text;
4 using Platform.Numbers;
```

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

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

        private readonly TreeElement[] _elements;
        private TElement _allocated;

        public TElement Root;

        public TElement Count => GetSizeOrZero(Root);

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```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 SizedAndThreadedAVLBalancedTree<TElement> :
    ↪   SizedAndThreadedAVLBalancedTreeMethods<TElement>
    {
        private struct TreeElement
        {
            public TElement Size;
            public TElement Left;
            public TElement Right;
            public sbyte Balance;
            public bool LeftIsChild;
            public bool RightIsChild;
        }

        private readonly TreeElement[] _elements;
        private TElement _allocated;

        public TElement Root;

        public TElement Count => GetSizeOrZero(Root);

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```csharp
using Xunit;

namespace Platform.Collections.Methods.Tests
{
    public static class TreesTests
    {
        private const int _n = 500;

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

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

```csharp
        }

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

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

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

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

Index