LinksPlatform's Platform.Collections.Methods Class Library

```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>
    /// <typeparam name="TElement"><para>Source type of conversion.</para><para>Исходный тип
    ///    конверсии.</para></typeparam>
    public abstract class GenericCollectionMethodsBase<TElement>
    {
        /// <summary>
        /// <para>Returns a null constant of type <see cref = "TElement" />.</para>
        /// <para>Возвращает нулевую константу типа <see cref="TElement" />.</para>
        /// </summary>
        /// <returns><para>A null constant of type <see cref = "TElement"
        ///    />.</para><para>Нулевую константу типа <see cref="TElement" />.</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;
```

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

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

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

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

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

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

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

        /// <summary>
        /// <para>Presents the Range in readable format.</para>
        /// <para>Представляет диапазон в удобном для чтения формате.</para>
        /// </summary>
        /// <returns><para>String representation of the Range.</para><para>Строковое
        ↪  представление диапазона.</para></returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
```

```csharp
126            protected virtual TElement Add(TElement first, TElement second) =>
      ↪    Arithmetic<TElement>.Add(first, second);
127
128            /// <summary>
129            /// <para>Presents the Range in readable format.</para>
130            /// <para>Представляет диапазон в удобном для чтения формате.</para>
131            /// </summary>
132            /// <returns><para>String representation of the Range.</para><para>Строковое
      ↪    представление диапазона.</para></returns>
133            [MethodImpl(MethodImplOptions.AggressiveInlining)]
134            protected virtual TElement Subtract(TElement first, TElement second) =>
      ↪    Arithmetic<TElement>.Subtract(first, second);
135
136            /// <summary>
137            /// <para>Returns minimum value of the range.</para>
138            /// <para>Возвращает минимальное значение диапазона.</para>
139            /// </summary>
140            protected readonly TElement Zero;
141
142            /// <summary>
143            /// <para>Returns minimum value of the range.</para>
144            /// <para>Возвращает минимальное значение диапазона.</para>
145            /// </summary>
146            protected readonly TElement One;
147
148             /// <summary>
149            /// <para>Returns minimum value of the range.</para>
150            /// <para>Возвращает минимальное значение диапазона.</para>
151            /// </summary>
152            protected readonly TElement Two;
153
154            /// <summary>
155            /// <para>Returns minimum value of the range.</para>
156            /// <para>Возвращает минимальное значение диапазона.</para>
157            /// </summary>
158            protected readonly EqualityComparer<TElement> EqualityComparer;
159
160            /// <summary>
161            /// <para>Returns minimum value of the range.</para>
162            /// <para>Возвращает минимальное значение диапазона.</para>
163            /// </summary>
164            protected readonly Comparer<TElement> Comparer;
165
166            /// <summary>
167            /// <para>Presents the Range in readable format.</para>
168            /// <para>Представляет диапазон в удобном для чтения формате.</para>
169            /// </summary>
170            /// <returns><para>String representation of the Range.</para><para>Строковое
      ↪    представление диапазона.</para></returns>
171            protected GenericCollectionMethodsBase()
172            {
173                EqualityComparer = EqualityComparer<TElement>.Default;
174                Comparer = Comparer<TElement>.Default;
175                Zero = GetZero(); //-V3068
176                One = Increment(Zero); //-V3068
177                Two = Increment(One); //-V3068
178            }
179        }
180    }
```

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);
```

```csharp
                IncrementSize();
        }

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

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

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

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

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

```csharp
using System.Runtime.CompilerServices;

```

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

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

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

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

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

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

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

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

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

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

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

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

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

        public void AttachAsFirst(TElement element)
        {
            var first = GetFirst();
            if (EqualToZero(first))
```

```
45          {
46              SetFirst(element);
47              SetLast(element);
48              SetPrevious(element, Zero);
49              SetNext(element, Zero);
50              IncrementSize();
51          }
52          else
53          {
54              AttachBefore(first, element);
55          }
56      }
57
58      public void AttachAsLast(TElement element)
59      {
60          var last = GetLast();
61          if (EqualToZero(last))
62          {
63              AttachAsFirst(element);
64          }
65          else
66          {
67              AttachAfter(last, element);
68          }
69      }
70
71      public void Detach(TElement element)
72      {
73          var elementPrevious = GetPrevious(element);
74          var elementNext = GetNext(element);
75          if (EqualToZero(elementPrevious))
76          {
77              SetFirst(elementNext);
78          }
79          else
80          {
81              SetNext(elementPrevious, elementNext);
82          }
83          if (EqualToZero(elementNext))
84          {
85              SetLast(elementPrevious);
86          }
87          else
88          {
89              SetPrevious(elementNext, elementPrevious);
90          }
91          SetPrevious(element, Zero);
92          SetNext(element, Zero);
93          DecrementSize();
94      }
95  }
96 }
```

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

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

```
24   }

1.6   ./csharp/Platform.Collections.Methods/Lists/RelativeCircularDoublyLinkedListMethods.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 RelativeCircularDoublyLinkedListMethods<TElement> :
     ↪    RelativeDoublyLinkedListMethodsBase<TElement>
6        {
7            public void AttachBefore(TElement headElement, TElement baseElement, TElement newElement)
8            {
9                var baseElementPrevious = GetPrevious(baseElement);
10               SetPrevious(newElement, baseElementPrevious);
11               SetNext(newElement, baseElement);
12               if (AreEqual(baseElement, GetFirst(headElement)))
13               {
14                   SetFirst(headElement, newElement);
15               }
16               SetNext(baseElementPrevious, newElement);
17               SetPrevious(baseElement, newElement);
18               IncrementSize(headElement);
19           }
20
21           public void AttachAfter(TElement headElement, TElement baseElement, TElement newElement)
22           {
23               var baseElementNext = GetNext(baseElement);
24               SetPrevious(newElement, baseElement);
25               SetNext(newElement, baseElementNext);
26               if (AreEqual(baseElement, GetLast(headElement)))
27               {
28                   SetLast(headElement, newElement);
29               }
30               SetPrevious(baseElementNext, newElement);
31               SetNext(baseElement, newElement);
32               IncrementSize(headElement);
33           }
34
35           public void AttachAsFirst(TElement headElement, TElement element)
36           {
37               var first = GetFirst(headElement);
38               if (EqualToZero(first))
39               {
40                   SetFirst(headElement, element);
41                   SetLast(headElement, element);
42                   SetPrevious(element, element);
43                   SetNext(element, element);
44                   IncrementSize(headElement);
45               }
46               else
47               {
48                   AttachBefore(headElement, first, element);
49               }
50           }
51
52           public void AttachAsLast(TElement headElement, TElement element)
53           {
54               var last = GetLast(headElement);
55               if (EqualToZero(last))
56               {
57                   AttachAsFirst(headElement, element);
58               }
59               else
60               {
61                   AttachAfter(headElement, last, element);
62               }
63           }
64
65           public void Detach(TElement headElement, TElement element)
66           {
67               var elementPrevious = GetPrevious(element);
68               var elementNext = GetNext(element);
69               if (AreEqual(elementNext, element))
70               {
71                   SetFirst(headElement, Zero);
72                   SetLast(headElement, Zero);
73               }
74               else
75               {
```

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

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

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

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

```
1  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
2
3  namespace Platform.Collections.Methods.Lists
4  {
5      public abstract class RelativeOpenDoublyLinkedListMethods<TElement> :
       ↪  RelativeDoublyLinkedListMethodsBase<TElement>
6      {
7          public void AttachBefore(TElement headElement, TElement baseElement, TElement newElement)
8          {
9              var baseElementPrevious = GetPrevious(baseElement);
10             SetPrevious(newElement, baseElementPrevious);
11             SetNext(newElement, baseElement);
12             if (EqualToZero(baseElementPrevious))
13             {
14                 SetFirst(headElement, newElement);
15             }
16             else
17             {
18                 SetNext(baseElementPrevious, newElement);
19             }
20             SetPrevious(baseElement, newElement);
```

```csharp
                    IncrementSize(headElement);
            }

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

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

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

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

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

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

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

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

```
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          {
```

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

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

```
110                        RightRotate(ref root);
111                    }
112                    else
113                    {
114                        var rootLeftNodeRightNode = GetRight(rootLeftNode);
115                        if (!EqualToZero(rootLeftNodeRightNode) &&
116                            (EqualToZero(rootRightNode) ||
                              ↪   GreaterThan(GetSize(rootLeftNodeRightNode), rootRightNodeSize)))
117                        {
118                            LeftRotate(ref GetLeftReference(root));
119                            RightRotate(ref root);
120                        }
121                        else
122                        {
123                            return;
124                        }
125                    }
126                    LeftMaintain(ref GetLeftReference(root));
127                    RightMaintain(ref GetRightReference(root));
128                    LeftMaintain(ref root);
129                    RightMaintain(ref root);
130                }
131            }
132        }
133
134        private void RightMaintain(ref TElement root)
135        {
136            if (!EqualToZero(root))
137            {
138                var rootRightNode = GetRight(root);
139                if (!EqualToZero(rootRightNode))
140                {
141                    var rootLeftNode = GetLeft(root);
142                    var rootLeftNodeSize = GetSize(rootLeftNode);
143                    var rootRightNodeRightNode = GetRight(rootRightNode);
144                    if (!EqualToZero(rootRightNodeRightNode) &&
145                        (EqualToZero(rootLeftNode) ||
                          ↪   GreaterThan(GetSize(rootRightNodeRightNode), rootLeftNodeSize)))
146                    {
147                        LeftRotate(ref root);
148                    }
149                    else
150                    {
151                        var rootRightNodeLeftNode = GetLeft(rootRightNode);
152                        if (!EqualToZero(rootRightNodeLeftNode) &&
153                            (EqualToZero(rootLeftNode) ||
                              ↪   GreaterThan(GetSize(rootRightNodeLeftNode), rootLeftNodeSize)))
154                        {
155                            RightRotate(ref GetRightReference(root));
156                            LeftRotate(ref root);
157                        }
158                        else
159                        {
160                            return;
161                        }
162                    }
163                    LeftMaintain(ref GetLeftReference(root));
164                    RightMaintain(ref GetRightReference(root));
165                    LeftMaintain(ref root);
166                    RightMaintain(ref root);
167                }
168            }
169        }
170    }
171 }
```

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

```
1  using System;
2  using System.Runtime.CompilerServices;
3  using System.Text;
4  #if USEARRAYPOOL
5  using Platform.Collections;
6  #endif
7  using Platform.Reflection;
8
9  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
10
11 namespace Platform.Collections.Methods.Trees
12 {
```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```csharp
                    SetRight(node, RightRotateWithBalance(right));
                    FixSize(node);
                }
                node = LeftRotateWithBalance(node);
            }
            return node;
        }
    }

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

    protected TElement RightRotateWithBalance(TElement node)
    {
        unchecked
        {
            var left = GetLeft(node);
            if (GetRightIsChild(left))
            {
                SetLeft(node, GetRight(left));
            }
            else
            {
                SetLeftIsChild(node, false);
                SetRightIsChild(left, true);
            }
            SetRight(left, node);
            // Fix size
            SetSize(left, GetSize(node));
            FixSize(node);
            // Fix balance
            var rootBalance = GetBalance(node);
```

```csharp
                        var leftBalance = GetBalance(left);
                        if (leftBalance <= 0)
                        {
                            if (leftBalance > rootBalance)
                            {
                                SetBalance(left, (sbyte)(leftBalance + 1));
                            }
                            else
                            {
                                SetBalance(left, (sbyte)(rootBalance + 2));
                            }
                            SetBalance(node, (sbyte)(rootBalance - leftBalance + 1));
                        }
                        else
                        {
                            if (rootBalance <= -1)
                            {
                                SetBalance(left, (sbyte)(leftBalance + 1));
                            }
                            else
                            {
                                SetBalance(left, (sbyte)(rootBalance + leftBalance + 2));
                            }
                            SetBalance(node, (sbyte)(rootBalance + 1));
                        }
                        return left;
                    }
        }

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

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

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

```
396                    {
397                        throw new InvalidOperationException("Cannot find a node.");
398                    }
399                    DecrementSize(currentNode);
400                    path[pathPosition++] = currentNode;
401                    currentNode = GetRight(currentNode);
402                }
403                else
404                {
405                    break;
406                }
407            }
408            var parent = path[--pathPosition];
409            var balanceNode = parent;
410            var isLeftNode = !AreEqual(parent, default) && AreEqual(currentNode,
    ↪  GetLeft(parent));
411            if (!GetLeftIsChild(currentNode))
412            {
413                if (!GetRightIsChild(currentNode)) // node has no children
414                {
415                    if (AreEqual(parent, default))
416                    {
417                        root = Zero;
418                    }
419                    else if (isLeftNode)
420                    {
421                        SetLeftIsChild(parent, false);
422                        SetLeft(parent, GetLeft(currentNode));
423                        IncrementBalance(parent);
424                    }
425                    else
426                    {
427                        SetRightIsChild(parent, false);
428                        SetRight(parent, GetRight(currentNode));
429                        DecrementBalance(parent);
430                    }
431                }
432                else // node has a right child
433                {
434                    var successor = GetNext(currentNode);
435                    SetLeft(successor, GetLeft(currentNode));
436                    var right = GetRight(currentNode);
437                    if (AreEqual(parent, default))
438                    {
439                        root = right;
440                    }
441                    else if (isLeftNode)
442                    {
443                        SetLeft(parent, right);
444                        IncrementBalance(parent);
445                    }
446                    else
447                    {
448                        SetRight(parent, right);
449                        DecrementBalance(parent);
450                    }
451                }
452            }
453            else // node has a left child
454            {
455                if (!GetRightIsChild(currentNode))
456                {
457                    var predecessor = GetPrevious(currentNode);
458                    SetRight(predecessor, GetRight(currentNode));
459                    var leftValue = GetLeft(currentNode);
460                    if (AreEqual(parent, default))
461                    {
462                        root = leftValue;
463                    }
464                    else if (isLeftNode)
465                    {
466                        SetLeft(parent, leftValue);
467                        IncrementBalance(parent);
468                    }
469                    else
470                    {
471                        SetRight(parent, leftValue);
472                        DecrementBalance(parent);
```

```csharp
                    }
                }
                else // node has a both children (left and right)
                {
                    var predecessor = GetLeft(currentNode);
                    var successor = GetRight(currentNode);
                    var successorParent = currentNode;
                    int previousPathPosition = ++pathPosition;
                    // find the immediately next node (and its parent)
                    while (GetLeftIsChild(successor))
                    {
                        path[++pathPosition] = successorParent = successor;
                        successor = GetLeft(successor);
                        if (!AreEqual(successorParent, currentNode))
                        {
                            DecrementSize(successorParent);
                        }
                    }
                    path[previousPathPosition] = successor;
                    balanceNode = path[pathPosition];
                    // remove 'successor' from the tree
                    if (!AreEqual(successorParent, currentNode))
                    {
                        if (!GetRightIsChild(successor))
                        {
                            SetLeftIsChild(successorParent, false);
                        }
                        else
                        {
                            SetLeft(successorParent, GetRight(successor));
                        }
                        IncrementBalance(successorParent);
                        SetRightIsChild(successor, true);
                        SetRight(successor, GetRight(currentNode));
                    }
                    else
                    {
                        DecrementBalance(currentNode);
                    }
                    // set the predecessor's successor link to point to the right place
                    while (GetRightIsChild(predecessor))
                    {
                        predecessor = GetRight(predecessor);
                    }
                    SetRight(predecessor, successor);
                    // prepare 'successor' to replace 'node'
                    var left = GetLeft(currentNode);
                    SetLeftIsChild(successor, true);
                    SetLeft(successor, left);
                    SetBalance(successor, GetBalance(currentNode));
                    FixSize(successor);
                    if (AreEqual(parent, default))
                    {
                        root = successor;
                    }
                    else if (isLeftNode)
                    {
                        SetLeft(parent, successor);
                    }
                    else
                    {
                        SetRight(parent, successor);
                    }
                }
            }
            // restore balance
            if (!AreEqual(balanceNode, default))
            {
                while (true)
                {
                    var balanceParent = path[--pathPosition];
                    isLeftNode = !AreEqual(balanceParent, default) && AreEqual(balanceNode,
                        GetLeft(balanceParent));
                    var currentNodeBalance = GetBalance(balanceNode);
                    if (currentNodeBalance < -1 || currentNodeBalance > 1)
                    {
                        balanceNode = Balance(balanceNode);
                        if (AreEqual(balanceParent, default))
```

```
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)]
```

```csharp
        protected abstract void SetLeft(TElement node, TElement left);

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```csharp
                return left;
            }

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

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

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

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

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

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

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public void Attach(ref TElement root, TElement node)
            {
#if ENABLE_TREE_AUTO_DEBUG_AND_VALIDATION
                ValidateSizes(root);
                Debug.WriteLine("--BeforeAttach--");
                Debug.WriteLine(PrintNodes(root));
                Debug.WriteLine("----------------");
                var sizeBefore = GetSize(root);
#endif
                if (EqualToZero(root))
                {
                    SetSize(node, One);
                    root = node;
                    return;
                }
                AttachCore(ref root, node);
```

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

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

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

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

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

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

        public void ValidateSize(TElement node)
        {
```

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

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

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

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

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

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

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

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

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

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

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

        private readonly TreeElement[] _elements;
        private TElement _allocated;
```

```
22          public TElement Root;
23
24          public TElement Count => GetSizeOrZero(Root);
25
26          public RecursionlessSizeBalancedTree(int capacity) => (_elements, _allocated) = (new
   ↪   TreeElement[capacity], One);
27
28          public TElement Allocate()
29          {
30              var newNode = _allocated;
31              if (IsEmpty(newNode))
32              {
33                  _allocated = Arithmetic.Increment(_allocated);
34                  return newNode;
35              }
36              else
37              {
38                  throw new InvalidOperationException("Allocated tree element is not empty.");
39              }
40          }
41
42          public void Free(TElement node)
43          {
44              while (!EqualityComparer.Equals(_allocated, One) && IsEmpty(node))
45              {
46                  var lastNode = Arithmetic.Decrement(_allocated);
47                  if (EqualityComparer.Equals(lastNode, node))
48                  {
49                      _allocated = lastNode;
50                      node = Arithmetic.Decrement(node);
51                  }
52                  else
53                  {
54                      return;
55                  }
56              }
57          }
58
59          public bool IsEmpty(TElement node) =>
   ↪   EqualityComparer<TreeElement>.Default.Equals(GetElement(node), default);
60
61          protected override bool FirstIsToTheLeftOfSecond(TElement first, TElement second) =>
   ↪   Comparer.Compare(first, second) < 0;
62
63          protected override bool FirstIsToTheRightOfSecond(TElement first, TElement second) =>
   ↪   Comparer.Compare(first, second) > 0;
64
65          protected override ref TElement GetLeftReference(TElement node) => ref
   ↪   GetElement(node).Left;
66
67          protected override TElement GetLeft(TElement node) => GetElement(node).Left;
68
69          protected override ref TElement GetRightReference(TElement node) => ref
   ↪   GetElement(node).Right;
70
71          protected override TElement GetRight(TElement node) => GetElement(node).Right;
72
73          protected override TElement GetSize(TElement node) => GetElement(node).Size;
74
75          protected override void PrintNodeValue(TElement node, StringBuilder sb) =>
   ↪   sb.Append(node);
76
77          protected override void SetLeft(TElement node, TElement left) => GetElement(node).Left =
   ↪   left;
78
79          protected override void SetRight(TElement node, TElement right) =>
   ↪   GetElement(node).Right = right;
80
81          protected override void SetSize(TElement node, TElement size) => GetElement(node).Size =
   ↪   size;
82
83          private ref TreeElement GetElement(TElement node) => ref
   ↪   _elements[UncheckedConverter<TElement, long>.Default.Convert(node)];
84      }
85  }
```

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

```
1  using System;
2  using System.Collections.Generic;
```

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

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

        private readonly TreeElement[] _elements;
        private TElement _allocated;

        public TElement Root;

        public TElement Count => GetSizeOrZero(Root);

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

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

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

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

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

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

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

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

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

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

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

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

```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);
```

```
63
64          protected override bool FirstIsToTheLeftOfSecond(TElement first, TElement second) =>
             ↪   Comparer.Compare(first, second) < 0;
65
66          protected override bool FirstIsToTheRightOfSecond(TElement first, TElement second) =>
             ↪   Comparer.Compare(first, second) > 0;
67
68          protected override sbyte GetBalance(TElement node) => GetElement(node).Balance;
69
70          protected override bool GetLeftIsChild(TElement node) => GetElement(node).LeftIsChild;
71
72          protected override ref TElement GetLeftReference(TElement node) => ref
             ↪   GetElement(node).Left;
73
74          protected override TElement GetLeft(TElement node) => GetElement(node).Left;
75
76          protected override bool GetRightIsChild(TElement node) => GetElement(node).RightIsChild;
77
78          protected override ref TElement GetRightReference(TElement node) => ref
             ↪   GetElement(node).Right;
79
80          protected override TElement GetRight(TElement node) => GetElement(node).Right;
81
82          protected override TElement GetSize(TElement node) => GetElement(node).Size;
83
84          protected override void PrintNodeValue(TElement node, StringBuilder sb) =>
             ↪   sb.Append(node);
85
86          protected override void SetBalance(TElement node, sbyte value) =>
             ↪   GetElement(node).Balance = value;
87
88          protected override void SetLeft(TElement node, TElement left) => GetElement(node).Left =
             ↪   left;
89
90          protected override void SetLeftIsChild(TElement node, bool value) =>
             ↪   GetElement(node).LeftIsChild = value;
91
92          protected override void SetRight(TElement node, TElement right) =>
             ↪   GetElement(node).Right = right;
93
94          protected override void SetRightIsChild(TElement node, bool value) =>
             ↪   GetElement(node).RightIsChild = value;
95
96          protected override void SetSize(TElement node, TElement size) => GetElement(node).Size =
             ↪   size;
97
98          private ref TreeElement GetElement(TElement node) => ref
             ↪   _elements[UncheckedConverter<TElement, long>.Default.Convert(node)];
99      }
100 }
```

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

```
1   using System;
2   using System.Collections.Generic;
3   using Xunit;
4   using Platform.Collections.Methods.Trees;
5   using Platform.Converters;
6
7   namespace Platform.Collections.Methods.Tests
8   {
9       public static class TestExtensions
10      {
11          public static void TestMultipleCreationsAndDeletions<TElement>(this
             ↪   SizedBinaryTreeMethodsBase<TElement> tree, Func<TElement> allocate, Action<TElement>
             ↪   free, ref TElement root, Func<TElement> treeCount, int maximumOperationsPerCycle)
12          {
13              for (var N = 1; N < maximumOperationsPerCycle; N++)
14              {
15                  var currentCount = 0;
16                  for (var i = 0; i < N; i++)
17                  {
18                      var node = allocate();
19                      tree.Attach(ref root, node);
20                      currentCount++;
21                      Assert.Equal(currentCount, (int)UncheckedConverter<TElement,
                         ↪   int>.Default.Convert(treeCount()));
22                  }
23                  for (var i = 1; i <= N; i++)
24                  {
```

```
25                    TElement node = UncheckedConverter<int, TElement>.Default.Convert(i);
26                    if (tree.Contains(node, root))
27                    {
28                        tree.Detach(ref root, node);
29                        free(node);
30                        currentCount--;
31                        Assert.Equal(currentCount, (int)UncheckedConverter<TElement,
   ↪    int>.Default.Convert(treeCount()));
32                    }
33                }
34            }
35        }
36
37        public static void TestMultipleRandomCreationsAndDeletions<TElement>(this
   ↪    SizedBinaryTreeMethodsBase<TElement> tree, ref TElement root, Func<TElement>
   ↪    treeCount, int maximumOperationsPerCycle)
38        {
39            var random = new System.Random(0);
40            var added = new HashSet<TElement>();
41            var currentCount = 0;
42            for (var N = 1; N < maximumOperationsPerCycle; N++)
43            {
44                for (var i = 0; i < N; i++)
45                {
46                    var node = UncheckedConverter<int, TElement>.Default.Convert(random.Next(1,
   ↪    N));
47                    if (added.Add(node))
48                    {
49                        tree.Attach(ref root, node);
50                        currentCount++;
51                        Assert.Equal(currentCount, (int)UncheckedConverter<TElement,
   ↪    int>.Default.Convert(treeCount()));
52                    }
53                }
54                for (var i = 1; i <= N; i++)
55                {
56                    TElement node = UncheckedConverter<int,
   ↪    TElement>.Default.Convert(random.Next(1, N));
57                    if (tree.Contains(node, root))
58                    {
59                        tree.Detach(ref root, node);
60                        currentCount--;
61                        Assert.Equal(currentCount, (int)UncheckedConverter<TElement,
   ↪    int>.Default.Convert(treeCount()));
62                        added.Remove(node);
63                    }
64                }
65            }
66        }
67    }
68 }
```

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

```
1  using Xunit;
2
3  namespace Platform.Collections.Methods.Tests
4  {
5      public static class TreesTests
6      {
7          private const int _n = 500;
8
9          [Fact]
10         public static void RecursionlessSizeBalancedTreeMultipleAttachAndDetachTest()
11         {
12             var recursionlessSizeBalancedTree = new RecursionlessSizeBalancedTree<uint>(10000);
13             recursionlessSizeBalancedTree.TestMultipleCreationsAndDeletions(recursionlessSizeBal↵
   ↪    ancedTree.Allocate, recursionlessSizeBalancedTree.Free, ref
   ↪    recursionlessSizeBalancedTree.Root, () => recursionlessSizeBalancedTree.Count,
   ↪    _n);
14         }
15
16         [Fact]
17         public static void SizeBalancedTreeMultipleAttachAndDetachTest()
18         {
19             var sizeBalancedTree = new SizeBalancedTree<uint>(10000);
20             sizeBalancedTree.TestMultipleCreationsAndDeletions(sizeBalancedTree.Allocate,
   ↪    sizeBalancedTree.Free, ref sizeBalancedTree.Root, () => sizeBalancedTree.Count,
   ↪    _n);
```

```csharp
21          }
22
23          [Fact]
24          public static void SizedAndThreadedAVLBalancedTreeMultipleAttachAndDetachTest()
25          {
26              var avlTree = new SizedAndThreadedAVLBalancedTree<uint>(10000);
27              avlTree.TestMultipleCreationsAndDeletions(avlTree.Allocate, avlTree.Free, ref
                ↪   avlTree.Root, () => avlTree.Count, _n);
28          }
29
30          [Fact]
31          public static void RecursionlessSizeBalancedTreeMultipleRandomAttachAndDetachTest()
32          {
33              var recursionlessSizeBalancedTree = new RecursionlessSizeBalancedTree<uint>(10000);
34              recursionlessSizeBalancedTree.TestMultipleRandomCreationsAndDeletions(ref
                ↪   recursionlessSizeBalancedTree.Root, () => recursionlessSizeBalancedTree.Count,
                ↪   _n);
35          }
36
37          [Fact]
38          public static void SizeBalancedTreeMultipleRandomAttachAndDetachTest()
39          {
40              var sizeBalancedTree = new SizeBalancedTree<uint>(10000);
41              sizeBalancedTree.TestMultipleRandomCreationsAndDeletions(ref sizeBalancedTree.Root,
                ↪   () => sizeBalancedTree.Count, _n);
42          }
43
44          [Fact]
45          public static void SizedAndThreadedAVLBalancedTreeMultipleRandomAttachAndDetachTest()
46          {
47              var avlTree = new SizedAndThreadedAVLBalancedTree<uint>(10000);
48              avlTree.TestMultipleRandomCreationsAndDeletions(ref avlTree.Root, () =>
                ↪   avlTree.Count, _n);
49          }
50      }
51  }
```

Index