

# LinksPlatform's Platform.Collections.Methods Class Library

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

```
1 using System.Collections.Generic;
2 using System.Runtime.CompilerServices;
3 using Platform.Numbers;
4
5 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6
7 namespace Platform.Collections.Methods
8 {
9     /// <summary>
10     /// <para>Represents a range between minimum and maximum values.</para>
11     /// <para>Представляет диапазон между минимальным и максимальным значениями.</para>
12     /// </summary>
13     /// <remarks>
14     /// <para>Based on <a href="http://stackoverflow.com/questions/5343006/is-there-a-c-sharp-type-for-representing-an-integer-range">the question at StackOverflow</a>.</para>
15     /// <para>Основано на <a href="http://stackoverflow.com/questions/5343006/is-there-a-c-sharp-type-for-representing-an-integer-range">вопросе в StackOverflow</a>.</para>
16     /// </remarks>
17     public abstract class GenericCollectionMethodsBase<TElement>
18     {
19
20         /// <summary>
21         /// <para>Presents the Range in readable format.</para>
22         /// <para>Представляет диапазон в удобном для чтения формате.</para>
23         /// </summary>
24         /// <returns><para>String representation of the Range.</para><para>Строковое представление диапазона.</para></returns>
25         [MethodImpl(MethodImplOptions.AggressiveInlining)]
26         protected virtual TElement GetZero() => default;
27
28         [MethodImpl(MethodImplOptions.AggressiveInlining)]
29         protected virtual bool EqualToZero(TElement value) => EqualityComparer.Equals(value, GetZero());
30
31         [MethodImpl(MethodImplOptions.AggressiveInlining)]
32         protected virtual bool AreEqual(TElement first, TElement second) => EqualityComparer.Equals(first, second);
33
34         [MethodImpl(MethodImplOptions.AggressiveInlining)]
35         protected virtual bool GreaterThanZero(TElement value) => Comparer.Compare(value, GetZero()) > 0;
36
37         [MethodImpl(MethodImplOptions.AggressiveInlining)]
38         protected virtual bool GreaterThan(TElement first, TElement second) => Comparer.Compare(first, second) > 0;
39
40         [MethodImpl(MethodImplOptions.AggressiveInlining)]
41         protected virtual bool GreaterOrEqualThanZero(TElement value) => Comparer.Compare(value, GetZero()) >= 0;
42
43         [MethodImpl(MethodImplOptions.AggressiveInlining)]
44         protected virtual bool GreaterOrEqualThan(TElement first, TElement second) => Comparer.Compare(first, second) >= 0;
45
46         [MethodImpl(MethodImplOptions.AggressiveInlining)]
47         protected virtual bool LessOrEqualThanZero(TElement value) => Comparer.Compare(value, GetZero()) <= 0;
48
49         [MethodImpl(MethodImplOptions.AggressiveInlining)]
50         protected virtual bool LessOrEqualThan(TElement first, TElement second) => Comparer.Compare(first, second) <= 0;
51
52         [MethodImpl(MethodImplOptions.AggressiveInlining)]
53         protected virtual bool LessThanZero(TElement value) => Comparer.Compare(value, GetZero()) < 0;
54
55         [MethodImpl(MethodImplOptions.AggressiveInlining)]
56         protected virtual bool LessThan(TElement first, TElement second) => Comparer.Compare(first, second) < 0;
57
58         [MethodImpl(MethodImplOptions.AggressiveInlining)]
59         protected virtual TElement Increment(TElement value) => Arithmetic<TElement>.Increment(value);
60
61         [MethodImpl(MethodImplOptions.AggressiveInlining)]
```

```

62     protected virtual TElement Decrement(TElement value) =>
        ↪ Arithmetic<TElement>.Decrement(value);
63
64     [MethodImpl(MethodImplOptions.AggressiveInlining)]
65     protected virtual TElement Add(TElement first, TElement second) =>
        ↪ Arithmetic<TElement>.Add(first, second);
66
67     [MethodImpl(MethodImplOptions.AggressiveInlining)]
68     protected virtual TElement Subtract(TElement first, TElement second) =>
        ↪ Arithmetic<TElement>.Subtract(first, second);
69
70     protected readonly TElement Zero;
71     protected readonly TElement One;
72     protected readonly TElement Two;
73     protected readonly EqualityComparer<TElement> EqualityComparer;
74     protected readonly Comparer<TElement> Comparer;
75
76     /// <summary>
77     /// <para>Presents the Range in readable format.</para>
78     /// <para>Представляет диапазон в удобном для чтения формате.</para>
79     /// </summary>
80     /// <returns><para>String representation of the Range.</para><para>Строковое
        ↪ представление диапазона.</para></returns>
81     protected GenericCollectionMethodsBase()
82     {
83         EqualityComparer = EqualityComparer<TElement>.Default;
84         Comparer = Comparer<TElement>.Default;
85         Zero = GetZero(); //-V3068
86         One = Increment(Zero); //-V3068
87         Two = Increment(One); //-V3068
88     }
89 }
90 }

```

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

```

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

```

```

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

```

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

```

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

```

```

28         protected void IncrementSize() => SetSize(Increment(GetSize()));
29
30         [MethodImpl(MethodImplOptions.AggressiveInlining)]
31         protected void DecrementSize() => SetSize(Decrement(GetSize()));
32     }
33 }

```

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

```

1  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
2
3  namespace Platform.Collections.Methods.Lists
4  {
5      public abstract class AbsoluteOpenDoublyLinkedListMethods<TElement> :
6          ↳ AbsoluteDoublyLinkedListMethodsBase<TElement>
7      {
8          public void AttachBefore(TElement baseElement, TElement newElement)
9          {
10              var baseElementPrevious = GetPrevious(baseElement);
11              SetPrevious(newElement, baseElementPrevious);
12              SetNext(newElement, baseElement);
13              if (EqualToZero(baseElementPrevious))
14              {
15                  SetFirst(newElement);
16              }
17              else
18              {
19                  SetNext(baseElementPrevious, newElement);
20              }
21              SetPrevious(baseElement, newElement);
22              IncrementSize();
23          }
24
25          public void AttachAfter(TElement baseElement, TElement newElement)
26          {
27              var baseElementNext = GetNext(baseElement);
28              SetPrevious(newElement, baseElement);
29              SetNext(newElement, baseElementNext);
30              if (EqualToZero(baseElementNext))
31              {
32                  SetLast(newElement);
33              }
34              else
35              {
36                  SetPrevious(baseElementNext, newElement);
37              }
38              SetNext(baseElement, newElement);
39              IncrementSize();
40          }
41
42          public void AttachAsFirst(TElement element)
43          {
44              var first = GetFirst();
45              if (EqualToZero(first))
46              {
47                  SetFirst(element);
48                  SetLast(element);
49                  SetPrevious(element, Zero);
50                  SetNext(element, Zero);
51                  IncrementSize();
52              }
53              else
54              {
55                  AttachBefore(first, element);
56              }
57          }
58
59          public void AttachAsLast(TElement element)
60          {
61              var last = GetLast();
62              if (EqualToZero(last))
63              {
64                  AttachAsFirst(element);
65              }
66              else
67              {
68                  AttachAfter(last, element);
69              }
70          }
71      }

```

```

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
9      ↪ list</a> implementation.
10     /// </remarks>
11     public abstract class DoublyLinkedListMethodsBase<TElement> :
12     ↪ GenericCollectionMethodsBase<TElement>
13     {
14         [MethodImpl(MethodImplOptions.AggressiveInlining)]
15         protected abstract TElement GetPrevious(TElement element);
16
17         [MethodImpl(MethodImplOptions.AggressiveInlining)]
18         protected abstract TElement GetNext(TElement element);
19
20         [MethodImpl(MethodImplOptions.AggressiveInlining)]
21         protected abstract void SetPrevious(TElement element, TElement previous);
22
23         [MethodImpl(MethodImplOptions.AggressiveInlining)]
24         protected abstract void SetNext(TElement element, TElement next);
25     }
26 }

```

### 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> :
6      ↪ RelativeDoublyLinkedListMethodsBase<TElement>
7      {
8          public void AttachBefore(TElement headElement, TElement baseElement, TElement newElement)
9          {
10             var baseElementPrevious = GetPrevious(baseElement);
11             SetPrevious(newElement, baseElementPrevious);
12             SetNext(newElement, baseElement);
13             if (AreEqual(baseElement, GetFirst(headElement)))
14             {
15                 SetFirst(headElement, newElement);
16             }
17             SetNext(baseElementPrevious, newElement);
18             SetPrevious(baseElement, newElement);
19             IncrementSize(headElement);
20         }
21
22         public void AttachAfter(TElement headElement, TElement baseElement, TElement newElement)
23         {
24             var baseElementNext = GetNext(baseElement);
25             SetNext(baseElement, newElement);
26             SetPrevious(newElement, baseElement);
27             if (AreEqual(baseElement, GetLast(headElement)))
28             {
29                 SetLast(headElement, newElement);
30             }
31             SetNext(newElement, baseElementNext);
32             IncrementSize(headElement);
33         }
34     }
35 }

```

```

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> :
6     ↳ RelativeDoublyLinkedListMethodsBase<TElement>
7     {
8         public void AttachBefore(TElement headElement, TElement baseElement, TElement newElement)
9         {
10             var baseElementPrevious = GetPrevious(baseElement);
11             SetPrevious(newElement, baseElementPrevious);
12             SetNext(newElement, baseElement);
13             if (EqualToZero(baseElementPrevious))
14             {
15                 SetFirst(headElement, newElement);
16             }
17             else
18             {
19                 SetNext(baseElementPrevious, newElement);
20             }
21             SetPrevious(baseElement, newElement);
22             IncrementSize(headElement);
23         }
24
25         public void AttachAfter(TElement headElement, TElement baseElement, TElement newElement)
26         {
27             var baseElementNext = GetNext(baseElement);
28             SetPrevious(newElement, baseElement);
29             SetNext(newElement, baseElementNext);
30             if (EqualToZero(baseElementNext))
31             {
32                 SetLast(headElement, newElement);
33             }
34             else
35             {
36                 SetPrevious(baseElementNext, newElement);
37             }
38             SetNext(baseElement, newElement);
39             IncrementSize(headElement);
40         }
41
42         public void AttachAsFirst(TElement headElement, TElement element)
43         {
44             var first = GetFirst(headElement);
45             if (EqualToZero(first))
46             {
47                 SetFirst(headElement, element);
48             }
49         }
50     }
51 }

```

```

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

```

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

```

1  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
2
3  namespace Platform.Collections.Methods.Trees
4  {
5      public abstract class RecursionlessSizeBalancedTreeMethods<TElement> :
6          ↳ SizedBinaryTreeMethodsBase<TElement>
7      {
8          protected override void AttachCore(ref TElement root, TElement node)
9          {
10              while (true)
11              {
12                  ref var left = ref GetLeftReference(root);
13                  var leftSize = GetSizeOrZero(left);
14                  ref var right = ref GetRightReference(root);
15                  var rightSize = GetSizeOrZero(right);
16                  if (FirstIsToTheLeftOfSecond(node, root)) // node.Key less than root.Key
17                  {
18                      if (EqualToZero(left))
19                      {
20                          IncrementSize(root);
21                          SetSize(node, One);
22                          left = node;
23                          return;
24                      }
25                      if (FirstIsToTheLeftOfSecond(node, left)) // node.Key less than left.Key
26                      {
27                          if (GreaterThan(Increment(leftSize), rightSize))

```



```

27         {
28             RightRotate(ref root);
29         }
30         else
31         {
32             IncrementSize(root);
33             root = ref left;
34         }
35     }
36     else // node.Key greater than left.Key
37     {
38         var leftRightSize = GetSizeOrZero(GetRight(left));
39         if (GreaterThan(Increment(leftRightSize), rightSize))
40         {
41             if (EqualToZero(leftRightSize) && EqualToZero(rightSize))
42             {
43                 SetLeft(node, left);
44                 SetRight(node, root);
45                 SetSize(node, Add(leftSize, Two)); // Two (2) - node the size of
46                 ↪ root and a node itself
47                 SetLeft(root, Zero);
48                 SetSize(root, One);
49                 root = node;
50                 return;
51             }
52             LeftRotate(ref left);
53             RightRotate(ref root);
54         }
55         else
56         {
57             IncrementSize(root);
58             root = ref left;
59         }
60     }
61     else // node.Key greater than root.Key
62     {
63         if (EqualToZero(right))
64         {
65             IncrementSize(root);
66             SetSize(node, One);
67             right = node;
68             return;
69         }
70         if (FirstIsToTheRightOfSecond(node, right)) // node.Key greater than
71         ↪ right.Key
72         {
73             if (GreaterThan(Increment(rightSize), leftSize))
74             {
75                 LeftRotate(ref root);
76             }
77             else
78             {
79                 IncrementSize(root);
80                 root = ref right;
81             }
82         }
83         else // node.Key less than right.Key
84         {
85             var rightLeftSize = GetSizeOrZero(GetLeft(right));
86             if (GreaterThan(Increment(rightLeftSize), leftSize))
87             {
88                 if (EqualToZero(rightLeftSize) && EqualToZero(leftSize))
89                 {
90                     SetLeft(node, root);
91                     SetRight(node, right);
92                     SetSize(node, Add(rightSize, Two)); // Two (2) - node the size
93                     ↪ of root and a node itself
94                     SetRight(root, Zero);
95                     SetSize(root, One);
96                     root = node;
97                     return;
98                 }
99                 RightRotate(ref right);
100                 LeftRotate(ref root);
101             }
102             else
103             {
104                 IncrementSize(root);

```

```

103         root = ref right;
104     }
105 }
106 }
107 }
108 }
109
110 protected override void DetachCore(ref TElement root, TElement node)
111 {
112     while (true)
113     {
114         ref var left = ref GetLeftReference(root);
115         var leftSize = GetSizeOrZero(left);
116         ref var right = ref GetRightReference(root);
117         var rightSize = GetSizeOrZero(right);
118         if (FirstIsToTheLeftOfSecond(node, root)) // node.Key less than root.Key
119         {
120             var decrementedLeftSize = Decrement(leftSize);
121             if (GreaterThan(GetSizeOrZero(GetRightOrDefault(right)),
122                 ↪ decrementedLeftSize))
123             {
124                 LeftRotate(ref root);
125             }
126             else if (GreaterThan(GetSizeOrZero(GetLeftOrDefault(right)),
127                 ↪ decrementedLeftSize))
128             {
129                 RightRotate(ref right);
130                 LeftRotate(ref root);
131             }
132             else
133             {
134                 DecrementSize(root);
135                 root = ref left;
136             }
137         }
138         else if (FirstIsToTheRightOfSecond(node, root)) // node.Key greater than root.Key
139         {
140             var decrementedRightSize = Decrement(rightSize);
141             if (GreaterThan(GetSizeOrZero(GetLeftOrDefault(left)), decrementedRightSize))
142             {
143                 RightRotate(ref root);
144             }
145             else if (GreaterThan(GetSizeOrZero(GetRightOrDefault(left)),
146                 ↪ decrementedRightSize))
147             {
148                 LeftRotate(ref left);
149                 RightRotate(ref root);
150             }
151             else
152             {
153                 DecrementSize(root);
154                 root = ref right;
155             }
156         }
157         else // key equals to root.Key
158         {
159             if (GreaterThanZero(leftSize) && GreaterThanZero(rightSize))
160             {
161                 TElement replacement;
162                 if (GreaterThan(leftSize, rightSize))
163                 {
164                     replacement = GetRighttest(left);
165                     DetachCore(ref left, replacement);
166                 }
167                 else
168                 {
169                     replacement = GetLefttest(right);
170                     DetachCore(ref right, replacement);
171                 }
172                 SetLeft(replacement, left);
173                 SetRight(replacement, right);
174                 SetSize(replacement, Add(leftSize, rightSize));
175                 root = replacement;
176             }
177             else if (GreaterThanZero(leftSize))
178             {
179                 root = left;
180             }
181         }
182     }
183 }

```

```

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

```

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> :
8      ↪ SizedBinaryTreeMethodsBase<TElement>
9      {
10         protected override void AttachCore(ref TElement root, TElement node)
11         {
12             if (EqualToZero(root))
13             {
14                 root = node;
15                 IncrementSize(root);
16             }
17             else
18             {
19                 IncrementSize(root);
20                 if (FirstIsToTheLeftOfSecond(node, root))
21                 {
22                     AttachCore(ref GetLeftReference(root), node);
23                     LeftMaintain(ref root);
24                 }
25                 else
26                 {
27                     AttachCore(ref GetRightReference(root), node);
28                     RightMaintain(ref root);
29                 }
30             }
31         }
32         protected override void DetachCore(ref TElement root, TElement nodeToDetach)
33         {
34             ref var currentNode = ref root;
35             ref var parent = ref root;
36             var replacementNode = Zero;
37             while (!AreEqual(currentNode, nodeToDetach))
38             {
39                 DecrementSize(currentNode);
40                 if (FirstIsToTheLeftOfSecond(nodeToDetach, currentNode))
41                 {
42                     parent = ref currentNode;
43                     currentNode = ref GetLeftReference(currentNode);
44                 }
45                 else if (FirstIsToTheRightOfSecond(nodeToDetach, currentNode))
46                 {
47                     parent = ref currentNode;
48                     currentNode = ref GetRightReference(currentNode);
49                 }
50                 else
51                 {
52                     throw new InvalidOperationException("Duplicate link found in the tree.");
53                 }
54             }
55             var nodeToDetachLeft = GetLeft(nodeToDetach);
56             var node = GetRight(nodeToDetach);
57             if (!EqualToZero(nodeToDetachLeft) && !EqualToZero(node))
58             {
59                 var lefttestNode = GetLefttest(node);
60                 DetachCore(ref GetRightReference(nodeToDetach), lefttestNode);
61                 SetLeft(lefttestNode, nodeToDetachLeft);

```

```

62     node = GetRight(nodeToDetach);
63     if (!EqualToZero(node))
64     {
65         SetRight(lefttestNode, node);
66         SetSize(lefttestNode, Increment(Add(GetSize(nodeToDetachLeft),
        ↪ GetSize(node))));
67     }
68     else
69     {
70         SetSize(lefttestNode, Increment(GetSize(nodeToDetachLeft)));
71     }
72     replacementNode = lefttestNode;
73 }
74 else if (!EqualToZero(nodeToDetachLeft))
75 {
76     replacementNode = nodeToDetachLeft;
77 }
78 else if (!EqualToZero(node))
79 {
80     replacementNode = node;
81 }
82 if (AreEqual(root, nodeToDetach))
83 {
84     root = replacementNode;
85 }
86 else if (AreEqual(GetLeft(parent), nodeToDetach))
87 {
88     SetLeft(parent, replacementNode);
89 }
90 else if (AreEqual(GetRight(parent), nodeToDetach))
91 {
92     SetRight(parent, replacementNode);
93 }
94 ClearNode(nodeToDetach);
95 }
96
97 private void LeftMaintain(ref TElement root)
98 {
99     if (!EqualToZero(root))
100     {
101         var rootLeftNode = GetLeft(root);
102         if (!EqualToZero(rootLeftNode))
103         {
104             var rootRightNode = GetRight(root);
105             var rootRightNodeSize = GetSize(rootRightNode);
106             var rootLeftNodeLeftNode = GetLeft(rootLeftNode);
107             if (!EqualToZero(rootLeftNodeLeftNode) &&
108                 (EqualToZero(rootRightNode) ||
109                 ↪ GreaterThan(GetSize(rootLeftNodeLeftNode), rootRightNodeSize)))
110             {
111                 RightRotate(ref root);
112             }
113             else
114             {
115                 var rootLeftNodeRightNode = GetRight(rootLeftNode);
116                 if (!EqualToZero(rootLeftNodeRightNode) &&
117                     (EqualToZero(rootRightNode) ||
118                     ↪ GreaterThan(GetSize(rootLeftNodeRightNode), rootRightNodeSize)))
119                 {
120                     LeftRotate(ref GetLeftReference(root));
121                     RightRotate(ref root);
122                 }
123                 else
124                 {
125                     return;
126                 }
127             }
128             LeftMaintain(ref GetLeftReference(root));
129             RightMaintain(ref GetRightReference(root));
130             LeftMaintain(ref root);
131             RightMaintain(ref root);
132         }
133     }
134 }
135
136 private void RightMaintain(ref TElement root)
137 {
138     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) ||
146                  ⇨ GreaterThan(GetSize(rootRightNodeRightNode), rootLeftNodeSize)))
147             {
148                 LeftRotate(ref root);
149             }
150             else
151             {
152                 var rootRightNodeLeftNode = GetLeft(rootRightNode);
153                 if (!EqualToZero(rootRightNodeLeftNode) &&
154                     (EqualToZero(rootLeftNode) ||
155                      ⇨ GreaterThan(GetSize(rootRightNodeLeftNode), rootLeftNodeSize)))
156                 {
157                     RightRotate(ref GetRightReference(root));
158                     LeftRotate(ref root);
159                 }
160                 else
161                 {
162                     return;
163                 }
164             }
165             LeftMaintain(ref GetLeftReference(root));
166             RightMaintain(ref GetRightReference(root));
167             LeftMaintain(ref root);
168             RightMaintain(ref root);
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 {
13     /// <summary>
14     /// Combination of Size, Height (AVL), and threads.
15     /// </summary>
16     /// <remarks>
17     /// Based on: <a href="https://github.com/programatom/TreeLib/blob/master/TreeLib/TreeLib/G_
18     ⇨ enerated/AVLTreeList.cs">TreeLib.AVLTreeList</a>.
19     /// Which itself based on: <a
20     ⇨ href="https://github.com/GNOME/glib/blob/master/glib/gtree.c">GNOME/glib/gtree</a>.
21     /// </remarks>
22     public abstract class SizedAndThreadedAVLBalancedTreeMethods<TElement> :
23     ⇨ SizedBinaryTreeMethodsBase<TElement>
24     {
25         private static readonly int _maxPath = 11 * NumericType<TElement>.BytesSize + 4;
26
27         [MethodImpl(MethodImplOptions.AggressiveInlining)]
28         protected override TElement GetRightest(TElement current)
29         {
30             var currentRight = GetRightOrDefault(current);
31             while (!EqualToZero(currentRight))
32             {
33                 current = currentRight;
34                 currentRight = GetRightOrDefault(current);
35             }
36             return current;
37         }
38
39         [MethodImpl(MethodImplOptions.AggressiveInlining)]
40         protected override TElement GetLeftest(TElement current)

```

```

38 {
39     var currentLeft = GetLeftOrDefault(current);
40     while (!EqualToZero(currentLeft))
41     {
42         current = currentLeft;
43         currentLeft = GetLeftOrDefault(current);
44     }
45     return current;
46 }
47
48 public override bool Contains(TElement node, TElement root)
49 {
50     while (!EqualToZero(root))
51     {
52         if (FirstIsToTheLeftOfSecond(node, root)) // node.Key < root.Key
53         {
54             root = GetLeftOrDefault(root);
55         }
56         else if (FirstIsToTheRightOfSecond(node, root)) // node.Key > root.Key
57         {
58             root = GetRightOrDefault(root);
59         }
60         else // node.Key == root.Key
61         {
62             return true;
63         }
64     }
65     return false;
66 }
67
68 protected override void PrintNode(TElement node, StringBuilder sb, int level)
69 {
70     base.PrintNode(node, sb, level);
71     sb.Append(' ');
72     sb.Append(GetLeftIsChild(node) ? 'l' : 'L');
73     sb.Append(GetRightIsChild(node) ? 'r' : 'R');
74     sb.Append(' ');
75     sb.Append(GetBalance(node));
76 }
77
78 [MethodImpl(MethodImplOptions.AggressiveInlining)]
79 protected void IncrementBalance(TElement node) => SetBalance(node,
80     ↪ (sbyte)(GetBalance(node) + 1));
81
82 [MethodImpl(MethodImplOptions.AggressiveInlining)]
83 protected void DecrementBalance(TElement node) => SetBalance(node,
84     ↪ (sbyte)(GetBalance(node) - 1));
85
86 [MethodImpl(MethodImplOptions.AggressiveInlining)]
87 protected override TElement GetLeftOrDefault(TElement node) => GetLeftIsChild(node) ?
88     ↪ GetLeft(node) : default;
89
90 [MethodImpl(MethodImplOptions.AggressiveInlining)]
91 protected override TElement GetRightOrDefault(TElement node) => GetRightIsChild(node) ?
92     ↪ GetRight(node) : default;
93
94 [MethodImpl(MethodImplOptions.AggressiveInlining)]
95 protected abstract bool GetLeftIsChild(TElement node);
96
97 [MethodImpl(MethodImplOptions.AggressiveInlining)]
98 protected abstract void SetLeftIsChild(TElement node, bool value);
99
100 [MethodImpl(MethodImplOptions.AggressiveInlining)]
101 protected abstract bool GetRightIsChild(TElement node);
102
103 [MethodImpl(MethodImplOptions.AggressiveInlining)]
104 protected abstract void SetRightIsChild(TElement node, bool value);
105
106 [MethodImpl(MethodImplOptions.AggressiveInlining)]
107 protected abstract sbyte GetBalance(TElement node);
108
109 [MethodImpl(MethodImplOptions.AggressiveInlining)]
110 protected abstract void SetBalance(TElement node, sbyte value);
111
112 protected override void AttachCore(ref TElement root, TElement node)
113 {
114     unchecked
115     {
116         // TODO: Check what is faster to use simple array or array from array pool

```

```

113 // TODO: Try to use stackalloc as an optimization (requires code generation,
114 // ↳ because of generics)
115 #if USEARRAYPOOL
116 var path = ArrayPool.Allocate<TElement>(MaxPath);
117 var pathPosition = 0;
118 path[pathPosition++] = default;
119 #else
120 var path = new TElement[_maxPath];
121 var pathPosition = 1;
122 #endif
123 var currentNode = root;
124 while (true)
125 {
126     if (FirstIsToTheLeftOfSecond(node, currentNode))
127     {
128         if (GetLeftIsChild(currentNode))
129         {
130             IncrementSize(currentNode);
131             path[pathPosition++] = currentNode;
132             currentNode = GetLeft(currentNode);
133         }
134         else
135         {
136             // Threads
137             SetLeft(node, GetLeft(currentNode));
138             SetRight(node, currentNode);
139             SetLeft(currentNode, node);
140             SetLeftIsChild(currentNode, true);
141             DecrementBalance(currentNode);
142             SetSize(node, One);
143             FixSize(currentNode); // Should be incremented already
144             break;
145         }
146     }
147     else if (FirstIsToTheRightOfSecond(node, currentNode))
148     {
149         if (GetRightIsChild(currentNode))
150         {
151             IncrementSize(currentNode);
152             path[pathPosition++] = currentNode;
153             currentNode = GetRight(currentNode);
154         }
155         else
156         {
157             // Threads
158             SetRight(node, GetRight(currentNode));
159             SetLeft(node, currentNode);
160             SetRight(currentNode, node);
161             SetRightIsChild(currentNode, true);
162             IncrementBalance(currentNode);
163             SetSize(node, One);
164             FixSize(currentNode); // Should be incremented already
165             break;
166         }
167     }
168     else
169     {
170         throw new InvalidOperationException("Node with the same key already
171         ↳ attached to a tree.");
172     }
173 }
174 // Restore balance. This is the goodness of a non-recursive
175 // implementation, when we are done with balancing we 'break'
176 // the loop and we are done.
177 while (true)
178 {
179     var parent = path[--pathPosition];
180     var isLeftNode = !AreEqual(parent, default) && AreEqual(currentNode,
181     ↳ GetLeft(parent));
182     var currentNodeBalance = GetBalance(currentNode);
183     if (currentNodeBalance < -1 || currentNodeBalance > 1)
184     {
185         currentNode = Balance(currentNode);
186         if (AreEqual(parent, default))
187         {
188             root = currentNode;
189         }
190         else if (isLeftNode)

```

```

188         {
189             SetLeft(parent, currentNode);
190             FixSize(parent);
191         }
192         else
193         {
194             SetRight(parent, currentNode);
195             FixSize(parent);
196         }
197     }
198     currentNodeBalance = GetBalance(currentNode);
199     if (currentNodeBalance == 0 || AreEqual(parent, default))
200     {
201         break;
202     }
203     if (isLeftNode)
204     {
205         DecrementBalance(parent);
206     }
207     else
208     {
209         IncrementBalance(parent);
210     }
211     currentNode = parent;
212 }
213 #if USEARRAYPOOL
214     ArrayPool.Free(path);
215 #endif
216 }
217 }
218
219 private TElement Balance(TElement node)
220 {
221     unchecked
222     {
223         var rootBalance = GetBalance(node);
224         if (rootBalance < -1)
225         {
226             var left = GetLeft(node);
227             if (GetBalance(left) > 0)
228             {
229                 SetLeft(node, LeftRotateWithBalance(left));
230                 FixSize(node);
231             }
232             node = RightRotateWithBalance(node);
233         }
234         else if (rootBalance > 1)
235         {
236             var right = GetRight(node);
237             if (GetBalance(right) < 0)
238             {
239                 SetRight(node, RightRotateWithBalance(right));
240                 FixSize(node);
241             }
242             node = LeftRotateWithBalance(node);
243         }
244         return node;
245     }
246 }
247
248 protected TElement LeftRotateWithBalance(TElement node)
249 {
250     unchecked
251     {
252         var right = GetRight(node);
253         if (GetLeftIsChild(right))
254         {
255             SetRight(node, GetLeft(right));
256         }
257         else
258         {
259             SetRightIsChild(node, false);
260             SetLeftIsChild(right, true);
261         }
262         SetLeft(right, node);
263         // Fix size
264         SetSize(right, GetSize(node));
265         FixSize(node);
266         // Fix balance

```



```

267     var rootBalance = GetBalance(node);
268     var rightBalance = GetBalance(right);
269     if (rightBalance <= 0)
270     {
271         if (rootBalance >= 1)
272         {
273             SetBalance(right, (sbyte)(rightBalance - 1));
274         }
275         else
276         {
277             SetBalance(right, (sbyte)(rootBalance + rightBalance - 2));
278         }
279         SetBalance(node, (sbyte)(rootBalance - 1));
280     }
281     else
282     {
283         if (rootBalance <= rightBalance)
284         {
285             SetBalance(right, (sbyte)(rootBalance - 2));
286         }
287         else
288         {
289             SetBalance(right, (sbyte)(rightBalance - 1));
290         }
291         SetBalance(node, (sbyte)(rootBalance - rightBalance - 1));
292     }
293     return right;
294 }
295
296
297 protected TElement RightRotateWithBalance(TElement node)
298 {
299     unchecked
300     {
301         var left = GetLeft(node);
302         if (GetRightIsChild(left))
303         {
304             SetLeft(node, GetRight(left));
305         }
306         else
307         {
308             SetLeftIsChild(node, false);
309             SetRightIsChild(left, true);
310         }
311         SetRight(left, node);
312         // Fix size
313         SetSize(left, GetSize(node));
314         FixSize(node);
315         // Fix balance
316         var rootBalance = GetBalance(node);
317         var leftBalance = GetBalance(left);
318         if (leftBalance <= 0)
319         {
320             if (leftBalance > rootBalance)
321             {
322                 SetBalance(left, (sbyte)(leftBalance + 1));
323             }
324             else
325             {
326                 SetBalance(left, (sbyte)(rootBalance + 2));
327             }
328             SetBalance(node, (sbyte)(rootBalance - leftBalance + 1));
329         }
330         else
331         {
332             if (rootBalance <= -1)
333             {
334                 SetBalance(left, (sbyte)(leftBalance + 1));
335             }
336             else
337             {
338                 SetBalance(left, (sbyte)(rootBalance + leftBalance + 2));
339             }
340             SetBalance(node, (sbyte)(rootBalance + 1));
341         }
342         return left;
343     }
344 }
345

```

```

346 [MethodImpl(MethodImplOptions.AggressiveInlining)]
347 protected override TElement GetNext(TElement node)
348 {
349     var current = GetRight(node);
350     if (GetRightIsChild(node))
351     {
352         return GetLefttest(current);
353     }
354     return current;
355 }
356
357 [MethodImpl(MethodImplOptions.AggressiveInlining)]
358 protected override TElement GetPrevious(TElement node)
359 {
360     var current = GetLeft(node);
361     if (GetLeftIsChild(node))
362     {
363         return GetRighttest(current);
364     }
365     return current;
366 }
367
368 protected override void DetachCore(ref TElement root, TElement node)
369 {
370     unchecked
371     {
372 #if USEARRAYPOOL
373         var path = ArrayPool.Allocate<TElement>(MaxPath);
374         var pathPosition = 0;
375         path[pathPosition++] = default;
376 #else
377         var path = new TElement[_maxPath];
378         var pathPosition = 1;
379 #endif
380         var currentNode = root;
381         while (true)
382         {
383             if (FirstIsToTheLeftOfSecond(node, currentNode))
384             {
385                 if (!GetLeftIsChild(currentNode))
386                 {
387                     throw new InvalidOperationException("Cannot find a node.");
388                 }
389                 DecrementSize(currentNode);
390                 path[pathPosition++] = currentNode;
391                 currentNode = GetLeft(currentNode);
392             }
393             else if (FirstIsToTheRightOfSecond(node, currentNode))
394             {
395                 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,
411 ↪ GetLeft(parent));
412         if (!GetLeftIsChild(currentNode))
413         {
414             if (!GetRightIsChild(currentNode)) // node has no children
415             {
416                 if (AreEqual(parent, default))
417                 {
418                     root = Zero;
419                 }
420                 else if (isLeftNode)
421                 {
422                     SetLeftIsChild(parent, false);
423                     SetLeft(parent, GetLeft(currentNode));
424                     IncrementBalance(parent);
425                 }
426             }
427         }
428     }
429 }

```

```

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);
473         }
474     }
475     else // node has a both children (left and right)
476     {
477         var predecessor = GetLeft(currentNode);
478         var successor = GetRight(currentNode);
479         var successorParent = currentNode;
480         int previousPathPosition = ++pathPosition;
481         // find the immediately next node (and its parent)
482         while (GetLeftIsChild(successor))
483         {
484             path[++pathPosition] = successorParent = successor;
485             successor = GetLeft(successor);
486             if (!AreEqual(successorParent, currentNode))
487             {
488                 DecrementSize(successorParent);
489             }
490         }
491         path[previousPathPosition] = successor;
492         balanceNode = path[pathPosition];
493         // remove 'successor' from the tree
494         if (!AreEqual(successorParent, currentNode))
495         {
496             if (!GetRightIsChild(successor))
497             {
498                 SetLeftIsChild(successorParent, false);
499             }
500             else
501             {

```

```

502         SetLeft(successorParent, GetRight(successor));
503     }
504     IncrementBalance(successorParent);
505     SetRightIsChild(successor, true);
506     SetRight(successor, GetRight(currentNode));
507 }
508 else
509 {
510     DecrementBalance(currentNode);
511 }
512 // set the predecessor's successor link to point to the right place
513 while (GetRightIsChild(predecessor))
514 {
515     predecessor = GetRight(predecessor);
516 }
517 SetRight(predecessor, successor);
518 // prepare 'successor' to replace 'node'
519 var left = GetLeft(currentNode);
520 SetLeftIsChild(successor, true);
521 SetLeft(successor, left);
522 SetBalance(successor, GetBalance(currentNode));
523 FixSize(successor);
524 if (AreEqual(parent, default))
525 {
526     root = successor;
527 }
528 else if (isLeftNode)
529 {
530     SetLeft(parent, successor);
531 }
532 else
533 {
534     SetRight(parent, successor);
535 }
536 }
537 }
538 // restore balance
539 if (!AreEqual(balanceNode, default))
540 {
541     while (true)
542     {
543         var balanceParent = path[--pathPosition];
544         isLeftNode = !AreEqual(balanceParent, default) && AreEqual(balanceNode,
545             ↪ GetLeft(balanceParent));
546         var currentNodeBalance = GetBalance(balanceNode);
547         if (currentNodeBalance < -1 || currentNodeBalance > 1)
548         {
549             balanceNode = Balance(balanceNode);
550             if (AreEqual(balanceParent, default))
551             {
552                 root = balanceNode;
553             }
554             else if (isLeftNode)
555             {
556                 SetLeft(balanceParent, balanceNode);
557             }
558             else
559             {
560                 SetRight(balanceParent, balanceNode);
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> :
14         ↳ GenericCollectionMethodsBase<TElement>
15     {
16         [MethodImpl(MethodImplOptions.AggressiveInlining)]
17         protected abstract ref TElement GetLeftReference(TElement node);
18
19         [MethodImpl(MethodImplOptions.AggressiveInlining)]
20         protected abstract ref TElement GetRightReference(TElement node);
21
22         [MethodImpl(MethodImplOptions.AggressiveInlining)]
23         protected abstract TElement GetLeft(TElement node);
24
25         [MethodImpl(MethodImplOptions.AggressiveInlining)]
26         protected abstract TElement GetRight(TElement node);
27
28         [MethodImpl(MethodImplOptions.AggressiveInlining)]
29         protected abstract TElement GetSize(TElement node);
30
31         [MethodImpl(MethodImplOptions.AggressiveInlining)]
32         protected abstract void SetLeft(TElement node, TElement left);
33
34         [MethodImpl(MethodImplOptions.AggressiveInlining)]
35         protected abstract void SetRight(TElement node, TElement right);
36
37         [MethodImpl(MethodImplOptions.AggressiveInlining)]
38         protected abstract void SetSize(TElement node, TElement size);
39
40         [MethodImpl(MethodImplOptions.AggressiveInlining)]
41         protected abstract bool FirstIsToTheLeftOfSecond(TElement first, TElement second);
42
43         [MethodImpl(MethodImplOptions.AggressiveInlining)]
44         protected abstract bool FirstIsToTheRightOfSecond(TElement first, TElement second);
45
46         [MethodImpl(MethodImplOptions.AggressiveInlining)]
47         protected virtual TElement GetLeftOrDefault(TElement node) => AreEqual(node, default) ?
48             ↳ default : GetLeft(node);
49
50         [MethodImpl(MethodImplOptions.AggressiveInlining)]
51         protected virtual TElement GetRightOrDefault(TElement node) => AreEqual(node, default) ?
52             ↳ default : GetRight(node);
53
54         [MethodImpl(MethodImplOptions.AggressiveInlining)]
55         protected void IncrementSize(TElement node) => SetSize(node, Increment(GetSize(node)));
56
57         [MethodImpl(MethodImplOptions.AggressiveInlining)]
58         protected void DecrementSize(TElement node) => SetSize(node, Decrement(GetSize(node)));
59
60         [MethodImpl(MethodImplOptions.AggressiveInlining)]

```

```

58     protected TElement GetLeftSize(TElement node) => GetSizeOrZero(GetLeftOrDefault(node));
59
60     [MethodImpl(MethodImplOptions.AggressiveInlining)]
61     protected TElement GetRightSize(TElement node) => GetSizeOrZero(GetRightOrDefault(node));
62
63     [MethodImpl(MethodImplOptions.AggressiveInlining)]
64     protected TElement GetSizeOrZero(TElement node) => EqualToZero(node) ? Zero :
        ↳ GetSize(node);
65
66     [MethodImpl(MethodImplOptions.AggressiveInlining)]
67     protected void FixSize(TElement node) => SetSize(node, Increment(Add(GetLeftSize(node),
        ↳ GetRightSize(node))));
68
69     [MethodImpl(MethodImplOptions.AggressiveInlining)]
70     protected void LeftRotate(ref TElement root) => root = LeftRotate(root);
71
72     [MethodImpl(MethodImplOptions.AggressiveInlining)]
73     protected TElement LeftRotate(TElement root)
74     {
75         var right = GetRight(root);
76         #if ENABLE_TREE_AUTO_DEBUG_AND_VALIDATION
77         if (EqualToZero(right))
78         {
79             throw new InvalidOperationException("Right is null.");
80         }
81         #endif
82         SetRight(root, GetLeft(right));
83         SetLeft(right, root);
84         SetSize(right, GetSize(root));
85         FixSize(root);
86         return right;
87     }
88
89     [MethodImpl(MethodImplOptions.AggressiveInlining)]
90     protected void RightRotate(ref TElement root) => root = RightRotate(root);
91
92     [MethodImpl(MethodImplOptions.AggressiveInlining)]
93     protected TElement RightRotate(TElement root)
94     {
95         var left = GetLeft(root);
96         #if ENABLE_TREE_AUTO_DEBUG_AND_VALIDATION
97         if (EqualToZero(left))
98         {
99             throw new InvalidOperationException("Left is null.");
100         }
101         #endif
102         SetLeft(root, GetRight(left));
103         SetRight(left, root);
104         SetSize(left, GetSize(root));
105         FixSize(root);
106         return left;
107     }
108
109     [MethodImpl(MethodImplOptions.AggressiveInlining)]
110     protected virtual TElement GetRighttest(TElement current)
111     {
112         var currentRight = GetRight(current);
113         while (!EqualToZero(currentRight))
114         {
115             current = currentRight;
116             currentRight = GetRight(current);
117         }
118         return current;
119     }
120
121     [MethodImpl(MethodImplOptions.AggressiveInlining)]
122     protected virtual TElement GetLefttest(TElement current)
123     {
124         var currentLeft = GetLeft(current);
125         while (!EqualToZero(currentLeft))
126         {
127             current = currentLeft;
128             currentLeft = GetLeft(current);
129         }
130         return current;
131     }
132
133     [MethodImpl(MethodImplOptions.AggressiveInlining)]
134     protected virtual TElement GetNext(TElement node) => GetLefttest(GetRight(node));

```

```

135 [MethodImpl(MethodImplOptions.AggressiveInlining)]
136 protected virtual TElement GetPrevious(TElement node) => GetRightest(GetLeft(node));
137
138 [MethodImpl(MethodImplOptions.AggressiveInlining)]
139 public virtual bool Contains(TElement node, TElement root)
140 {
141     while (!EqualToZero(root))
142     {
143         if (FirstIsToTheLeftOfSecond(node, root)) // node.Key < root.Key
144         {
145             root = GetLeft(root);
146         }
147         else if (FirstIsToTheRightOfSecond(node, root)) // node.Key > root.Key
148         {
149             root = GetRight(root);
150         }
151         else // node.Key == root.Key
152         {
153             return true;
154         }
155     }
156     return false;
157 }
158
159 [MethodImpl(MethodImplOptions.AggressiveInlining)]
160 protected virtual void ClearNode(TElement node)
161 {
162     SetLeft(node, Zero);
163     SetRight(node, Zero);
164     SetSize(node, Zero);
165 }
166
167 [MethodImpl(MethodImplOptions.AggressiveInlining)]
168 public void Attach(ref TElement root, TElement node)
169 {
170     #if ENABLE_TREE_AUTO_DEBUG_AND_VALIDATION
171         ValidateSizes(root);
172         Debug.WriteLine("--BeforeAttach--");
173         Debug.WriteLine(PrintNodes(root));
174         Debug.WriteLine("-----");
175         var sizeBefore = GetSize(root);
176     #endif
177     if (EqualToZero(root))
178     {
179         SetSize(node, One);
180         root = node;
181         return;
182     }
183     AttachCore(ref root, node);
184     #if ENABLE_TREE_AUTO_DEBUG_AND_VALIDATION
185         Debug.WriteLine("--AfterAttach--");
186         Debug.WriteLine(PrintNodes(root));
187         Debug.WriteLine("-----");
188         ValidateSizes(root);
189         var sizeAfter = GetSize(root);
190         if (!AreEqual(Arithmetic.Increment(sizeBefore), sizeAfter))
191         {
192             throw new InvalidOperationException("Tree was broken after attach.");
193         }
194     #endif
195 }
196
197 protected abstract void AttachCore(ref TElement root, TElement node);
198
199 [MethodImpl(MethodImplOptions.AggressiveInlining)]
200 public void Detach(ref TElement root, TElement node)
201 {
202     #if ENABLE_TREE_AUTO_DEBUG_AND_VALIDATION
203         ValidateSizes(root);
204         Debug.WriteLine("--BeforeDetach--");
205         Debug.WriteLine(PrintNodes(root));
206         Debug.WriteLine("-----");
207         var sizeBefore = GetSize(root);
208         if (EqualToZero(root))
209         {
210             throw new InvalidOperationException($"Элемент с {node} не содержится в
211                 ↳ дереве.");
212         }
213     #endif
214 }

```

```

213 #endif
214     DetachCore(ref root, node);
215 #if ENABLE_TREE_AUTO_DEBUG_AND_VALIDATION
216     Debug.WriteLine("--AfterDetach--");
217     Debug.WriteLine(PrintNodes(root));
218     Debug.WriteLine("-----");
219     ValidateSizes(root);
220     var sizeAfter = GetSize(root);
221     if (!AreEqual(Arithmetic.Decrement(sizeBefore), sizeAfter))
222     {
223         throw new InvalidOperationException("Tree was broken after detach.");
224     }
225 #endif
226 }
227
228 protected abstract void DetachCore(ref TElement root, TElement node);
229
230 public void FixSizes(TElement node)
231 {
232     if (AreEqual(node, default))
233     {
234         return;
235     }
236     FixSizes(GetLeft(node));
237     FixSizes(GetRight(node));
238     FixSize(node);
239 }
240
241 public void ValidateSizes(TElement node)
242 {
243     if (AreEqual(node, default))
244     {
245         return;
246     }
247     var size = GetSize(node);
248     var leftSize = GetLeftSize(node);
249     var rightSize = GetRightSize(node);
250     var expectedSize = Arithmetic.Increment(Arithmetic.Add(leftSize, rightSize));
251     if (!AreEqual(size, expectedSize))
252     {
253         throw new InvalidOperationException($"Size of {node} is not valid. Expected
254             ↳ size: {expectedSize}, actual size: {size}.");
255     }
256     ValidateSizes(GetLeft(node));
257     ValidateSizes(GetRight(node));
258 }
259
260 public void ValidateSize(TElement node)
261 {
262     var size = GetSize(node);
263     var leftSize = GetLeftSize(node);
264     var rightSize = GetRightSize(node);
265     var expectedSize = Arithmetic.Increment(Arithmetic.Add(leftSize, rightSize));
266     if (!AreEqual(size, expectedSize))
267     {
268         throw new InvalidOperationException($"Size of {node} is not valid. Expected
269             ↳ size: {expectedSize}, actual size: {size}.");
270     }
271 }
272
273 public string PrintNodes(TElement node)
274 {
275     var sb = new StringBuilder();
276     PrintNodes(node, sb);
277     return sb.ToString();
278 }
279
280 [MethodImpl(MethodImplOptions.AggressiveInlining)]
281 public void PrintNodes(TElement node, StringBuilder sb) => PrintNodes(node, sb, 0);
282
283 public void PrintNodes(TElement node, StringBuilder sb, int level)
284 {
285     if (AreEqual(node, default))
286     {
287         return;
288     }
289     PrintNodes(GetLeft(node), sb, level + 1);
290     PrintNode(node, sb, level);
291 }

```



```

289         sb.AppendLine();
290         PrintNodes(GetRight(node), sb, level + 1);
291     }
292
293     public string PrintNode(TElement node)
294     {
295         var sb = new StringBuilder();
296         PrintNode(node, sb);
297         return sb.ToString();
298     }
299
300     [MethodImpl(MethodImplOptions.AggressiveInlining)]
301     protected void PrintNode(TElement node, StringBuilder sb) => PrintNode(node, sb, 0);
302
303     protected virtual void PrintNode(TElement node, StringBuilder sb, int level)
304     {
305         sb.Append('\t', level);
306         sb.Append(node);
307         PrintNodeValue(node, sb);
308         sb.Append(' ');
309         sb.Append('s');
310         sb.Append(GetSize(node));
311     }
312
313     protected abstract void PrintNodeValue(TElement node, StringBuilder sb);
314 }
315 }

```

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

```

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

```

```

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;
3  using System.Text;
4  using Platform.Numbers;
5  using Platform.Collections.Methods.Trees;
6  using Platform.Converters;
7
8  namespace Platform.Collections.Methods.Tests
9  {
10     public class SizeBalancedTree<TElement> : SizeBalancedTreeMethods<TElement>
11     {
12         private struct TreeElement
13         {
14             public TElement Size;
15             public TElement Left;
16             public TElement Right;
17         }
18
19         private readonly TreeElement[] _elements;
20         private TElement _allocated;
21
22         public TElement Root;
23
24         public TElement Count => GetSizeOrZero(Root);
25
26         public SizeBalancedTree(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) =>
60     ↪ EqualityComparer<TreeElement>.Default.Equals(GetElement(node), default);
61
62 protected override bool FirstIsToTheLeftOfSecond(TElement first, TElement second) =>
63     ↪ Comparer.Compare(first, second) < 0;
64
65 protected override bool FirstIsToTheRightOfSecond(TElement first, TElement second) =>
66     ↪ Comparer.Compare(first, second) > 0;
67
68 protected override ref TElement GetLeftReference(TElement node) => ref
69     ↪ GetElement(node).Left;
70
71 protected override TElement GetLeft(TElement node) => GetElement(node).Left;
72
73 protected override ref TElement GetRightReference(TElement node) => ref
74     ↪ GetElement(node).Right;
75
76 protected override TElement GetRight(TElement node) => GetElement(node).Right;
77
78 protected override TElement GetSize(TElement node) => GetElement(node).Size;
79
80 protected override void PrintNodeValue(TElement node, StringBuilder sb) =>
81     ↪ sb.Append(node);
82
83 protected override void SetLeft(TElement node, TElement left) => GetElement(node).Left =
84     ↪ left;
85
86 protected override void SetRight(TElement node, TElement right) =>
87     ↪ GetElement(node).Right = right;
88
89 protected override void SetSize(TElement node, TElement size) => GetElement(node).Size =
90     ↪ size;
91
92 private ref TreeElement GetElement(TElement node) => ref
93     ↪ _elements[UncheckedConverter<TElement, long>.Default.Convert(node)];
94 }
95 }

```

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

```

1  using System;
2  using System.Collections.Generic;
3  using System.Text;
4  using Platform.Numbers;
5  using Platform.Collections.Methods.Trees;
6  using Platform.Converters;
7
8  namespace Platform.Collections.Methods.Tests
9  {
10     public class SizedAndThreadedAVLBalancedTree<TElement> :
11         ↪ SizedAndThreadedAVLBalancedTreeMethods<TElement>
12     {
13         private struct TreeElement

```

```

13 {
14     public TElement Size;
15     public TElement Left;
16     public TElement Right;
17     public sbyte Balance;
18     public bool LeftIsChild;
19     public bool RightIsChild;
20 }
21
22 private readonly TreeElement[] _elements;
23 private TElement _allocated;
24
25 public TElement Root;
26
27 public TElement Count => GetSizeOrZero(Root);
28
29 public SizedAndThreadedAVLBalancedTree(int capacity) => (_elements, _allocated) = (new
    ↳ TreeElement[capacity], One);
30
31 public TElement Allocate()
32 {
33     var newNode = _allocated;
34     if (IsEmpty(newNode))
35     {
36         _allocated = Arithmetic.Increment(_allocated);
37         return newNode;
38     }
39     else
40     {
41         throw new InvalidOperationException("Allocated tree element is not empty.");
42     }
43 }
44
45 public void Free(TElement node)
46 {
47     while (!EqualityComparer.Equals(_allocated, One) && IsEmpty(node))
48     {
49         var lastNode = Arithmetic.Decrement(_allocated);
50         if (EqualityComparer.Equals(lastNode, node))
51         {
52             _allocated = lastNode;
53             node = Arithmetic.Decrement(node);
54         }
55         else
56         {
57             return;
58         }
59     }
60 }
61
62 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) =>
87         ↪ GetElement(node).Balance = value;
88
89     protected override void SetLeft(TElement node, TElement left) => GetElement(node).Left =
90         ↪ left;
91
92     protected override void SetLeftIsChild(TElement node, bool value) =>
93         ↪ GetElement(node).LeftIsChild = value;
94
95     protected override void SetRight(TElement node, TElement right) =>
96         ↪ GetElement(node).Right = right;
97
98     protected override void SetRightIsChild(TElement node, bool value) =>
99         ↪ GetElement(node).RightIsChild = value;
100
101     protected override void SetSize(TElement node, TElement size) => GetElement(node).Size =
102         ↪ size;
103
104     private ref TreeElement GetElement(TElement node) => ref
105         ↪ _elements[UncheckedConverter<TElement, long>.Default.Convert(node)];
106 }

```

## 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
12             ↪ SizedBinaryTreeMethodsBase<TElement> tree, Func<TElement> allocate, Action<TElement>
13             ↪ free, ref TElement root, Func<TElement> treeCount, int maximumOperationsPerCycle)
14         {
15             for (var N = 1; N < maximumOperationsPerCycle; N++)
16             {
17                 var currentCount = 0;
18                 for (var i = 0; i < N; i++)
19                 {
20                     var node = allocate();
21                     tree.Attach(ref root, node);
22                     currentCount++;
23                     Assert.Equal(currentCount, (int)UncheckedConverter<TElement,
24                         ↪ int>.Default.Convert(treeCount()));
25                 }
26                 for (var i = 1; i <= N; i++)
27                 {
28                     TElement node = UncheckedConverter<int, TElement>.Default.Convert(i);
29                     if (tree.Contains(node, root))
30                     {
31                         tree.Detach(ref root, node);
32                         free(node);
33                         currentCount--;
34                         Assert.Equal(currentCount, (int)UncheckedConverter<TElement,
35                             ↪ int>.Default.Convert(treeCount()));
36                     }
37                 }
38             }
39         }
40
41         public static void TestMultipleRandomCreationsAndDeletions<TElement>(this
42             ↪ SizedBinaryTreeMethodsBase<TElement> tree, ref TElement root, Func<TElement>
43             ↪ treeCount, int maximumOperationsPerCycle)
44         {
45             var random = new System.Random(0);
46             var added = new HashSet<TElement>();
47             var currentCount = 0;
48             for (var N = 1; N < maximumOperationsPerCycle; N++)
49             {
50                 for (var i = 0; i < N; i++)
51                 {
52                     var node = UncheckedConverter<int, TElement>.Default.Convert(random.Next(1,
53                         ↪ N));
54                     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(recursionlessSizeBalancedTree.Allocate, recursionlessSizeBalancedTree.Free, ref
                    ↪ recursionlessSizeBalancedTree.Root, () => recursionlessSizeBalancedTree.Count,
                    ↪ _n);
14         }
15
16         [Fact]
17         public static void SizeBalancedTreeMultipleAttachAndDetachTest()
18         {
19             var sizeBalancedTree = new SizeBalancedTree<uint>(10000);
20             sizeBalancedTree.TestMultipleCreationsAndDeletions(sizeBalancedTree.Allocate,
                    ↪ sizeBalancedTree.Free, ref sizeBalancedTree.Root, () => sizeBalancedTree.Count,
                    ↪ _n);
21         }
22
23         [Fact]
24         public static void SizedAndThreadedAVLBalancedTreeMultipleAttachAndDetachTest()
25         {
26             var avlTree = new SizedAndThreadedAVLBalancedTree<uint>(10000);
27             avlTree.TestMultipleCreationsAndDeletions(avlTree.Allocate, avlTree.Free, ref
                    ↪ avlTree.Root, () => avlTree.Count, _n);
28         }
29
30         [Fact]
31         public static void RecursionlessSizeBalancedTreeMultipleRandomAttachAndDetachTest()
32         {
33             var recursionlessSizeBalancedTree = new RecursionlessSizeBalancedTree<uint>(10000);
34             recursionlessSizeBalancedTree.TestMultipleRandomCreationsAndDeletions(ref
                    ↪ recursionlessSizeBalancedTree.Root, () => recursionlessSizeBalancedTree.Count,
                    ↪ _n);
35         }
36
37         [Fact]
38         public static void SizeBalancedTreeMultipleRandomAttachAndDetachTest()
39         {
40             var sizeBalancedTree = new SizeBalancedTree<uint>(10000);
41             sizeBalancedTree.TestMultipleRandomCreationsAndDeletions(ref sizeBalancedTree.Root,
                    ↪ () => sizeBalancedTree.Count, _n);
42         }
43
44         [Fact]

```

```
45     public static void SizedAndThreadedAVLBalancedTreeMultipleRandomAttachAndDetachTest()
46     {
47         var avlTree = new SizedAndThreadedAVLBalancedTree<uint>(10000);
48         avlTree.TestMultipleRandomCreationsAndDeletions(ref avlTree.Root, () =>
49             ↪ avlTree.Count, _n);
50     }
51 }
```

## Index

- ./csharp/Platform.Collections.Methods.Tests/RecursionlessSizeBalancedTree.cs, 25
- ./csharp/Platform.Collections.Methods.Tests/SizeBalancedTree.cs, 26
- ./csharp/Platform.Collections.Methods.Tests/SizedAndThreadedAVLBalancedTree.cs, 27
- ./csharp/Platform.Collections.Methods.Tests/TestExtensions.cs, 29
- ./csharp/Platform.Collections.Methods.Tests/TreesTests.cs, 30
- ./csharp/Platform.Collections.Methods/GenericCollectionMethodsBase.cs, 1
- ./csharp/Platform.Collections.Methods/Lists/AbsoluteCircularDoublyLinkedListMethods.cs, 2
- ./csharp/Platform.Collections.Methods/Lists/AbsoluteDoublyLinkedListMethodsBase.cs, 3
- ./csharp/Platform.Collections.Methods/Lists/AbsoluteOpenDoublyLinkedListMethods.cs, 4
- ./csharp/Platform.Collections.Methods/Lists/DoublyLinkedListMethodsBase.cs, 5
- ./csharp/Platform.Collections.Methods/Lists/RelativeCircularDoublyLinkedListMethods.cs, 5
- ./csharp/Platform.Collections.Methods/Lists/RelativeDoublyLinkedListMethodsBase.cs, 6
- ./csharp/Platform.Collections.Methods/Lists/RelativeOpenDoublyLinkedListMethods.cs, 7
- ./csharp/Platform.Collections.Methods/Trees/RecursionlessSizeBalancedTreeMethods.cs, 8
- ./csharp/Platform.Collections.Methods/Trees/SizeBalancedTreeMethods.cs, 11
- ./csharp/Platform.Collections.Methods/Trees/SizedAndThreadedAVLBalancedTreeMethods.cs, 13
- ./csharp/Platform.Collections.Methods/Trees/SizedBinaryTreeMethodsBase.cs, 21